View-Dependent Image-Based Techniques for Fast Rendering of Complex Environments

Jonathan Ward Shade

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

2004

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Jonathan Ward Shade

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Co-Chairs of Supervisory Committee:

_____

Brian L. Curless

_____

Michael F. Cohen

Reading Committee:

_____

Brian L. Curless

_____

Michael F. Cohen

_____

Steven M. Seitz

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with “fair use” as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Proquest Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, to whom the author has granted “the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform.”

Signature_____

Date_____

University of Washington

Abstract

View-Dependent Image-Based Techniques for Fast Rendering of Complex Environments

by Jonathan Ward Shade

Co-Chairs of Supervisory Committee:

Associate Professor Brian L. Curless
Computer Science and Engineering

Affiliate Professor Michael F. Cohen
Computer Science and Engineering

One of the primary goals of computer graphics has been the creation of interactive photorealistic imagery. Unfortunately, the dual objectives of interactivity and photorealism are at odds with each other. Image-based rendering provides a method whereby an offline, computationally intensive image synthesis or computer vision algorithm can be paired with an online, interactive image synthesis algorithm to produce real-time renderings of complex scenes. This dissertation presents three novel view-dependent image-based representations that can be used to accelerate rendering of complex, naturalistic scenes. Hierarchical Image Caching automatically and dynamically caches parts of a scene into images that are rendered in place of geometry, at a much lower computational cost. Layered Depth Images efficiently render complex scenes in software using an image-order warp of pixels with associated depth. Storing multiple depth pixels along each ray of an image, a scene is sampled into a sparse, compact data structure in a view-dependent fashion. Tiling Layered Depth Images uses Layered Depth Images as a basic modeling primitive to create expansive

renderings of richly textured terrains. A novel stochastic tiling algorithm is given that guarantees non-periodic tilings of the plane while using a small set of square tiles and a simple construction procedure. Lastly, all image-based representations can be cast as approximations to the plenoptic function, a seven dimensional function describing all of the light in a scene. A novel classification is presented that relates image-based representations, such as the three outlined above, through four common techniques used to reduce the complexity of the plenoptic function.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I have had the pleasure of collaborating with many people over the years: Michael Bailey, Fran Berman, Michael Cohen, Brian Curless, Tony DeRose, Oliver Deussen, Steven “Shlomo” Gortler, Li-wei He, Stefan Hiller, Hugues Hoppe, Peter Kochevar, Marc Levoy, Dani Lischinski, Don Mitchell, Szymon Rusinkiewicz, David Salesin, John Snyder, Richard Szeliski, Len Wanger, and all of the folks on the Digital Michelangelo project. Kevin Bacon, eat your heart out.

In particular, I would like to thank Michael Bailey for giving me a start in computer graphics and being my first mentor, and Michael Cohen for advising, prodding, cajoling, and showing me how to do research. Of course, I never would have made it without my family, Mom, Kathy, Daniel and Sean. Thank you. But, most of all, I’d like to thank Erin for everything she’s done for me. Who, above all others, has supported me, encouraged me, and believed in me. Erin, thank you.

Chapter 1

INTRODUCTION

One of the primary goals of computer graphics is the creation of interactive photorealistic imagery. Unfortunately, the dual objectives of interactivity and photorealism are at odds with each other. The process of creating physics-based photorealistic images has traditionally involved the simulation of the propagation of light through an environment. This requires modeling the geometry of the objects in the environment, the properties of the materials that constitute the geometry, the light sources that emit energy into the environment, and the cameras that record the light. A computationally expensive light transport algorithm, such as ray tracing [90] or radiosity [31, 15, 66] is then used to compute the distribution of light energy throughout the scene. Modeling a scene that resembles the real world with this process is exceedingly difficult because the complexity of the geometry of the real world is so great that traditional geometric representations result in data sets so large that it is impractical to build a computer with enough memory to hold them, and no simulation of light transport will finish in a reasonable amount of time.

At the other end of the spectrum, interactive graphics has focused on hardware implementations of rendering algorithms. In order to achieve interactive rates these systems must make sacrifices that greatly reduce the realism of a scene: low level of geometric detail, simplified shadows, simplified material properties, and local instead of global illumination. While the complexity of scenes that can be rendered by interactive systems is continually increasing, they are still a long way from producing photorealistic images.

Since the goal is producing photorealistic images, in the end, what matters is the final

Figure 1.1: Parallax in the real world. The eight photographs in this figure and the next were captured while translating a camera from left-to-right.

result of an image synthesis algorithm: the light that enters the eye of the observer. If there were a way of capturing, storing, and re-rendering light, then a computationally expensive image-synthesis algorithm could be run once as a pre-process to create a light-based data structure that is used by an inexpensive, interactive run-time image synthesis algorithm. This observation is the basis of the field of image-based rendering.

1.1 Image-based rendering

In image-based rendering, images (samples of light) are used as the fundamental modeling primitive. Rendering from this sampled representation consists of reprojecting and interpolating the samples of light, a process that can be computed at interactive rates. One of the consequences of using images as a modeling primitive is that image-based algorithms are, for the most part, agnostic with respect to how the images are made. Because of this, images of the real world work just as well as images of synthetic environments. Thus, image-based rendering provides a framework for producing interactive photorealistic renderings of both real and synthetic environments.

Changes in the appearance of a three dimensional scene can be grouped into three broad categories: changes in shape, changes in shading, and changes due to motion of an observer.

Figure 1.2: Parallax in the real world continued.

The work in this dissertation focusses solely on motion of an observer. Shading is assumed to be static: the color of an object does not depend on the position of an observer and does not change over time. The shape of a scene is also static: the objects in the scene do not move or deform over time. Motion of an observer induces parallax: the change in apparent position of one object with respect to another. Parallax is an essential cue to the three dimensional character of a scene. Reproducing the parallax inherent in complex, naturalistic, scenes using image-based rendering is the focus of the work in this dissertation.

In the eight photographs in Figure 1.1 and Figure 1.2, parallax is evident in two ways: between the tree and the mountain, and among the branches and leaves within the tree. While the appearance of the tree changes a great deal, the mountain changes very little across the sequence of images. In general, the further an object is from an observer, the less its appearance will change due to movement of the observer. This observation is the foundation of the first system presented in this dissertation, Hierarchical Image Caching. This system is designed to accelerate rendering of complex scenes using graphics hardware. An image is much faster to render than the polygons it contains. Rendering of scenes like this one can be accelerated by caching rendered images of far-away objects that are re-rendered in place of the geometrical model. Hierarchical Image Caching provides an automatic method which determines the parts of a scene that can be cached into images,

allowing graphics hardware resources to be concentrated on rendering rapidly changing objects such as the tree.

Image caches project geometry onto a plane, throwing away any knowledge of the relative depth of objects in a scene. Because of this, image caches cannot reproduce the parallax in rapidly changing objects. By augmenting images with per-pixel depth parallax can be preserved, allowing rendering of near-by objects to be accelerated using images. The second system in this dissertation, Layered Depth Images, takes this a step further by storing multiple pixels-with-depth at each location in an image.

There are many ways to take advantage of the use of images as a modeling primitive. Chapter 2 gives a detailed overview of the published image-based rendering systems. By caching samples of light, a great deal of effort can be spent creating detailed source images that are warped and re-rendered at run-time to create much more realistic images than can be attained using conventional approximations employed in real-time systems.

1.2 Contributions

This dissertation describes the work done while the author constructed three image-based rendering systems: Hierarchical Image Caching, Layered Depth Images, and Tiling Layered Depth Images. There is a chapter dedicated to each system, and each chapter draws heavily from a related paper [81, 80, 16]. The contributions of this dissertation are:

A novel classification of image-based representations. This chapter has introduced the concept of using image-based representations in real-time rendering. In a more general context, image-based representations provide an approximation of the plenoptic function: all of the light passing through all points in space at all times. Densely sampling this six-dimensional function produces data sets that quickly become unmanageably large. Image-based representations use a common set of techniques to pare down the plenoptic function: taking a subset of the function, reducing the dimension of the function by throwing away a degree of freedom, sparsely sampling a dimension, and finding and eliminating redundancy

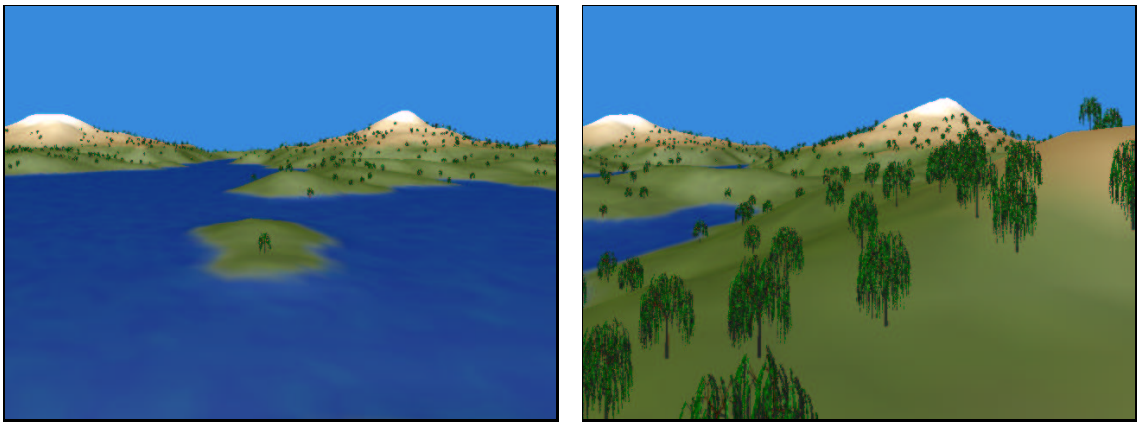


Figure 1.3: Island rendered using hierarchical image caching.

in the function. Chapter 2 reviews the literature of image-based representations and shows how each representation can be derived using combinations of these techniques.

Hierarchical Image Caching: a dynamic, hierarchical image-based representation. The first system presented accelerates the rendering of complex scenes by dynamically caching images of portions of the scene. As a preprocess, a binary space partitioning is used to divide a scene into portions with equal numbers of triangles. At run-time, the portions of the scene far from the eye are rendered and cached in an image that is then reused the next time the scene is rendered. To gain a further benefit, the caches are combined hierarchically according to the space partitioning. This system, while restricted to static scenes, can be used in scenes with arbitrarily complex lighting. Since the sampled representation is created dynamically, the trade-off between accurate evaluation of the rendering equation and speed can be specified by a user-controllable error metric. Figure 1.3 shows two renderings using hierarchical image caching.

An error metric for controlling updates to hierarchical image caches. In conjunction with the hierarchical image cache representation, an error-driven cost-benefit analysis is presented that allows a user to control how often image caches are updated. Using a screen-

Figure 1.4: Example renderings of LDIs.

space metric, we conservatively guarantee that the perspective projection of the contents of an image cache are within a user-specified number of pixels of where they should be. This metric is indirect in that it only measures the distortion of the surfaces captured in the image cache; it does not account for changes in the occlusion relationships among the objects in an image cache.

Layered Depth Images: a novel image-based representation. The Layered Depth Image (LDI) is a novel image-based representation that can accurately represent complex objects that have a high degree of depth complexity. An ordinary image stores one pixel of color per ray of the image. A depth image stores depth at each pixel in addition to color. A Layered Depth Image stores multiple pixels-with-depth per ray of the image. McMillan's occlusion-compatible warp ordering of depth images [57] can be adapted to LDIs, facilitating real-time rendering in software. Figure 1.4 shows two example renderings of LDIs.

Figure 1.5: Terrain textured with sunflowers.

Tiling Layered Depth Images: A system for rendering terrains with homogeneous three-dimensional textures. The third system described addresses a long-standing problem in computer graphics: modeling and rendering three-dimensional textures in real-time. While the specific application demonstrated is real-time rendering of textured terrains, the solution provided is applicable to any situation in which a three-dimensional texture is applied to a two-dimensional domain. There are two key contributions in this work: a multi-view, multiresolution formulation of layered depth images, and a new algorithm for computing non-periodic tilings of the plane.

A multi-view, multiresolution, formulation of LDIs. LDIs provide high-fidelty renderings of complex scenes when the novel view is close to the view from which the LDI was created. This works very well for producing panoramic outward-looking views of a complex scene. A multi-view LDI is a set of inward-looking LDIs that surround an object. Creating multi-view LDIs at multiple resolutions allows for fast, high quality renderings of an object from many directions and many distances. Chapter 5 shows how this representation can used as the basic building block of a three-dimensional texture draped over an expansive terrain. Figure 1.5 shows three renderings of a terrain textured with sunflowers.

A novel algorithm for computing tilings of the plane. Chapter 5 describes a novel algorithm for computing non-periodic tilings of the plane. A constructive algorithm is given for computing tilings of the plane using a set of eight Wang tiles. While the set of Wang tiles presented is not an aperiodic set, the stochastic procedure given for computing tilings ensures that, in practice, a non-periodic tiling is always found.

1.3 Overview of the dissertation

The next chapter gives a review of image-based rendering systems and places these systems in a novel classification. Chapter 3 describes the hierarchical image caching representation. Chapter 4 describes the LDI representation. Chapter 5 introduces multi-view multiresolution layered depth images and a novel two-dimensional tiling scheme that produces non-periodic tilings of the plane. Finally, Chapter 6 concludes with some final thoughts and suggestions for future work.

Chapter 2

APPROXIMATING THE PLENOPTIC FUNCTION

Image-based models are sampled representations of the *plenoptic function*. As defined by Adelson and Bergen [1], the plenoptic function describes all of the light passing through all points in space at all moments in time. This function has seven dimensions: three for the position of a point in space, two for all of the directions from which a ray of light can pass through a point, one for wavelength, and one for time. The goal of image-based rendering is to capture, store, and reconstruct the light that passes through a set of points in such a way that a human looking at a computer-driven display sees the same thing they would have seen had they been in the environment that was captured.

The process of image-based rendering can be broken down into two major stages as depicted in Figure 2.1: sampling and reconstruction. Sampling is typically conducted in one of two ways. In systems that use pictures of the real world, a digital camera or video-camera is used to record the environment of interest. This data is then resampled to fit into the organization of the chosen image-based representation. If a synthetic scene is being sampled, an image-synthesis scheme, such as ray tracing, can be used to sample the scene directly into the intended representation, thus bypassing the resampling step. In both cases the intensity of light is measured in terms of radiance: power per unit projected area per unit solid angle. This is the quantity captured by sensing devices, and it is directly related to the brightness of a pixel produced by an image synthesis algortihm. A complete discussion of radiance and its use in computer graphics can be found in [17]. The classification presented in this chapter is concerned with the structure of image-based representations, not how they are created. Therefore, the mechanics of sampling is not considered in the rest of the chapter.

Figure 2.1: Image-based rendering pipeline.

Sampling is reconstruction-driven: enough samples are taken to guarantee some measure of quality at reconstruction time. There are a wide variety of techniques used to reconstruct the plenoptic function from a sampled representation, and Section 2.1 discusses the details of reconstruction on a system-by-sytem basis. A simple example will illustrate the primary issue: a trade-off between storage space, the range of possible views, and reconstruction quality. The first tradeoff made by all of the systems reviewed here is to reduce the plenoptic function from seven dimensions to five by fixing time to a single moment and wavelength to three colors. Suppose one wanted to capture this 5D plenoptic function for an average graduate student office. The office is 5 meters wide and 7 meters long. The plenoptic function is sampled by capturing a panoramic image of the office at the nodes of a regular grid with one centimeter spacing. To simplify the task we will drop the 5D plenoptic function to just 4D by restricting the captured light to only that passing through a plane at the average height of an observer. To reconstruct the plenoptic function, a different image is displayed to each eye of an observer (human eyes are spaced about three inches apart). The observer is allowed to walk around and turn (but not tilt or nod) their head. To estimate the amount of storage needed for this data set assume that each eye of the observer will get an image that is 256 pixels wide by 256 pixels tall with a field of view of 90 degrees. The panoramic image thus needs to be 1000 pixels wide. Since the head can't tilt, the vertical resolution of the panoramic image can be limited to 256 pixels. A grid with centimeter spacing over a room 5 meters wide and 7 meters long has 700x500 grid points, and thus 350,000 panoramic images. Capturing 24-bit images means each panoramic image requires 768KB of space. Storing 350,000 images using the JPEG format and assuming a

Figure 2.2: Planar perspective image.

10:1 compression ratio requires 26 GB of space. Even when restricting the view to a single height, an unmanageable amount of data is required.

Sampled representations are, by nature, a trade-off in favor of increased size for less computation. Image-based representations are precomputed evaluations of the the plenoptic function stored in a table. As the example above illustrates, there is a direct correlation between the dimension of a function and the size of the domain over which it is represented, and the amount of space required to store a sampled representation of it. As a consequence, all image-based models make simplifying assumptions in order to reduce the size of the representation used to approximate the plenoptic function. I have identified four techniques image-based systems have used to simplify the plenoptic function:

Taking a subset of the plenoptic function. A sampled representation must have a finite domain. In this respect, all image-based representations sample a subset of the plenoptic function. As shown by the office example, representing the plenoptic function in even a small environment can require a huge data set. Because of this, image-based representations that uniformly sample the plenoptic function are typically limited to a very small spatial domain.

Figure 2.3: Cylindrical image.

Reducing the dimension of the plenoptic function. As in the office example, throwing away a degree of freedom in choosing samples reduces the dimension of the plenoptic function. We call such a representation a reduction of the plenoptic function. The size of the representation decreases at the cost of reducing by a degree of freedom the space of views that can be reconstructed. In the office example, restricting the panoramic images to lie on a plane reduces the degrees of freedom in position from three to two. This representation is therefore a 4D reduction of the 5D plenoptic function.

Sparsely sampling a dimension. Sampling a dimension very coarsely can be thought of as producing a set of representations of one dimension lower. Applying this to the office example, the grid of panoramic images can be sampled only inside each cubicle within the office. Movement between cubicles is handled by "teleporting" the viewer, like in Quicktime VR [12]. Reconstruction is continuous at each of the sampled locations, but not when moving between them. Using this technique, large environments can be represented with collections of densely sampled data sets placed at strategic points. Representations that use this strategy are designated with a fractional dimension. For instance, the sparse sampling of the office would be a 3.5D representation.

Figure 2.4: Spherical image.

Eliminating redundancy. Another way to reduce the amount of data in the plenoptic function is to find and eliminate redundancy in the function. The *lumigraph* [32] and *light field* [48] representations do this by exploiting the *ray law*: the radiance along a ray remains unchanged in the absence of occluding geometry or participating media. For any two points along such a ray, the radiance leaving one point must equal the radiance arriving at the other point. By assuming the space around the viewer is free of geometry and participating media, the plenoptic function can be described with a 4D function without reducing the range of views that can be reconstructed. This technique is elegant because the assumption of constancy is physics-based and correct. No information is lost in reducing the plenoptic function from five to four dimensions.

This chapter presents a classification of image-based representations in two ways. First, we review the literature and show how these four techniques have been applied. Second, we take a systematic approach to applying these four techniques. As Tables 2.2 and 2.3 show, a great variety of representations can be made from the singular 0D representation, point samples, through 1D, 2D, 2.5D, 3D, 3.5D, and 4D representations.

Figure 2.5: Cubical image.

2.1 Overview of Representations

There has been a great deal of work in image-based modeling and rendering. This section gives an overview of some of the significant papers.

2.1.1 Images

There are many ways to approximate the plenoptic function. The most familiar is the 2D image. The most general definition of an image is: every ray of light that passes through a point in space (the center of projection of the image). Choosing a single point in space fixes three of the degrees of freedom of the 5D plenoptic function. The remaining two parameters (the directions of the rays of incoming light) can be defined in a variety of ways. Four common choices are: planar, cylindrical, spherical, and cubical. Figures 2.2 through 2.5 show examples of each type of image. The cylindrical, spherical and cubical projections are often called panoramic images. The cubical projection has become synonymous with

Figure 2.6: Planar orthographic (top) and line-perspective (bottom) images.

the term *environment map* in the computer graphics literature.

Sampling the plenoptic function from just a single point (though over all directions), not only greatly reduces the size of the function sampled, but also the degrees of freedom in reconstruction. Strictly speaking, there is only one correct reconstruction possible, viewing the image from its center of projection. If the image is panoramic the viewer may turn and look in any direction, but they can't translate from their current position. Sections 2.1.2 and 2.1.3 discuss representations that warp images to create approximate reconstructions near the center of projection, allowing the viewer to move a short distance.

Standard images have a single center of projection. In some instances, it is advanta-

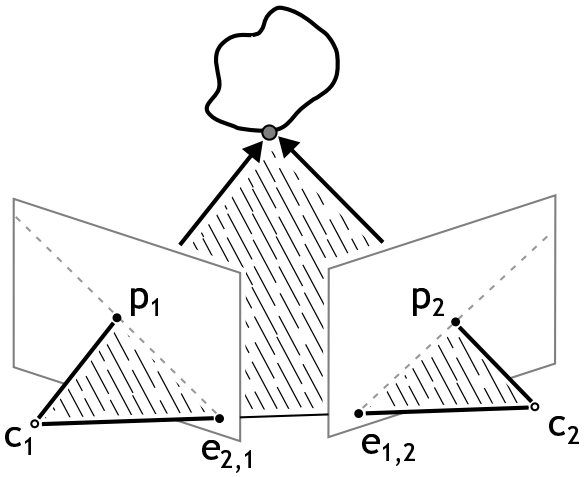
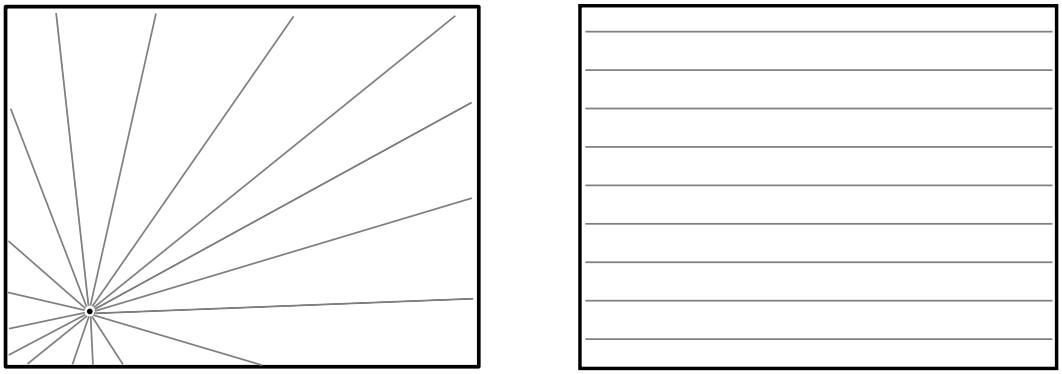


Figure 2.7: Epipolar geometry. The centers of projection of two cameras, c_1 and c_2, and a point on an object, p, form an epipolar plane. The image of one camera's center in the other camera's image plane is called an epipole. The epipole $e_{2,1}$ ($e_{1,2}$) is the projection of c_2 (c_1) onto the image plane of c_1 (c_2). The intersection of an epipolar plane and the image plane of a camera is an epipolar line. The projection into a camera of any point lying on a particular epipolar plane in the scene must lie on the corresponding epipolar line. The points p_1 and p_2 show two such projections.

geous to use more than one center of projection within a single image. Multiperspective images (MPIs) can be made in many different ways. A familiar example is the planar orthographic image (see Figure 2.6, top). An orthographic image can be thought of in two ways. The first is that it is a perspective image with a center of projection that is infinitely far away. The second is that it is a multiperspective image where each pixel has associated with it a unique camera such that the pixel represents the ray going through the center of the camera's image plane. Along this ray there is no perspective foreshortening. Another way to create a multiperspective image is to use a line of cameras all facing the same direction. This MPI is created by taking the central vertical line of pixels from each camera (see Figure 2.6, bottom). This vertical-slit camera structure is similar to images used in a lenticular display [74]. The first system reviewed uses the structure of epipolar geometry to build multiperspective images built not from vertical slits, but from horizontal slits.



(a) (b)

Figure 2.8: Epipolar lines. The intersection of an epipolar plane with the image plane of a camera is an epipolar line. (a) A fan of epipolar lines. (b) When the two cameras are parallel the epipoles project to points at infinity, resulting in parallel epipolar lines.

Epipolar geometry defines the relationship between two pinhole cameras with centers of projection c_1 and c_2, and a point in the scene, as illustrated in Figure 2.7. The image plane of each camera shows two projected points: p is the projection of the point in the scene, and e, the *epipole*, is the projection of the other camera's center of projection. The plane formed by the point in the scene and the two camera centers is called the *epipolar plane*. The intersection of this plane with the image plane of each of the cameras forms an *epipolar line* in each image plane. Epipolar lines have been used to automatically compute point correspondences between images [26], to automatically compute per-pixel depth [9], and as a tool to resolve visibility when using 3D image warping [57].

An interesting property of epipolar lines is that for a given pair of cameras, the set of epipolar lines for all points in the scene radiate in a fan-like configuration about each epipole, as shown in Figure 2.8(a). If the two cameras are parallel (positioned adjacent to one another and looking in the same direction), then the epipoles project to infinity and the epipolar lines become parallel, as shown in Figure 2.8(b).

Figure 2.9 shows an example of a set of input images and the Epipolar Plane Images (EPIs) created from them. An EPI is derived by translating a camera parallel to its horizontal scanline direction, extracting the same scanline from each image, and then stacking the

extracted scanlines bottom to top to form the EPI. Thus, there are the same number of EPIs as there are horizontal lines of resolution in the camera that captured the scene, and the EPI vertical resolution is equal to the number of original images. This representation is a 3D reduction of the plenoptic function: a set of 2D images taken along a linear path. The space of reconstruction views is limited to those that lie on the line of the input cameras and look in the same direction as the original cameras.

EPIs were originally created by Bolles et al. [9] to automatically extract depth in stereo pairs of images, but they can also be used to display view-dependent stereoscopic reconstructions of a real scene. The representation built by Katayama et al. [42] uses the parallel-camera configuration. A series of images of a scene was acquired by translating a camera horizontally along a rigid slide stage with the orientation of the camera fixed to be perpendicular to the direction of motion. After the images were captured, they were factored into EPIs.

EPIs capture the parallax in a scene: the lines in an EPI trace the movement, across the image plane, of points in the scene. If a point in the scene is moving very quickly across the field of view of a camera in motion, the projection of the point will create a line on the image plane with a steep slope (i.e. more horizontal than vertical). As a consequence, the slope of a line in an EPI is directly proportional to the depth of the point in the scene that traces out that line [9]. Katayama used the slope of these lines to create interpolated novel views of the scene that lie between the acquired images. Since each horizontal line in an EPI represents a different horizontal camera, a novel view can be synthesized by synthesizing an intermediate horizontal line in an EPI that correctly interpolates the trace lines that cross it.

Wood et al. [92] used multiperspective images to create distorted panoramas for use in cel animation. Cel animation is fundamentally a 2D process in which a number of layers of paintings are composited in a back-to-front order to compose a pseudo 3D scene. Translating the layers at different rates produces an approximation to the parallax that would be seen in a true 3D scene. Long sweeping camera motions can be produced by panning a

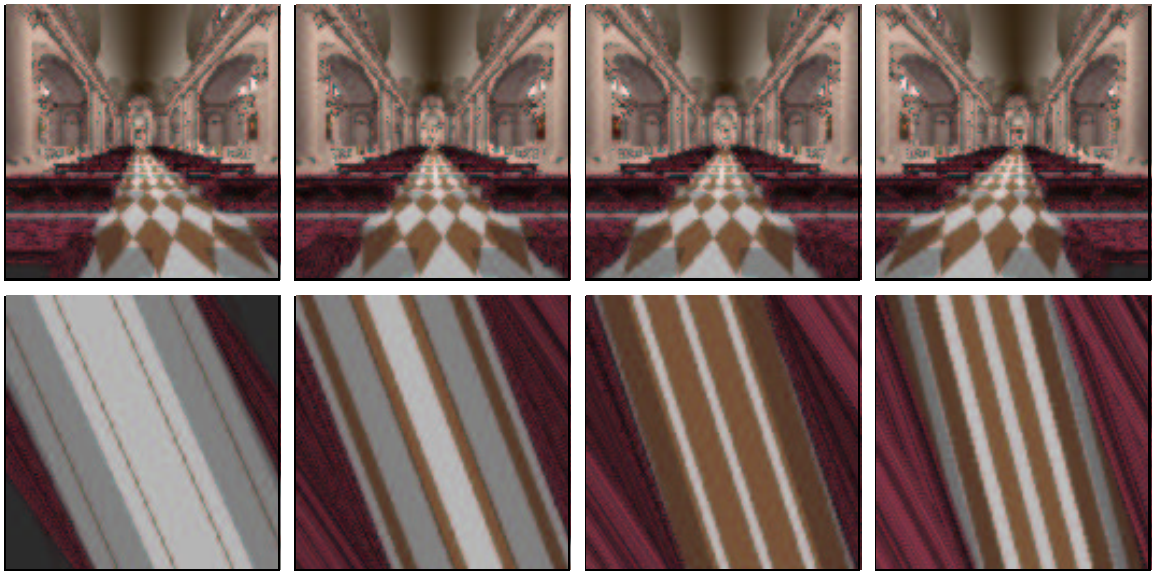


Figure 2.9: Epipolar plane images. The top row shows 4 out of a set of 128 images taken as a camera translates left-to-right. The bottom row shows 4 epipolar plane images (EPIs). Each EPI is constructed by selecting the same scanline from each image in a sequence and stacking them top to bottom to form a new image. The four EPIs shown were built from scanlines near the bottom of the input sequence shown on the top row.

camera over a large distorted panorama of a scene. The panorama shown in Figure 2.10 portrays a sweeping camera motion simulating the view from a helicopter as it rotates and descends into a city. For every frame of an animation, a small window is cropped from the source panorama. As the window is moved around the panorama a convincing 3D animation is produced. In Figure 2.10, the illusion that the camera is rotating and descending towards the city is created by rotating a cropped window in a counter-clockwise fashion starting in the upper-right corner of the panorama.

In order to maintain the illusion, implausible and unintuitive perspectives often have to be employed. For instance, in Figure 2.10 the yellow-shaded building is seen twice, with different perspectives. The work presented by Wood et al. sought to automate this process by creating a rough draft of the warped panorama using 3D geometric models. The computer-generated panorama was then used as a guide by an artist who painted the

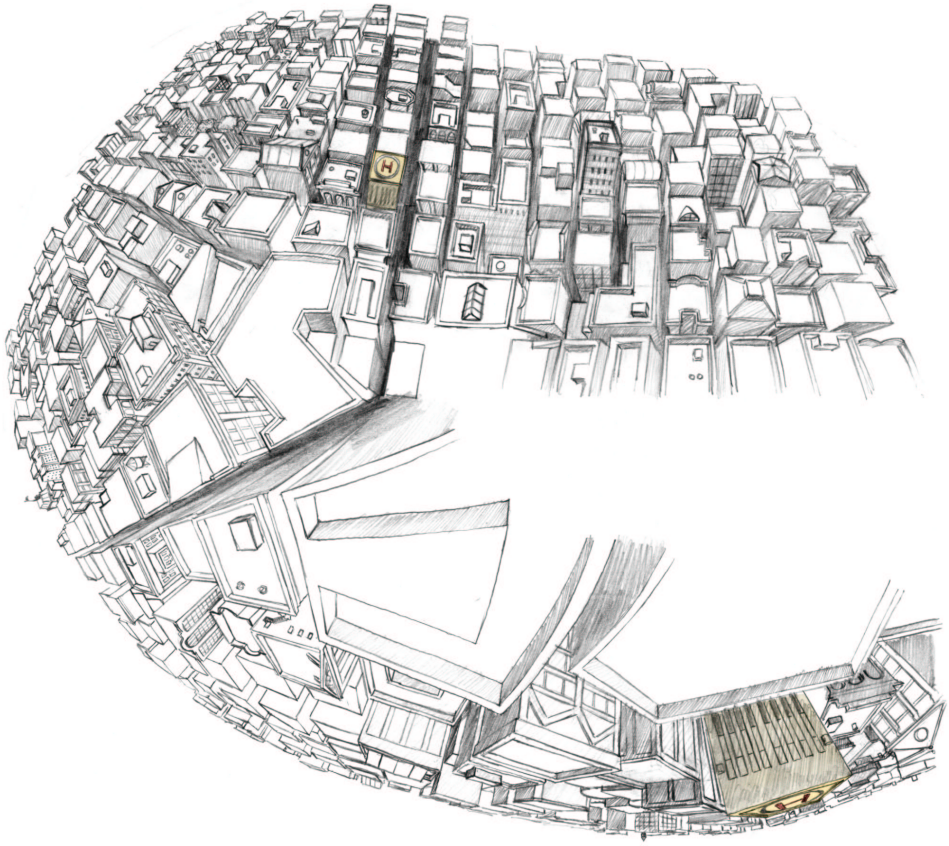


Figure 2.10: A multiperspective panorama. From Wood et al. [92].

final panorama by hand. The warped panorama is a multiperspective image created using a series of vertical wedges from the central areas of a camera placed along a space curve. Since the source cameras are not constrained to linear horizontal motion, taking a single vertical slit from each image is not sufficient. The freedom of the cameras to tilt and pan means a wedge of pixels is needed from each camera to ensure that there are no holes in the MPI.

A standard panoramic image is a 2D reduction of the plenoptic function in which the center of projection is fixed and many directions are sampled. Multi-perspective panoramas are a 2D reduction of the plenoptic function in which the center of projection is allowed to move and only a small subset of the directions are sampled for any one center of projection.

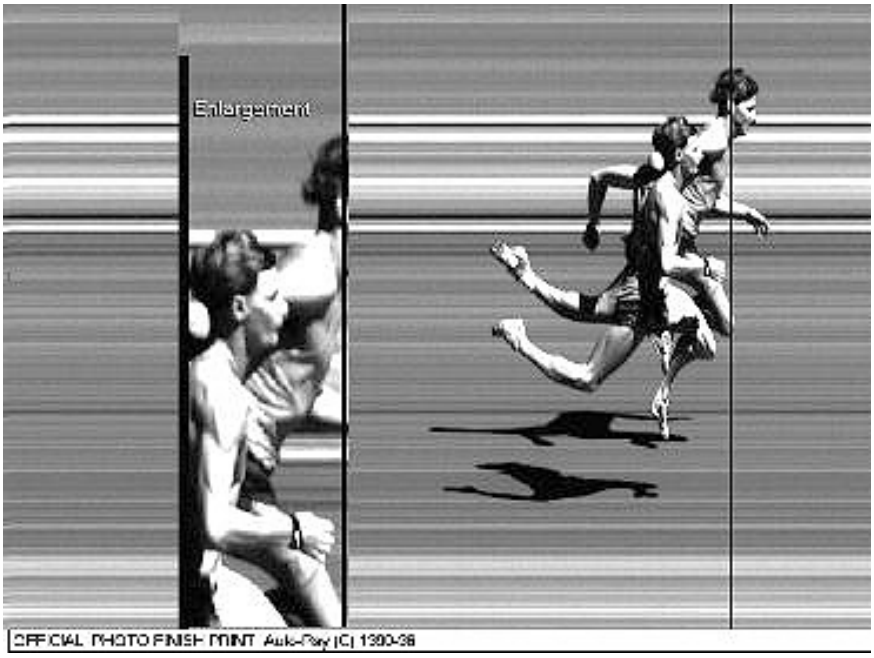


Figure 2.11: A single-perspective multi-time image. Used by permission of the Auto-Ray Corporation.

Even though reconstruction is restricted to panning a 2D window across the panorama, possible visual effects include trucking, dollying, and zooming in addition to panning.

In related work, Glassner [30] proposes using multiperspective panoramas to facilitate storytelling. Instead of viewing the panorama through a window that pans across the image, the panorama is viewed as a whole. Using this technique a scene can be viewed from multiple perspectives simultaneously.

A related type of image is what might be called a single-perspective multi-time image, by keeping the camera fixed and letting time vary. Figure 2.11 shows an example of such an image created by a photo finish camera. In this image, a vertical-slit camera is fixed in position and time is allowed to run forward. The same vertical slice of space is represented at each column of the image but at a different time, causing the background to appear smeared across the image. If an object moves across the camera's field of view at a constant rate, the image of the object appears as it would in a line-perspective image. If an animating

object moves past the camera (like the runners in Figure 2.11), the parts of the object moving quickly appear slender, while the parts moving slowly appear wider. This is evident in the sickle shape of the runners' legs. Since time has been included, this type of image is a 2D reduction of a 6D plenoptic function. Reconstruction is straightforward because, like Glassner's multiperspective panoramas, they are meant to be viewed as a whole.

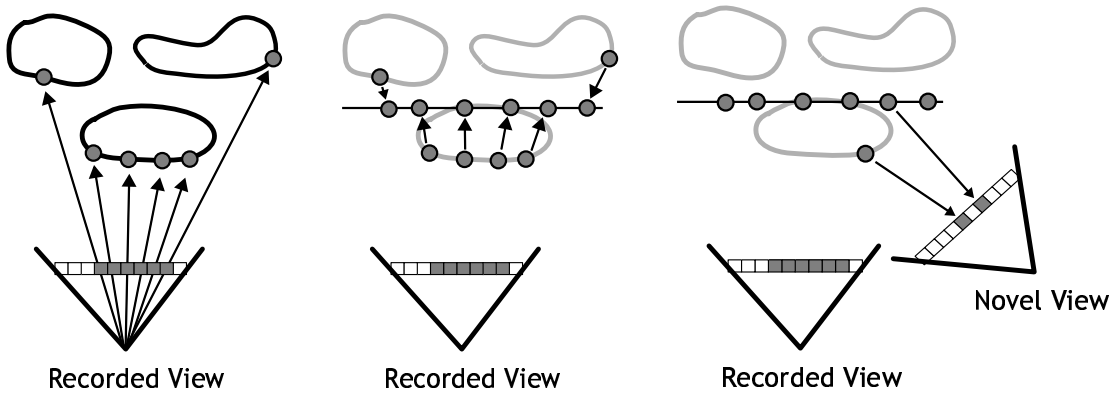
Lastly, turning a 2D image inside out and considering the radiance flowing through a point as leaving rather than arriving describes a directionally-varying point light source (where an image would be a point light sink). Since the term point light source is already used in computer graphics to mean something more specific, we'll use the notation of Wood et al. [91] and call these inverted images lumispheres. Images record "incoming" radiance and lumispheres record "outgoing" radiance.

2.1.2 Collections of Images

The second group of representations use a set of 2D images (or photographs) to model an environment.

Movie Maps [50] is a system that allows virtual walkthroughs of a small town by replaying predetermined video sequences stored on a videodisc. The video sequences follows the edges of a graph overlayed on the streets of Aspen, Colorado. At the nodes in the graph, the user decides which subsequent path (edge in the graph) to follow, and the appropriate video sequence is played. Since Movie Maps use a sequence of 2D images, it is a 3D representation of the plenoptic function. However, it is a very restricted 3D representation: the view position can not deviate from that of the video camera used to record the images. Creating a set of movies over a predetermined set of paths is a form of sparsely sampling a degree of freedom. So, this representation lies somewhere between 3D and 4D; it is a 3.5D representation.

Shum and He [82] built a system called Concentric Mosaics that captures a dense set of images that are tangential to a circular path. A viewer is allowed to roam anywhere within the plane enclosed by the circular path, but the gaze direction of the novel camera



(a) Sampling. (b) The representation. (c) Rendering.

Figure 2.12: Image cache. In (a), the image cache samples are generated by sampling the scene. In (b), the image cache representation is created by projecting each of the samples onto a plane. In (c), we illustrate the error incurred using this method. A point sample and its representation in the image cache project to different pixels in the novel view.

is restricted to lie in the plane. Novel views are reconstructed by rescaling selected vertical scanlines from the set of captured images. The reconstructed images exhibit significant distortions since the views reconstructed are typically far away from the path along which the source images are captured. The authors discuss an alternate representation that stores the captured images as a set of multi-perspective images in which the center of projection varies across vertical scanlines.

The image caching systems built by Shade et al. [81], Schaufler and Sturzlinger [76], and Aliaga [4], used dynamically created images, image caches, to replace the geometry of a synthetic scene. The image cache can be used to render an *approximate* view of the same scene by projecting it onto a plane in 3D, as illustrated in Figure 2.12. When the plane is seen from a nearby viewpoint the scene is convincingly reproduced. As the novel view gets further from the view that recorded the image, the reproduction becomes warped and unconvincing. These systems trade the lack of parallax in 2D images for speed of rendering.

Presented in Chapter 3, the first novel system in this dissertation, Hierarchical Image

Caching [81], renders complex scenes by automatically creating and rendering a set of cached images that are updated as a viewer moves through a scene. A scene is partitioned hierarchically and the parts of the scene far from the camera are rendered as image caches while the parts close to the camera are render as geometry. A geometric error metric is used to determine when an image cache needs to be updated. A similar system was built contemporaneously by Schaufler and Sturzlinger [76]. These systems cache a sparse set of images covering the entire scene, so they are 2.5D representations.

Aliaga's system [4] took a slightly different approach. When the cached image is warped, the portions of the scene adjacent to the image that are still being rendered as geometry are warped slightly to match the distortion in the image. This minimizes the abrupt boundary that would be noticeable between the warped image cache and the geometry. Since this system creates a sparse set of images it is a 2.5D representation.

Talisman [85] is a hardware graphics system designed to use image caching. Talisman differs from the previous image caching systems in that it uses an affine warp instead of a planar perspective warp. Affine warps are less expensive to compute, but are more limited in scope. In a related paper, Lengyel and Snyder [47] present a quantitative analysis of various warping methods and show convincing results of affine warps used in place of planar perspective warps.

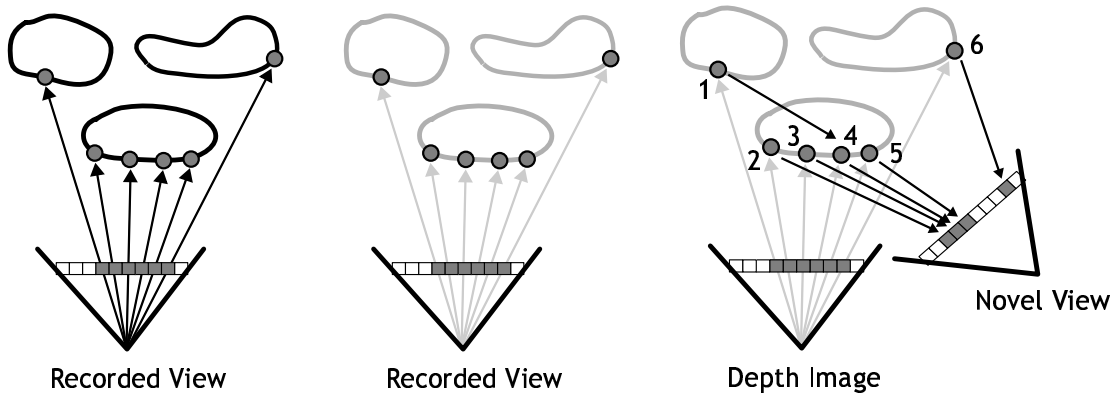
Image caching systems [81, 76, 4, 85, 47] exhibit two primary artifacts: discontinuities in shading and in occlusion relationships. Since these systems dynamically create images, when a new image cache is created there will often be large changes in the shading and occlusion relationships of the objects within the image. This sudden change is referred to as "popping". The error metrics presented by Lengyel and Snyder [47] can be used to control the severity of the artifacts, but there is a fundamental tension between the amount of error that is allowed and the frequency with which new image caches are being created. Popping due to shading discontinuities can be eliminated if the scenes being sampled consist entirely of ideal diffuse surfaces. A diffuse surface has no specular highlights or glossy reflections and looks the same from every direction. This assumption is an example of reducing the

dimension of the plenoptic function by assuming there is constancy in the function, effectively removing two dimensions of the plenoptic function. However, in practice, diffuse surfaces are rarely encountered. When the assumption of constancy of shading over viewing directions is imposed on a non-diffuse scene, information is being thrown away. As will be shown later, the light field and lumigraph systems use constancy in a way that does not throw away any information.

2.1.3 Flow-based Representations

Knowing the depth of each pixel in an image permits more accurate rendering through the use of 3D image warping. A *flow field* can be computed for a pair of images (call them the source and destination images) by finding a corresponding point in the destination image for every pixel in the source image. The difference in image coordinates of the two points defines a vector in the flow field. Points that are far away project to nearly the same pixel in both images and have vectors of small magnitude in the flow field. Points that are close to the cameras project to different points on the destination image and have vectors of large magnitude in the flow field. Thus, there is a direct correspondance between the magnitude of the flow field and the depth of a point in the image. Knowing the flow field and the parameters of the cameras is equivalent to knowing the depth of every point in the scene. This observation has been used in computer vision to compute per-pixel depth values for acquired images using point correspondences between images [26].

Flow-based representations use *depth images*, also known as *range images*, as the fundamental modeling primitive. Up to this point we have assumed no knowledge about how the light traveling along a ray passing through a pixel is created. If we associate a depth value with every pixel, the pixel is more than just a sample of light, it is the light that has been reflected or emitted from a known piece of geometry. Knowing the depth of every pixel in an image is equivalent to knowing the world space coordinates of the point of geometry represented by the pixel. Adding per-pixel depth transforms an image from a 2D representation into a 2.5D representation. The reconstruction step in the image-based ren-



(a) Sampling. (b) The representation. (c) Rendering.

Figure 2.13: Depth image. In (a), the depth image samples are generated by sampling the scene. In (b) the depth image representation is exactly those points captured during sampling. In (c) we render a depth image by projecting each of the point samples onto the image plane of the novel view. Notice that sample 4 overwrites sample 3 and sample 5 overwrites sample 1, correctly reproducing the parallax in the scene. However, there is a gap in the novel view because surfaces not seen by the recording camera have become visible.

dering pipeline (see Figure 2.1) is very simple for 2D representations: either display the entire image, or display a regular subset of the image. Reconstruction for the 3D Movie Maps system is also simple: the 3D representation is projected onto a 2D view by taking an appropriate 2D slice of the volume (a frame of the movie). Reconstruction for a 2.5D representation is a process of reprojection: 2D data is "unprojected" into 3D by adding depth, and then projected back to the 2D view of an observer.

Representations based on depth images typically sample one dimension of space sparsely. The two dimensions covered by the width and height of the view volume of the image are sampled densely, but the dimension corresponding to depth is sampled sparsely. When these samples (now 3D points) are viewed from an oblique angle, holes will appear where data is missing. Figure 2.13 shows the process of sampling and rendering a depth image. In Figure 2.12(c) the inaccuracy of image caches causes points in the scene to project to the wrong place in the novel view. As Figure 2.13(c) shows, depth images allow

correct projection of points, but holes in the reconstructed view will appear when parts of the scene not visible to the recording camera are visible to the novel camera. Finding ways to fill these gaps in reconstruction is often referred to as the "hole-filling problem".

There are three basic methods for rendering depth images: polygon rendering, forward 3D image warping, and backward 3D image warping. In polygon rendering a depth image is converted into a 3D mesh. Each pixel is treated as a vertex in a regular triangulation of the image. Drawing triangles between pixels in this manner helps fill holes in the reconstruction process. However, the interstitial triangles cause the image of the scene to appear stretched or smeared.

An alternative is to use 3D image warping, which can be implemented efficiently in software. A correspondence between pixels in the two images can be constructed by concatenating the inverse of one camera's projection matrix with the projection matrix of the other camera into a 4*x*4 matrix. A pixel in the first image can be warped to the other image by mapping the homogeneous vector, $< x, y, z, 1 >$, composed of the pixel's screen coordinates and its depth through this matrix and then renormalizing the vector by the homogeneous coordinate. Since this operation is symmetric, there are two choices: given a source and a destination image we can either *forward warp* all of the pixels in the source image to the destination image, or we can *backward warp* every pixel in the destination image to find out which source image pixel to paint at that location.

In forward warping, every pixel in the source image is mapped to the output image. In order to fill holes, source pixels are typically *splatted* [89] into the output image. A splat is an idealized point sample rendered as a semi-transparent disc. The size of the disc is adjusted based on the position and resolution of the source and destination views.

In backward warping, every pixel in the destination image is mapped to a location in the source image. The color for the pixel in the destination image is determined by interpolating between the pixels in the neighborhood around the location found in the source image. When all of the source image pixels are assumed to lie on a plane, this interpolation is straightforward. However, when each pixel can have an arbitrary depth, this interpolation

becomes a search. Like triangulation, interpolation can lead to smearing artifacts when there are large gaps between pixels. Laveau and Faugeras [46] used epipolar geometry to show that this search can be restricted to a line. However, compared to splatting, even a linear search is time-consuming. As a result, forward warping with splatting is the most common approach to 3D image warping. The three systems we review in this section use polygon rendering, a 2D approximation to forward warping, and full 3D forward warping, respectively.

Greene and Kass [34] extended the work of Movie Maps by relaxing the restriction of the viewpoint to a predetermined path. Greene and Kass use a regular grid of cubical panoramic images with per-pixel depth. Using the per-pixel depth, the panoramic image closest to the viewpoint was reprojected by rendering each pixel as a 3D quadrilateral. A z-buffer was used to resolve visibility. They solved the occlusion problem by only using the panoramic depth image to render objects that were far away. Nearby objects were rendered using their actual geometry.

Chen and Williams' view interpolation [13] also uses a set of images with per-pixel depth, but employs image warping instead of polygon rendering to reproject the depth images. Using per-pixel depth, flow fields are computed for every pair of adjacent input images. Because flow fields are many-to-one mappings, two fields must be computed for every pair of images, one for each direction. An intermediate view between two of the input images is created by linearly interpolating between the two flow fields. An interpolation parameter controlls how far along each flow field the images are warped. For instance, if the new view is exactly halfway between each of two input images, the warping algorithm follows halfway along the vectors of each of the two flow fields. The warping is computed using an efficient 2D approximation to full 3D image warping [7]. Hole filling is done as a post-process by interpolating across unfilled regions of the image using an algorithm akin to flood filling. Visibility is determined by pre-sorting the pixels in pairs of images by depth. When rendering an intermediate view the pixels from the two input images are warped simultaneously in a back-to-front order.

The Plenoptic Modeling work of McMillan and Bishop [58] extends the work of view interpolation to use cylindrical images of acquired data and 3D image warping. McMillan and Bishop use a novel analysis of epipolar geometry to augment standard computer vision techniques to compute flow fields between pairs of cylindrical images of acquired data. They also present an algorithm that reprojects the pixels of a cylidrical image in the order needed such that they are painted in a back-to-front order on the image plane of the novel view.

Like the image caching systems, these flow-based systems also use 2.5D representations. However, using per-pixel depth facilitates the use of more accurate reprojection algorithms. The quality of reconstruction provided by these systems depends on the number and resolution of depth images created. In the limit, if a panorama is created at every possible position, the 5D plenoptic function is sampled comprehensively.

2.1.4 Hybrid Representations

The next group of papers use representations that extend depth images to incorporate more geometric information. These representations fall into two broad categories: mesh-based representations and volumetric representations.

Mesh-based Representations. The first class of hybrid creates 3D geometric meshes from depth images. Darsa et al. [20] present a system for interactively rendering complex static scenes by creating meshes from depth images. Given a cubical depth image of a scene, a mesh is created by triangulating the depth information. The mesh is then texture-mapped with the color image. For viewpoints close to the original one, the renderings made using this technique will exhibit the proper parallax. However, the shading captured is static, so this will only faithfully render scenes that are completely diffuse. In addition, renderings exhibit smearing artifacts common to mesh-based representations of depth images. This is a 2.5D representation that solves the hole-filling problem by rendering a mesh.

Mark et al. [55], construct a similar system, post-rendering 3D warping, that creates

meshes from depth images dynamically. This work is targeted towards the real-time display of a complex environment in a walk-through style application. As a viewer moves through a scene, depth images are created on a server at 5 frames per second and sent to a local workstation where they are rendered as meshes at 60 frames per second. Since the depth images are created dynamically, there is no restriction on the movement of the viewer. Like the image caching schemes, the transitions between an out-of-date depth mesh and the newly made one can produce discontinuities in shading. Like that of Darsa et al., this is a 2.5D representation. However, since depth meshes are created dynamically, this system is less susceptible to smearing artifacts.

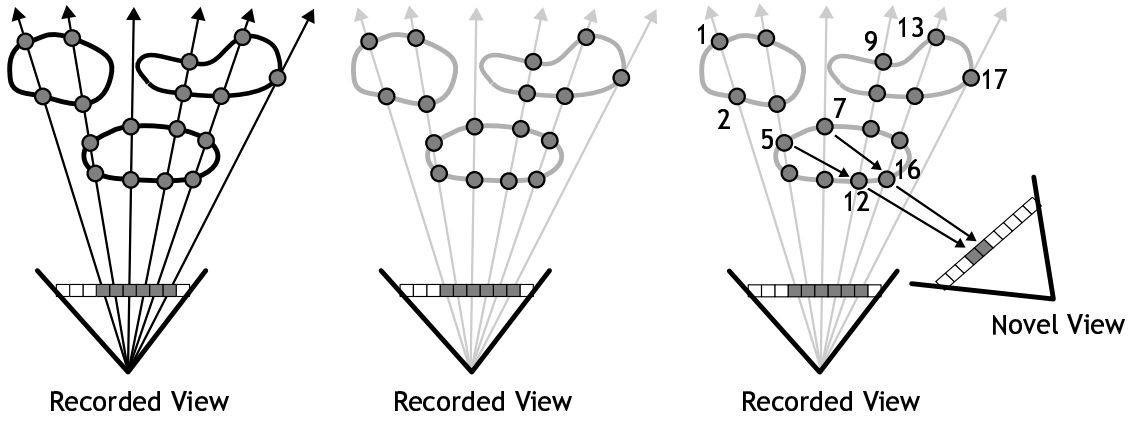
The view-based rendering system of Pulli et al. [69] is geared towards displaying scanned real objects. A collection of colored range images are acquired from viewpoints surrounding an object. The object is reconstructed by rendering meshes captured from the three recorded viewpoints closest to the virtual viewpoint. This technique captures both the shape and the shading of an object using a sparse set of acquired images. While the shape of an object can be faithfully reproduced using a sparse set of acquired images, shading can require a dense set of images. So, this technique exhibits popping artifacts in the shading as the viewpoint orbits about the object. Acquiring multiple images per range image would alleviate this problem at the expense of higher storage cost. Since this technique uses a small set of depth images, we consider it to be a 2.5D representation.

Debevec et al. [22] present a view-dependent rendering algorithm, view-dependent texture mapping (VDTM), that uses a single geometric mesh along with a collection of images. Given a geometric model and a set of registered images of that model, novel views of the scene are created by texture mapping the polygons of the model with images that are closest to the viewpoint. Unlike Pulli et al., it is assumed that every polygon in the scene is visible in more than one photograph. Blending together the image information from the three closest images produces smooth transitions in shading of the model as the viewpoint orbits the model. The degree to which this technique can faithfully reproduce specularity is determined by the number of images used. We consider this to be a 2.5D representation.

Figure 2.14: Layered depth images.(a) shows an LDI from its center of projection. (b) shows the LDI from an oblique angle, but only one layer is rendered (mimicking the behavior of a depth image). (c) shows the LDI from the same oblique angle, but using all of the layers.

Building a single 3D mesh for an object fixes one of the degrees of freedom in position to lie on a 2D surface, creating a 2D reduction of the plenoptic function. Sparsely sampling the space of directions about each point of the surfaces adds another half dimension to the representation. A dense sampling of the direction space would make this a 4D representation. As we'll see in the next section, the surface light field [91, 61] does just this.

Rademacher and Bishop [70] employ a style of MPI similar to the multiperspective panoramas of Wood et al. [92] to create what they call multiple-center-of-projection (MCOP) images. An MCOP image is an MPI with per-pixel depth in which each vertical strip of pixels is taken from the center vertical strip of a perspective range camera that is allowed to move freely along a space curve. It is assumed that the camera positions and viewing directions vary smoothly, ensuring that the epipolar geometry varies smoothly across the columns of an MCOP image. As a consequence, incremental warping can be used to efficiently reproject MCOP images. Unfortunately, McMillan's occulsion-compatible warp ordering can't be used with multiple cameras simultaneously, so a z-buffer is used to resolve visibility. Alternatively, a 3D mesh can be constructed using the per-pixel range image. The 3D mesh can then be rendered by texture-mapping it with the color information



(a) Sampling. (b) The representation. (c) Rendering.

Figure 2.15: Layered depth image. In (a), the LDI samples are generated by sampling the scene at multiple points along each ray extending from the recorded view. In (b) the LDI representation is exactly those points captured during sampling. In (c) we render an LDI by projecting each of the point samples onto the image plane of the novel view using McMillan's occlusion-compatible warp ordering. Since the novel view is to the right of the recorded view, the samples in the recorded view are warped in a left-to-right order. In addition, along each ray we warp samples in a back-to-front order. A selection of the samples are labeled according to their position in the warp ordering. Notice that sample 5, a second layer sample is overwritten by sample 12, a first layer sample. The same situation exists for samples 7 and 16. After projection, splatting is used to create a continuous reconstruction.

from the MCOP. Like Pulli et al., each triangle in the mesh is seen from only one view, so this is a 2.5D representation.

Volumetric Representations. The second class of hybrid representations we consider build sparse volumes of point-sampled geometry. Layered Depth Images (LDIs) [80] extend depth images by adding multiple samples of the plenoptic function along each ray of an image. LDIs, the second novel image-based representation presented in this dissertation, are discussed in detail in Chapter 4. The work is summarized here to complete the range of image-based representations.

The major shortcoming of depth images is the holes produced in reconstructions when surfaces hidden to the recording camera become visible to the novel view. LDIs solve this

problem by storing multiple samples along each ray of an image. Figure 2.14(a) shows an LDI from its center of projection. Figure 2.14(b) shows the LDI from an oblique angle, but only one layer is rendered (mimicking the behavior of a depth image). Figure 2.14(c) shows the LDI from the same oblique angle, but using all of the layers. Using McMillan's occlusion compatible warp ordering for planar images [57], LDIs can be rendered efficiently in software without the use of a z-buffer. Figure 2.15 shows the process of sampling and rendering an LDI. In practice, LDIs sparsely sample the dimension of a scene corresponding to depth in the image. A naive way of creating an LDI would be to treat each ray of an image as a skewer, and to record every intersection of the ray with the geometry of the scene. This produces a dense sampling of the scene and thus a 3D representation. This method isn't used for several reasons, one of which is run-time performance; the data set would be too large to render in real-time. To combat this problem, Shade et al. [80] only store intersections that are visible from a view nearby the center of projection of the LDI. This visibility-driven sampling results in a sparse data set that can be rendered in real-time. In addition, if an intersection is visible from more than one nearby view, the radiance values from all views are averaged to create a single view-independent sample. However, this averaging produces the correct radiance value only when the scene consists solely of purely diffuse surfaces. A similar data structure was used by Max [56] to create anti-aliased renderings of trees.

Similar in nature to Lippman's Movie Map [50], a Layered Depth Movie can be created from a series of LDIs placed along a predetermined path. The viewer is constrained to move along the path, but since an LDI is being displayed instead of an image, the viewer can move freely in a volume of space surrounding the path. Creating a dense set of LDIs along a path results in a 3.5D representation.

Lischinski and Rappoport [51] extend LDIs to handle non-diffuse scenes. Two sets of LDIs are store for a scene: a Layered Depth Cube (LDC) consisting of three high resolution LDIs (corresponding to three orthogonal directions) that stores the view-independent component of the sampled geometry (diffuse color, depth, normal, and an index into a table

of bidirectional reflection distribution functions, BRDFs); and a Layered Light Field (LLF) consisting of a dense set, 66 to 1026, of low resolution LDIs that store the view-dependent component of the sampled geometry (the total radiance leaving the scene in the direction of projection). The scene is rendered by first projecting the LDC to an intermediate image. After hole filling, each point in the image is shaded by integrating the incoming radiance from the LLF and applying this to the BRDF. This representation adds a dense sampling of the space of directions to an LDI, resulting in a 4.5D representation.

2.1.5 Holographic Representations

The roots of image-based rendering can be found in the field of holography. A hologram is a photographic record of the interaction of light with an object which has the property that when it is illuminated properly, it displays the object of record with full parallax. Holograms have become commonplace objects. Most credit or debit cards have a rainbow hologram (also known as a Benton hologram) in the corner. The rainbow hologram is a horizontal parallax only (HPO) hologram: a change in viewpoint in the vertical direction changes the hue of the object, while a change in the horizontal direction changes the perspective view of the object. More precisely, a hologram is the record of the interference pattern created when two beams of correlated light interact. One is a reference light and the other is perturbed by placing the object to be recorded in the path of the beam of light. When this interference pattern is illuminated by the same reference light source, the pattern of light that hits the photographic plate from the perturbed source is reconstructed, resulting in an image of the object. Holograms are very effective, but they have a number of limitations: The reconstructed image is necessarily the same size as the subject that was recorded. The recording apparatus must be stable to within one tenth of a wavelength of light during exposure (meaning that holograms are more or less confined to a table top setting). Natural color is very hard to reproduce. Finally, the images created are snapshots in time: the scene being imaged must be static.

Holographic stereograms are a hybrid of holography and photography. The primary

advantage holographic sterograms have over holograms is that they can be used to make images of anything that can be photographed. A holographic stereogram is created by exposing a piece of holographic film to a series of perspective views. When the hologram is illuminated with the reference light, a different perspective image will be view by each eye of the observer resulting in a stereo image of the scene. Holograms and holographic stereograms effectively store a 2D array of 2D images and are a 4D reduction of the plenoptic function. Like the lightfield and Lumigraph representations in the next section, these two representations apply the ray law to reduce by one the number of dimensions of the plenoptic function.

2.1.6 Digital Analogs to Holographic Representations

The light field invented by Levoy and Hanrahan [48] and the lumigraph invented by Gortler et al. [32] are 4D representations that are essentially digital analogs to holographic stereograms. Assuming that a viewer is going to stay within a volume of free space, every ray of light that enters this space remains unchanged until it exits. A simple example of such a volume is a sphere. All of the rays intersecting a sphere can be parameterized using 4 coordinates: the lattitude and longitude of the ray as it enters and exits the sphere. This reduces the number of dimensions of the plenoptic function from five to four. Normally, it takes five numbers to describe a ray: three for position and two for direction. Consider a line intersecting a sphere. The free space assumption implies that every 5D ray originating on that line, looking along the line is equivalent: they all see the same value of radiance. Because the behavior of light is constant in the region of free space, this collection of 5D rays can be named by one 4D ray.

In practice, these representations use two planes to parameterize the space of rays. To represent the entire environment surrounding a viewer, six sets of planes can be used to describe all of the rays entering a box. The rays for a set of planes are typically stored as a 2D array of 2D images. The near plane acts as a locus of camera centers, while the far plane acts as the focal plane for the cameras. Each image in the representation is a skewed

perspective pinhole projection. To avoid aliasing, camera centers on the near plane need to be spaced no further apart than the aperture of the reconstruction camera [37]. This is a very dense representation that can produce very high quality images, but requires an enormous amount of storage.

A surface light field [61, 91] is an alternative parameterization of a light field. Like view-dependent texture mapping (VDTM), the surface light field representation assumes that a 3D mesh of an object exists. In contrast to VDTM, however, surface light fields record the radiance leaving the surface using a dense set of cameras. This results in a 4D representation: for every point on a surface there is an associated lumisphere, a 2D image of the radiance leaving the surface at that point. By taking advantage of the fact that lumispheres vary smoothly over the surface of an object, surface light fields can be compressed with higher fidelity than the standard two-plane parameterization.

Light field representations can reproduce the plenoptic function exactly in regions of free space. However, since light fields are a 4D reduction of the plenoptic function, they typically require a great deal of storage space. If the object being represented can be represented by a 3D mesh, surface light fields offer the advantages of a 4D representation in concert with a compressed representation.

2.1.7 Summary

Table 2.1 shows a summary of the papers we have reviewed. The representations are listed in the order they have been presented. Each line of the table lists the dimension of the representation and which of the four tools for paring-down the plenoptic data reduction techniques are employed.

2.2 Taxonomy of Representations

This section presents a walk through the representations made possible by applying the four tools for paring-down the plenoptic function: taking a subset of the plenoptic func-

Table 2.1: A summary of IBMR papers: the dimension of the representation and the techniques used.

Representation	# Dimensions	Technique		
		Reducing dimension	**Sparsely sampling**	**Eliminating redundancy**
Image	2	√		
Katayama et al. [42]	3	√		
MPPCA [92]	2	√		
Photo Finish	2	√		
Concentric Mosaics [82]	3	√		√
Movie Maps [50]	3.5	√	√	
Image Caching[81]	2.5	√	√	
Greene and Kass [34]	2.5	√	√	
View Interpolation [13]	2.5	√	√	
Plenoptic Modeling [58]	2.5	√	√	
Darsa et al. [20]	2.5	√	√	√
Post-rendering 3D warping [55]	2.5	√	√	
MCOP [70]	2.5	√	√	
View-based rendering [69]	2.5	√	√	
VDTM [22]	2.5	√	√	
Layered Depth Image [80]	2.5	√	√	√
Layered Depth Movie	3.5	√	√	√
Lischinski and Rappoport [51]	4.5		√	√
Hologram	4			√
Holographic stereogram	4			√
Light field [48]	4			√
Lumigraph [32]	4			√
Surface light field [91]	4			√

tion, reducing the dimension of the plenoptic function, sparsely sampling a dimension, and eliminating redundancy. Tables 2.2 and 2.3 give a concise description of each of the representations. There are six distinctions to each representation: how many of the dimensions are fixed, how many dimensions are free, which parameters are fixed, which parameters are free, whether radiance is considered entering or leaving, and what the representation is called. The representations are grouped into categories based on the number of dimensions that are fixed, starting with the singular 0 dimensional object and working up to four dimen-

sional objects. Within each category are a number of variations of which parameters are fixed and which are left free. It is assumed the 5D plenoptic function is being approximated, but it will sometimes be interesting to reincorporate time and consider approximations to a 6D plenoptic function. The three degrees of freedom in position are *x*, *y*, and *z*, and the two degrees of freedom in direction are θ and ϕ. In each entry, we'll fix some of the degrees of freedom and let other vary. For instance, an image is described by fixing *x*, *y*, and *z* and leaving θ and ϕ free. Fixing θ, ϕ, and *z*, and leaving *x* and *y* free describes an orthographic image. Fixing *z* chooses some plane in space. The fact that *z* was the position variable chosen to be fixed is unimportant: any of the three could have been chosen. To describe basic geometric constructs like points, lines, and planes, one or more of *x*, *y*, and *z* will be fixed. To describe more general geometry, free variables will vary over the geometric object. For example, a surface can be represented by fixing *z* while *x* and *y* are left to vary over the surface.

2.2.1 0D Representations

Fix:	*x, y, z,* θ, ϕ	**Vary:**	*None*	**Radiance:**	*both*

Point: A point sample of the plenoptic function is defined by a point in space and a direction. This is specified using 5 parameters: 3 for a point $< x, y, z >$, and 2 for a direction $< \theta, \phi >$. A point in this function describes a single radiance value. If the radiance is arriving, this is a pixel in an image. If the radiance is leaving, this is a pixel in a radiance map.

2.2.2 1D Representations

Fix:	*x, y, z,* θ	**Vary:**	ϕ	**Radiance:**	*arriving*

1D slice: This representation is simply a 1D slice of an image. Picking a point and rendering an image of the scene (or taking a photograph of the world), and then sampling along a line or curve through the image is a 1D reduction. The number of free parameters de-

fines the dimension of a reduction. Extracting a single line of latitude or longitude from a spherical image is a 1D slice of the image. Likewise, taking a vertical line from a planar image is also a 1D slice. This representation forms the basis of work done using vertical slit cameras in holography, as illustrated in Figure 2.6.

Fix:	*x, y, z, θ*	**Vary:**	*φ*	**Radiance:**	*leaving*

Consider the radiance as leaving a point instead of as arriving, results in a 1D slice of a lumisphere.

Fix:	*x, y, θ, φ*	**Vary:**	*z*	**Radiance:**	*arriving*

This representation is equivalent to choosing a single pixel in a set of images that are taken along a line. If planar images are used and θ and ϕ are chosen to look in a direction perpendicular to the line of motion, this amounts to a vertical slice of an Epipolar Plane Image. Choosing θ and ϕ to point along the direction of the line of motion builds a table of all the radiance values along a single ray as it travels through a scene.

Fix:	*x, y, θ, φ*	**Vary:**	*z*	**Radiance:**	*leaving*

The same representation, but with radiance leaving. One interpretation of this structure is that it is a slice of an area light source.

2.2.3 2D Representations

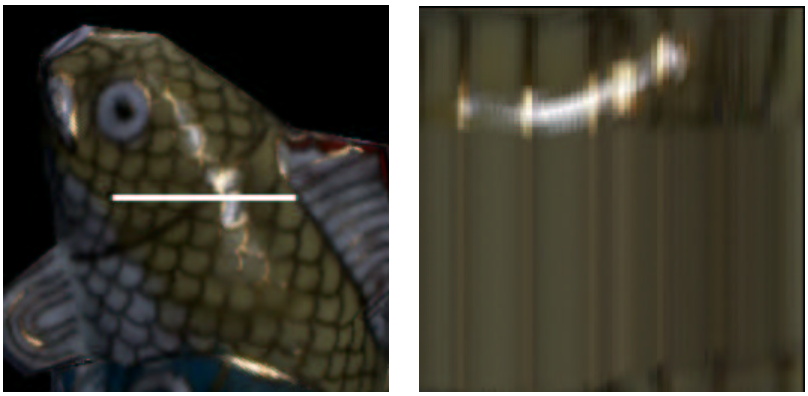
Fix:	*x, y, z*	**Vary:**	*θ, φ*	**Radiance:**	*arriving*

Image: This representation is the standard 2D image. Depending on how θ and ϕ are represented, one of the 2D image types is created: planar, cubical, spherical, or cylindrical.

Fix:	*x, y, z*	**Vary:**	*θ, φ*	**Radiance:**	*leaving*

Lumisphere: An image that records outgoing radiance. This is also a point light source. A similar structure is used in projective texture mapping and shadow mapping.

Fix:	*x, y, θ*	**Vary:**	*z, φ*	**Radiance:**	*arriving*



(a) (b)

Figure 2.16: (a) Surface light field. (b) Transect of surface light field. Horizontal axis shows position along white line across fish. Vertical axis shows a slice of a lumisphere on a user-selected great circle.

Multiperspective image: Varying one parameter of position (a line or curve) and one parameter of direction (a slice through an image) creates a class of multiperspective images. Choosing a horizontal slice through each image results the epipolar plane images used by Katayama et al. [42]. Choosing a vertical slice through each image results in the multiperspective panoramas of Wood et al. [92] and Glassner [30], and the MCOP images of Rademacher and Bishop [70].

Fix:	x, y, θ	**Vary:**	z, ϕ	**Radiance:**	*leaving*

Lumisphere slices: As mentioned above, the position parameter can be varied over an arbitrary line or curve. A series of slices through the lumispheres of a surface light field can be represented by fixing position to a curve that lies on the surface of an object and direction to a great circle on the sphere of directions. Wood et al. show this type of image in Figure 5 of [91], reproduced here in Figure 2.16.

Fix:	x, y, z, θ	**Vary:**	ϕ, *time*	**Radiance:**	*arriving*

Single-perspective, multi-time image: This type of image uses the same center of projection for each vertical slit, but a different point in time. Figure 2.11 shows an example of

this type of image taken with a “photo finish” camera.

Fix:	*z*, θ, ϕ	**Vary:**	*x, y*	**Radiance:**	*arriving*

Orthographic planar image: A planar orthographic image can be created by fixing a camera to lie in a plane with a gaze perpendicular to the plane. An oblique parallel projection can be created by choosing a different fixed gaze direction.

Fix:	*z*, θ, ϕ	**Vary:**	*x, y*	**Radiance:**	*leaving*

Area directional light source: The inverse of a planar image can be thought of as a directional area light source: an area light source that emits light in only one direction.

2.2.4 2.5D Representations

Fix:	*x, y, z*	**Vary:**	θ, ϕ, *sparse z*	**Radiance:**	*arriving*

Layered depth image: Layered depth images [80] record several samples along each ray of the image. Like a 2D image, position is fixed and direction varies. However, taking multiple samples along each ray increase the dimension of the data set. This is a case of sparsely sampling the plenoptic function. Densely sampling along the rays would effectively unfix one of the position dimensions, creating a 3D representation. Sparsely sampling along the rays puts layered depth images between 2D and 3D representations.

Fix:	*x,y,z*	**Vary:**	θ, ϕ, *sparse z*	**Radiance:**	*arriving*

Multi-layered image: Forcing all of the rays of a layered depth image to be sampled at the same set of locations results in a set of concentric images (or if using a planar image, a set of planes). This type of representation has been used in cel animation, side-scrolling video games, and in the immersive virtual reality system built by Regan and Pose [71].

Fix:	*set of x,y,z*	**Vary:**	θ, ϕ	**Radiance:**	*arriving*

Set of 2D images: This representation encompasses all systems that use a collection of 2D images, including: image caching [81], view interpolation [13], and QuickTime VR [12].

Fix:	*set of* θ, ϕ	**Vary:**	*x,y,z over a surface*	**Radiance:**	*arriving*

View-dependent texture mapping: This is the representation used in view-based rendering [69] and view-dependent texture mapping [22]: a sparse set of photographs of a surface.

2.2.5 3D Representations

Fix:	*x, y*	**Vary:**	*z, θ, φ*	**Radiance:**	*arriving*

Space movie: In this representation, images are taken as a camera moves along a line. Since time is frozen, the environment is static. Freezing camera position and letting time move forward still results in a 3D representation, but now it is a time movie. Letting both time and camera position vary results in a 4D representation, a space-time movie. Space movies serve as input data to several representations reviewed in the previous section [9] [42] [32] [48] [82].

Fix:	*x, y*	**Vary:**	*z, θ, φ*	**Radiance:**	*leaving*

Set of lumispheres: Constraining the position of a line to lie along the surface and recording the outgoing radiance captures a set of lumispheres. When viewed as a movie, this sequence shows the change in reflectance of an object as a point moves along its surface.

Fix:	*x, y*	**Vary:**	*z, θ, φ*	**Radiance:**	*arriving*

Dense Layered Depth Image: The standard LDI stores only some of the intersections of a ray with the geometry in a scene. Storing every intersection builds a complete 3D volumetric representation of the geometry of a scene.

Fix:	*x, y, z*	**Vary:**	*θ, φ, time*	**Radiance:**	*arriving*

Time movie from a fixed point: Adding time to an image results in a movie in which the camera is stationary, but the world is moving: a time movie. This is a 3D reduction of a 6D function.

Fix:	*x, y*	**Vary:**	*z, θ, φ, time*	**Radiance:**	*arriving*

Space-time movie: Taking this notion a step further and adding movement to the camera results in a space-time movie: a 4D reduction of a 6D function. The familiar motion picture

(or video) is a 3D reduction of this 4D space because the *x* and *y* position parameters are a function of time.

2.2.6 3.5D Representations

Fix:	*x, y*	**Vary:**	*z, θ, φ*	**Radiance:**	*arriving*

Movie Map: In this representation, the free position parameter is restricted to lie on one of a set of predetermined paths. Connecting the paths together in a graph builds a movie map.

Fix:	*x, y*	**Vary:**	*z, θ, φ, sparse y*	**Radiance:**	*arriving*

Layered Depth Movie: Just as taking a series of images along a line results in a movie, taking a series of layered depth images along a line results in a layered depth movie.

2.2.7 4D Representations

Fix:	*z*	**Vary:**	*x, y, θ, φ*	**Radiance:**	*arriving*

Light field: The lightfield and lumigraph representations use a two-plane parameterization to describe all of the rays passing through a region of space: 2 parameters for position on the near plane (locus of camera centers), and 2 for position on the far plane (direction). Following the terminology of Levoy and Hanrahan [48], a pair of planes is a slab, and the light entering (or leaving) the box can be described by a set of six slabs.

Fix:	*z*	**Vary:**	*x, y (over surface), θ, φ*	**Radiance:**	*leaving*

Surface light field: A lightfield need not be restricted to a plane parameterization. Any two dimensional set of positions combined with the sphere of directions defines a light-field. Restricting *x* and *y* to a surface results in the surface light fields of Miller et al.and Wood et al..

Fix:	*z*	**Vary:**	*x, y, θ, φ, sparse z*	**Radiance:**	*arriving*

Planar set of panoramas: This representation is a set of sets of panoramic images restricted to a plane. Each element of the representation is a set of panoramic images with the image plane positioned at several predetermined distances from the center of projection. Conceptually, this is a 3D version of cel animation or an extension of the work of Regan and Pose [71] to represent viewpoints on a plane.

2.2.8 4.5D Representations

Fix:	*none*	**Vary:**	x, y, z, θ, ϕ	**Radiance:**	*arriving*

Non-diffuse layered depth image: This is a layered depth image with view-independent shading information. Since an LDI sparsely samples one direction of space and assumes diffuse shading, it is a 2.5D representation. The work of Lischinski and Rappoport [51] adds view-independent shading to LDIs and thus increases the dimension of the representation by two dimensions. This is very close to being a 5D representation of the plenoptic function.

2.3 Summary

This chapter has presented a novel classification of image-based representations, summarized in tables 2.2 and 2.3. All of these representations can be classified in terms of four common techniques used to simplify the plenoptic function: taking a subset of the function, reducing the dimension of the function, sparsely sampling a dimension of the function, and eliminating redundancy within the function.

Table 2.2: Plenoptic representations

# Dims fixed	# Dims free	Fixed/Free Parameters	Radiance	Object
5	O (Point)	fix: x, y, z, θ, ϕ vary: none	N/A	single radiance value
4	1 (Line)	fix: x, y, z, θ vary: ϕ	arriving	1D slice of an image
			leaving	1D slice of a lumispere
		fix: x, y, θ, ϕ vary: z	arriving	line in an EPI
		fix: x, y, θ, ϕ vary: z	leaving	line in an area light source.
3	2 (Image)	fix: x, y, z vary: θ, ϕ	arriving	image
		fix: x, y, z vary: θ, ϕ	leaving	lumisphere
		fix: x, y, θ vary: z, ϕ	arriving	multiperspective image
		fix: x, y, θ vary: z, ϕ	leaving	lumisphere slices
		[2D slice of 6D] fix: x, y, z, θ vary: ϕ, time	arriving	single-perspective, multi-time image
		fix: z, θ, ϕ vary: x, y	arriving	orthographic planar image
		fix: z, θ, ϕ vary: x, y	leaving	area directional light source
3.5	2.5	fix: x, y, z vary: θ, ϕ	arriving	layered depth image
		fix: x, y, z vary: θ, ϕ	arriving	multi-layered image
		fix: set of x, y, z vary: θ, ϕ	arriving	Quicktime VR
		fix: set of θ, ϕ vary: x, y, z over a surface	arriving	view-dependent texture mapping

Table 2.3: Plenoptic representations continued

# Dims fixed	# Dims free	Fixed/Free Parameters	Radiance	Object
2	3 (Movie)	fix: *x*, *y* vary: *z*, θ, ϕ	arriving	space movie, epipolar plane images, HPO holographic stereogram, HPO light field
		fix: *x*, *y* vary: *z*, θ, ϕ	leaving	set of lumispheres
		fix: *x*, *y* vary: *z*, θ, ϕ	arriving	dense LDI
		[3D slice of 6D] fix: *x*, *y*, *z* vary: θ, ϕ, time	arriving	time movie
		[4D slice of 6D] fix: *x*, *y* vary: *z*, θ, ϕ, time	arriving	space-time movie
2.5	3.5	fix: *x*, *y* vary: *z*, θ, ϕ	arriving	movie map
		fix: *x*, *y* vary: *z*, θ, ϕ	arriving	layered depth movie
1	4	fix: *z* vary: *x*, *y*, θ, ϕ	arriving	light field
		fix: *z* vary: *x*, *y* (over surface), θ, ϕ	leaving	surface light field
		fix: *z*, ray parameter vary: *x*, *y* (over plane), θ, ϕ	arriving	set of panoramas
0.5	4.5	fix: vary: *x*, *y*, *z*, θ, ϕ	arriving	non-diffuse LDI

Chapter 3

HIERARCHICAL IMAGE CACHING

This chapter describes the first of two novel view-dependent image-based representations presented in this dissertation. The system presented in this chapter uses image caches as a basic building block for creating an image-based representation of a scene [81]. As a viewer navigates through a virtual environment, the appearance of distant parts of the scene changes little from frame to frame. We exploit this *path coherence* by replacing parts of the scene with image caches that are created in one frame for possible reuse in many subsequent frames. In this way, the 3D scene is represented as a collection of 2D image caches, resulting in a 2.5D representation.

The major shortcoming of image caches is that they cannot account for changes of occlusion that should appear within the image cache as a novel view moves away from the recorded view. To mitigate this, the system presented here uses a simple error metric to determine when an image cache no longer adequately represents the portion of the scene it replaces. When this happens, the image cache is discarded, and a new one is created from the point of view of the current novel view. In this manner, the collection of image caches that comprise the scene is continually updated to avoid visual artifacts. The "on demand" nature of this algorithm has two important consequences. First, since the images are cached dynamically, they are continually capturing new values of the light being reflected by the geometry in the scene. So, even though the representation is view-dependent locally, dynamically updating the representation as the viewer moves through the scene gives it a quasi-view-independent characteristic globally. The amount of view-independence is determined by the update rate. While not done in this system, it would be possible to create an error metric that takes into account the changes in shading due to the angle of the novel view

with respect to an image cache. The second consequence of "on demand" caching is that in order to maintain interactive frame rates, caches must be fast to compute (or capture). For this reason, the system presented here focuses solely on scenes that can be rendered using modern graphics hardware.

3.1 Algorithm

The basic idea behind our algorithm is to exploit path coherence by caching images of objects rendered in one frame for possible reuse in many subsequent frames. However, instead of simply redrawing the image, we apply the image as a texture map to a fixed quadrilateral placed in world space at the center of the object. This textured quadrilateral is then rendered instead of the original object during several successive frames, using the current viewing transformation at each frame. In this way, at each frame, the image of the object is slightly warped, approximately correcting for the slight changes in the perspective projection of the original object as the viewer moves through the scene. Compensating for motion parallax in this manner results in fewer "snapping" artifacts when the cached image is updated and increases the number of frames for which the cache yields an acceptable approximation to the object's appearance.

To gain the most from image caching, it is not enough to cache images for individual objects. If too many objects are visible, the sheer number of textured polygons that must be rendered at each frame may overwhelm the hardware. However, distant objects that require infrequent updates can be grouped into clusters, and a single image can be cached and rendered in place of the entire cluster. Thus, our algorithm operates on a hierarchical representation of the entire scene, rather than on a collection of individual objects. An image can be computed and cached for any node in the hierarchy; hence the name "hierarchical image caching".

We construct the hierarchy as a preprocessing step by computing a BSP-tree [27] partitioning of the environment, as described in Section 3.3. We chose to use a BSP-tree since it

allows us to traverse the scene in back-to-front order, which is necessary to ensure that the partially transparent textured quadrilaterals are composited correctly in the frame buffer. In addition, BSP-trees are more flexible than other spatial partitioning data structures, making it is easier to avoid splitting objects.

The leaf nodes of the BSP-tree correspond to convex regions of space and have associated with them a set of geometric primitives. This set consists of all the geometric primitives contained inside the node. In addition, it also contains nearby primitives from the neighboring nodes, as will be explained in Section 3.3. Any node in the tree may also contain a cached image.

At each frame we traverse the BSP-tree twice. The first traversal culls nodes that are outside the view frustum and updates the image caches of the visible nodes:

```
UpdateCaches(node, viewpoint)
if node is outside the view frustum then
    node.status ← CULL
else if node.cache is valid for viewpoint then
    node.status ← DRAWCACHE
else if node is a leaf then
    UpdateNode(node, viewpoint)
else
    UpdateCaches(node.back, viewpoint)
    UpdateCaches(node.front, viewpoint)
    UpdateNode(node, viewpoint)
```

For a leaf node, the routine *UpdateNode* decides whether, for the current viewpoint, it is more cost-effective to draw the geometry stored with the node, or to compute and cache an image:

```
UpdateNode(node, viewpoint)
if viewpoint ∈ node then
    if node is a leaf then
        node.status ← DRAWGEOM
    else
        node.status ← RECURSE
    return
k ← EstimateCacheLifeSpan(node, viewpoint)
amortizedCost ← ⟨cost to create cache⟩/k + ⟨cost to draw cache⟩
if amortizedCost < ⟨cost to draw contents⟩ then
    CreateCache(node, viewpoint)
    node.status ← DRAWCACHE
    node.drawingCost ← ⟨cost to draw cache⟩
else
    if node is a leaf then
        node.status ← DRAWGEOM
        node.drawingCost ← ⟨cost to draw geometry⟩
    else
        node.status ← RECURSE
        node.drawingCost ← node.back.drawingCost + node.front.drawingCost
```

Geometry is always drawn if the viewpoint is inside the node. Otherwise, the routine *EstimateCacheLifeSpan* computes an estimate of the number of frames k for which we expect the cached image to remain valid, as will be described in Section 3.2. This estimate is used to compute an amortized cost-per-frame for this node for each of the next k frames. We compute and cache an image only if the amortized cost is smaller than the cost of simply drawing the node's contents. For a leaf node, this cost is simply the cost of drawing the contained geometry, while for an interior node, this cost is the cost of drawing the node's children. The costs to draw geometric primitives and to create a cached image are established experimentally on each platform and are given as input to our system.

The routine *CreateCache* starts by computing an axis-aligned rectangle that is guaranteed to contain the image of the node's contents on the screen. This rectangle is obtained by transforming the corners of the node's bounding box from world coordinates to screen coordinates and taking the minima and maxima along each axis. If the dimensions of the

rectangle exceed those of the viewport, no image is cached. Otherwise, we redefine the viewing frustum so that it contains the entire node without changing the viewpoint or the view direction, and render the node. For a leaf node we draw all of its geometry, while for an interior node we draw its children. In many cases, the children are drawn using their cached images, if any exist. Thus, caching an image typically does not involve drawing all the geometry contained in the corresponding subtree. After drawing the contents of the node, we copy the corresponding rectangular block of pixels into the node's image cache. As mentioned earlier, we use the cached image as a texture map that is applied to a quadrilateral representing the entire node. In order to define an appropriate quadrilateral in world space, we project the corners of the image rectangle onto a plane of constant depth with respect to the viewpoint that goes through the center of the node's bounding box.

Once the cached images have been updated, we can proceed to render the scene into the frame buffer, during a second traversal of the BSP-tree from back to front:

```
Render(node, viewpoint)
if node.status == CULL then
    return
else if node.status ∈ {DRAWCACHE, DRAWGEOM} then
    Draw(node)
else if viewpoint is in front of node.splittingPlane then
    Render(node.back, viewpoint)
    Render(node.front, viewpoint)
else
    Render(node.front, viewpoint)
    Render(node.back, viewpoint)
```

To complete the description of our algorithm, the next section describes the error metric we use to determine whether a cached image is valid with respect to a given viewpoint and to estimate the lifespan of a cached image. Section 3.3 describes in more detail our BSP-tree construction algorithm.

3.2 Error Metric

The algorithm described in the previous section requires answers to the following two closely related questions:

1. Given a node with a cached image computed for some previous view, is the cached image valid for the current view?

2. Given a node in the hierarchy and the current view, if we were to compute and cache an image of this node, for how many frames is the cached image likely to remain valid?

In order to answer these questions efficiently we need to define an error metric, which, given a node in the hierarchy, its cache, and the current viewpoint, quantifies the difference between the appearance of the cached image and that of the actual geometry. If this difference is smaller than some user-specified threshold ϵ, the approximation is deemed acceptable, and the cache is considered valid. An important requirement for an acceptable error metric is that it must be fast to compute. For example, we cannot afford to analyze the geometric contents of the node, as the number of primitives contained in a node can be very large.

Our algorithm employs an error metric that measures the maximum angular discrepancy between a point inside a node and the point that represents it in the cached image. We shall use the 2D diagram shown in Figure 3.1 to define our error metric more precisely. The rectangle in this diagram represents the bounding box of a node in the hierarchy. The thick line segment crossing the bounding box represents the quadrilateral onto which the cached image is texture-mapped, as described in Section 3.1. The viewpoint for which the cache was computed is v_0. Let a be a point inside the node. The point that corresponds to a on the quadrilateral is $\tilde{a}$. By construction, a and $\tilde{a}$ coincide when viewed from v_0; however, for most other views, the two points subtend some angle $\theta > 0$, as illustrated by viewpoint v_1 in the diagram. Our error metric measures the maximum angular discrepancy over all points

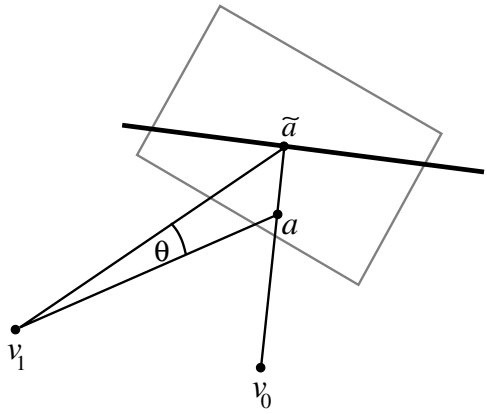


Figure 3.1: Angular discrepancy.

a inside the node:

$$Error\,(v, v_0) = \max_{a} \theta\,(a, v, \tilde{a}) \tag{3.1}$$

For a given view direction and field of view, the smaller the maximum angular discrepancy is allowed to be, the closer the projections of points *a* and $\tilde{a}$ are in the image. Thus, using a smaller error threshold results in fewer visual artifacts caused by using the cached images instead of rendering the geometry.

The right hand side of equation (3.1) may be approximated by computing the angular discrepancy for each of the eight corners of a node's bounding box. This is not a conservative estimate, but it is fast to compute, and has been found to work well in practice.

In order to predict the life span of a cached image before creating it for some view v_0, we must estimate how far from v_0 we can travel while keeping the error under ϵ. If the view trajectory is known to us in advance, we can simply search along the trajectory for the farthest point for which the error is within tolerance. This is probably the best course of action for recording a walkthrough or fly-by off-line. For an interactive walkthrough, the path of the viewer is not known in advance; however, the current velocity and acceleration are known at any frame, and an upper bound on the acceleration is typically available. In this situation, for each node in the hierarchy we can attempt to find a *safety zone* around v_0,

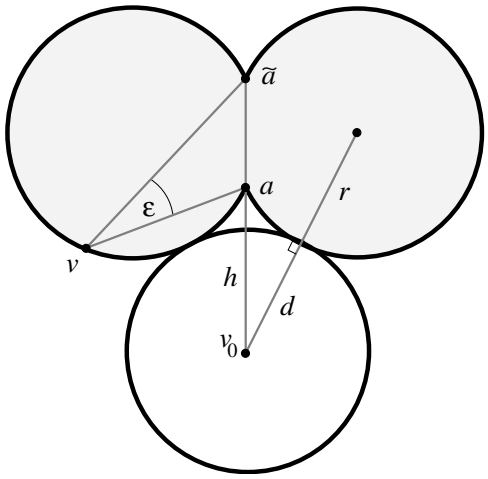


Figure 3.2: The safety zone. The shaded region contains all the viewpoints v from which a and $\tilde{a}$ subtend an angle greater than or equal to ϵ. The lower circle is a conservative safety zone.

that is, a set of viewpoints v such that for each viewpoint in this set the error is less than ϵ:

$$SafetyZone(v_0) \tag{3.2}$$

Given the safety zone and using bounds on velocity and acceleration, we can compute a lower bound on the number of frames for which the cache will remain valid. Alternatively, we can obtain a non-conservative estimate by extrapolating the viewer's path and intersecting it with the safety zone. Our implementation uses non-conservative estimates. Next we describe how safety zones are computed in our algorithm.

Consider the 2D diagram in Figure 3.2. Let v_0 be the current viewpoint, a a point inside a node, and $\tilde{a}$ its projection onto the textured quadrilateral, as in Figure 3.1. Note that all viewpoints v from which the angle subtended by a and $\tilde{a}$ is equal to ϵ must lie on one of the two circles of radius

$$r = \frac{\|a - \tilde{a}\|}{2 \sin \epsilon} \tag{3.3}$$

passing through a and $\tilde{a}$. Thus, we can conservatively define a circular safety zone around v_0 (a sphere in 3D), whose radius d is given by the shortest distance between these circles and v_0:

$$d = \sqrt{h^2 + r^2 + 2hr\sin\epsilon} - r \tag{3.4}$$

where h is the distance between v_0 and a.

In order to approximate the safety zone for a leaf node in the hierarchy, we evaluate d for each corner of the node's bounding volume and take the smallest of these distances. We then set the safety zone to be the axis-aligned cube inscribed inside a sphere of radius d around v_0. The safety zone of an interior node is computed by first computing the safety zone using the bounding box of the node, and then finding the intersection of this safety zone with the safety zones of the children.

In our implementation, the user specifies the error threshold in pixels. This threshold is converted to an angular error threshold using the current resolution and field-of-view angle. If either the resolution or the field-of-view change in the course of a walkthrough, the angular error threshold must be adjusted accordingly.

3.3 Partitioning

As a preprocessing step, we construct a BSP-tree [27] partitioning of the scene. The goals of the partitioning algorithm are as follows:

1. split as few objects as possible;

2. make the hierarchy as balanced as possible (in terms of the number of geometric primitives contained in each subtree);

3. make the aspect ratio of each node's bounding volume as close to 1 as possible.

The first goal aims to reduce visual artifacts. The second and third goals help improve performance: balanced trees facilitate hierarchical view frustum culling, and cached images

of nodes with good aspect ratios tend to remain valid longer. Computing the optimal BSP-tree that satisfies these potentially contradictory goals appears difficult. Therefore, our partitioning algorithm employs a simple greedy approach that is not optimal, but seems to work well in practice.

Given a list of objects to partition, we look for gaps between objects, place a splitting plane in the "best" gap we can find, and then recurse on the lists of objects on each side of that plane. To facilitate finding the gaps between objects, we compute their extents with a method similar to the parallelepiped bounding volumes of Kay and Kajiya [43]. For each object, we compute its extent along each of N different directions on the unit sphere. Each splitting plane in the BSP-tree is constrained to be perpendicular to one of the N vectors. For example, if we chose the three coordinate axes as our direction vectors, our partitioning algorithm would yield a binary tree of axis-aligned boxes.

For each of the N directions, we create two sorted lists of objects: one, according to the lower bound of each object's extent; the other, according to the upper bound. We then scan these lists, while keeping track of the number of "active" objects (i.e., objects whose extents we are currently in). Intervals where the number of active objects is a local minimum are the gaps that we are looking for. Ideally, we are looking for a gap with zero active objects, such that the number of geometric primitives on each side of the gap is roughly equal. Such a gap does not always exist, so we compute a cost for each gap that is a function of the number of its active objects and the ratio of the number of primitives on either side of the gap. For each of the N directions, we choose the gap with the smallest cost. To create good aspect ratios, we tend to choose the best gap from the direction along which the combined extent of all the objects on the list is greatest.

The current implementation of our system is geared towards visualization of complex landscapes. Such scenes have a special structure: they essentially consist of a height field representing land and water, and of objects such as trees and houses scattered on that height field. Thus, assuming that the positive Y axis points up, all of the objects are spread above the XZ plane. Our partitioning algorithm takes advantage of this structure by using N

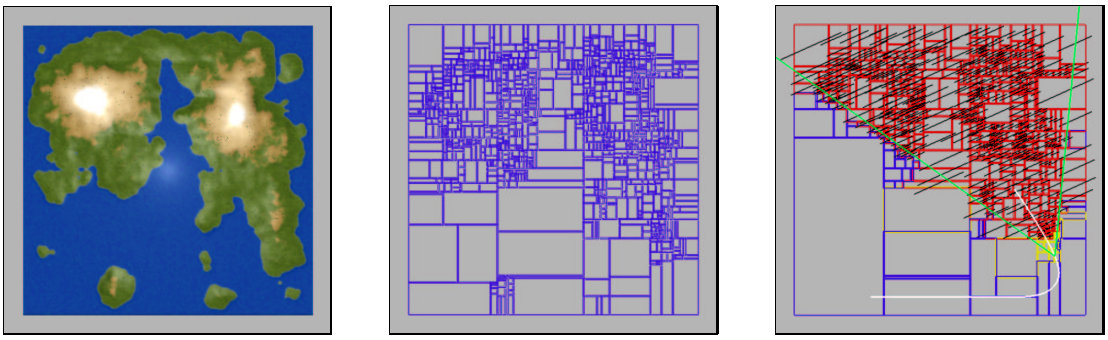
direction vectors that evenly divide the unit circle perpendicular to the *Y* axis. As a result, all of the splitting planes of the BSP-tree are perpendicular to the *XZ* plane. In all of the experiments reported in Section 3.4, two direction vectors were used, resulting in axis-aligned boundaries between regions.

When objects are split between two or more leaf nodes, visual artifacts that look like gaps or cracks sometimes appear in the split surfaces. This problem results from approximating a single object by multiple images, with no constraint that the images match along the split boundary. Such artifacts can occur even with small error thresholds because of the discrete sampling involved in creating the caches and rendering the textured quadrilaterals. For small error thresholds, it is possible to overcome these artifacts by ensuring a small amount of overlap in the geometry contained in neighboring leaf nodes. To achieve this overlap, we construct a slightly "inflated" version of each leaf region, and associate with each leaf node the extra geometry that is contained in its inflated region, in addition to the geometry contained in the original region. In our current implementation, the amount by which regions are inflated is a user-specified parameter (typically 10 to 20 percent).

3.4 Results

This section demonstrates the performance of our method using a walkthrough of a complex outdoor scene. All tests were performed on a Silicon Graphics Indigo2 workstation with a 250MHz R4400 processor, 320 megabytes of RAM, and a Maximum Impact graphics board with 4 megabytes of texture memory.

The outdoor scene used in these tests is a terrain of an island populated with 1117 willow trees. The terrain consists of 131,072 triangles, and each tree consists of 36,230 triangles. The total number of triangles in the database is 40,599,982. To keep the storage requirements down the trees were instanced, and the total amount of storage for the database before any processing by our method is 20 megabytes. The amount of storage required for this scene without instancing is 3.5 gigabytes. Figure 3.3(a) shows a bird's eye view of the



(a) (b) (c)

Figure 3.3: Partitioning. (a) A bird's eye view of the island scene. (b) The partitioning of the scene. (c) A viewpoint on the walkthrough path.

island.

Constructing the BSP-tree for this database took 46 seconds. The resulting partitioning (shown in Figure 3.3(b)) has 13 levels, 1072 leaf nodes, and is fairly balanced in terms of the geometric primitives contained in each subtree. Most leaf nodes contain a single tree and a portion of the terrain. The partitioning algorithm managed to avoid splitting any of the trees, and the only object split was the terrain.

Partitioning the database causes an increase in the required storage. This increase is primarily due to the need to "inflate" the leaf regions, as described in Section 3.3. For this database, we used an inflation factor of 17 percent, increasing the storage to 150 megabytes. Note that the increase is only 4 percent relative to the storage the original database would have required if we did not use instancing on the trees.

We recorded timings for several walkthroughs of the island. Each of the walkthroughs was along the same path, defined by a B-spline space curve shown in white in Figure 3.3(c). This path was designed to help us study the relative performance of image caching over a range of visible scene complexities: the camera first tracks along the edge of the model, then flies in toward the center of the island at treetop level. Although the path was known in advance, we did not take advantage of this information, in order to get a better sense of

how the algorithm would behave under interactive control.

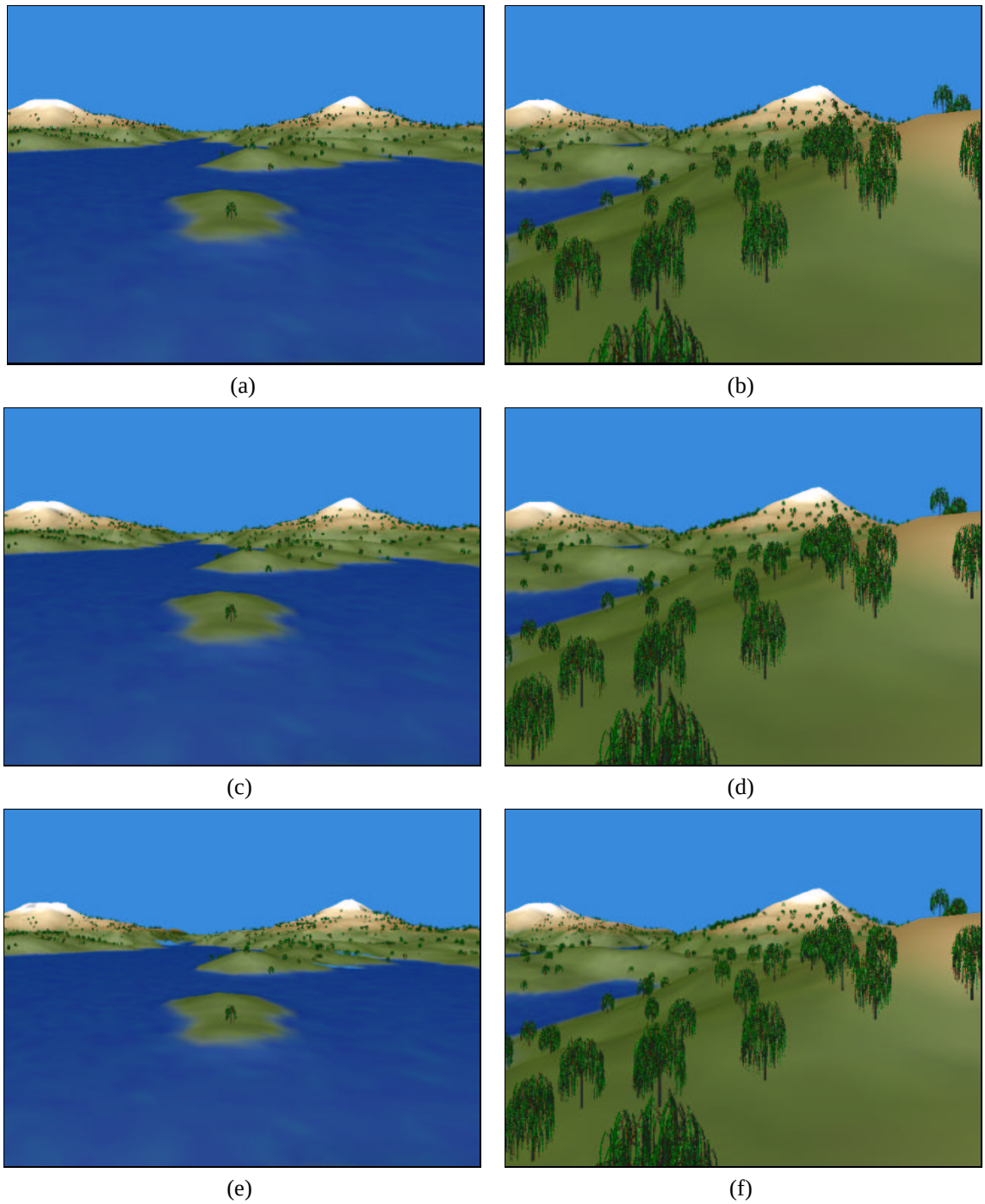
Figure 3.3(c) provides a snapshot illustrating our algorithm for a particular viewpoint on the path. The view frustum for that viewpoint is indicated by green lines. Nodes outlined in purple are culled, as they lie outside the view frustum. Nodes outlined in yellow are rendered using their geometry. Nodes outlined in red are rendered using their cached images. The quadrilaterals onto which these images are mapped are shown in black.

To assess the relative performance of our algorithm, we first computed two 1200-frame walkthroughs. Each frame was rendered at a resolution of 640×480. The first walkthrough was performed using an algorithm that employs hierarchical view frustum culling (using the same BSP-tree), but renders all of the original geometry contained in leaf nodes that are inside the view frustum. The second walkthrough was performed using our method with an error threshold of two pixels.

The top row of images in Figure 3.4 shows two different frames from the walkthrough rendered using the original geometry. The second row shows the same frames rendered by our method. The images are not identical to those in the top row, but it is very hard to tell them apart, except for the distant trees that appear slightly softer and less blocky when rendered with our method, because of the bilinear filtering used when rendering texture-mapped primitives.

The plot in Figure 3.5 shows the rendering times for the two walkthroughs. For each frame, we plot the rendering time spent by each of the two methods. It takes our method 134 seconds to compute the very first frame of the walkthrough, which is two times longer than the time required when rendering the geometry. However, once the initial image caches have been computed, subsequent frames can be rendered 4.1 to 25.2 times faster with our method, with an overall speedup factor of 11.9 for the entire sequence.

In the experiment above, our method used a fairly small error threshold: an angle subtended by roughly two pixels on the image plane. As a result, there are almost no perceptible visual artifacts in the walkthrough, as compared to rendering the geometry. If the error threshold is relaxed, more visual artifacts start to appear, but the rendering becomes



(a) (b) (c) (d) (e) (f)

Figure 3.4: Frames from walkthroughs of the island. The top row shows two frames rendered using the original geometry. The second row shows the same frames rendered with image caching using an error threshold of two pixels. The third row illustrates the visual artifacts resulting from a larger error threshold (eight pixels).

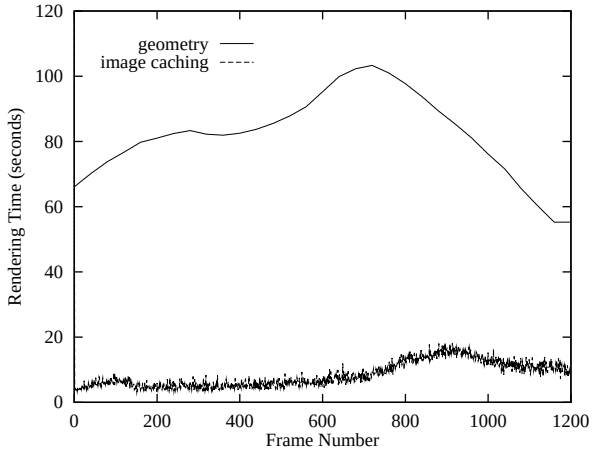


Figure 3.5: Image caching versus rendering geometry.

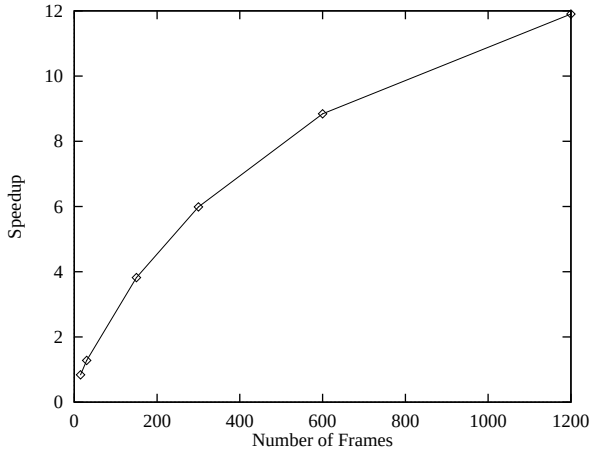


Figure 3.6: Speedup as a function of frame rate.

faster, as cached images have longer life spans. For instance, with the error threshold set to eight pixels, the overall speedup increases to 14.1. Frames that were rendered with this error threshold are shown in the bottom row of Figure 3.4. Comparing these images with the ones rendered using geometry (in the top row) reveals increased “ruggedness” along the silhouette of the mountains, as well as some “cracks” in the terrain, through which the blue background shows through.

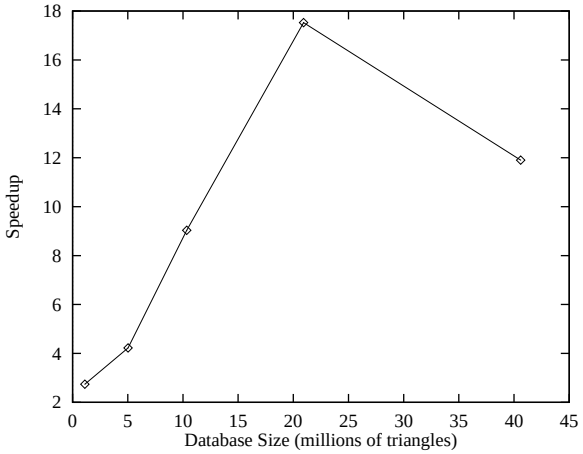


Figure 3.7: Speedup as a function of scene complexity.

Since our method utilizes path coherence, it is interesting to examine how different frame rates along the same path affect performance. Therefore, we rendered the same walkthrough using different numbers of frames, equally spaced along the path. For example, when using two frames, the first frame is computed at the beginning of the path and the second in the middle of the path. Thus, for very small numbers of frames there is not much frame-to-frame coherence at all. For each walkthrough the overall speedup factor was computed, and the results are plotted in Figure 3.6. As expected, the speedup factor becomes larger, as more frames are rendered along the same path. Note that our method is faster than geometry with as few as 30 frames along the path.

Another interesting statistic is the behavior of our method as a function of overall scene complexity. The same walkthrough path was computed for several versions of the scene, each containing a different number of trees. Except for the number of trees, all of the scenes were identical. The overall speedup factors for these scenes (for a 1200-frame walkthrough with a two pixel error threshold) are plotted in Figure 3.7. The speedup factor introduced by our algorithm first rapidly increases with the geometrical complexity of the scene, but there is a drop in the speedup when the number of triangles increases from 20 million (574 trees)

to 40 million (1117 trees). The reason for this behavior is that increasing the tree density on the island causes significantly more extra geometry to be added to each leaf node when its region is inflated. This extra geometry makes the overhead of creating a cached image for the node substantially larger.

An important limiting factor on the performance of image caching is the constraint imposed by OpenGL [63] that texture maps have dimensions in powers of 2. Because of these limitations on texture size, almost half of the pixels in the textures defined by our method go unused. The handling of so many unused pixels results in a performance penalty for our image caching method.

3.5 Discussion

Related work on accelerating the rendering of complex environments can be classified into three major categories: visibility culling, level-of-detail modeling, and image-based rendering.

Visibility culling

Visibility culling algorithms attempt to avoid drawing objects that are not visible in the image. This approach was first investigated by Clark [14], who used an object hierarchy to rapidly cull surfaces that lie outside the viewing frustum. Garlick et al. [29] applied this idea to spatial subdivisions of scenes. View frustum culling techniques are most effective when only a small part of the scene's geometry is inside the view frustum at any single frame. In a complex environment enough geometry remains inside the view frustum to overload the graphics pipeline, and additional acceleration techniques are required.

Airey et al. [2] and Teller [84] described methods for interactive walkthroughs of complex buildings that compute the potentially visible set of polygons for each room in a building. Only the potentially visible set of polygons for the room currently containing the viewer needs to be rendered at each frame. Both of these methods require a lengthy pre-

processing step for large models. Luebke and Georges [52] developed a dynamic version of this algorithm that eliminates the preprocessing. Such methods can be very effective for densely occluded polyhedral environments, such as building interiors, but they are not suited for mostly unoccluded outdoor scenes.

The hierarchical Z-buffer [35] is another approach to fast visibility culling that allows a region of the scene to be culled whenever its closest depth value is greater than those of the pixels that have already been drawn at its projected screen location. Like previous approaches, this method can achieve dramatic speed-ups for environments with significant occlusion but is less effective for largely unoccluded environments with high visible complexity, such as a landscape containing thousands of trees.

Level-of-detail modeling

Another approach for accelerating rendering is the use of multiresolution or *level-of-detail* (LOD) modeling. The idea is to render progressively coarser representations of a model as it moves further from the viewer. Such an approach has been used since the early days of flight simulators, and has more recently been incorporated in walkthrough systems for complex environments by Funkhouser and Séquin [28], Maciel and Shirley [53], and Chamberlain et al. [11].

One of the chief difficulties with the LOD approach is the problem of generating the various coarse-level representations of a model. Funkhouser and Séquin [28] created the different LOD models manually. Eck et al. [25] described methods based on wavelet analysis that can be used to automatically create reasonably accurate low detail models of surfaces. Maciel and Shirley [53] used a number of LOD representations, including geometric simplifications created by hand, texture maps, and colored bounding boxes. Chamberlain et al. [11] partitioned the scene into a spatial hierarchy of cells and associated with each cell a colored box representing its contents. Another approach to creating LOD models is described by Rossignac and Borrel [73], in which objects of arbitrary topology are simplified by collapsing groups of nearby vertices into a single representative vertex, re-

gardless of whether they belong to the same logical part.

Another problem with geometric LOD approaches is that the shading function becomes undersampled as geometry is decimated. This undersampling causes shading artifacts, especially with Gouraud shading hardware, which evaluates the shading function only at the (decreasing number of) polygon vertices.

Our approach can be thought of as a technique for automatically and dynamically creating view-dependent image-based LOD models. Among the above LOD approaches, ours is closest to that of Maciel and Shirley. However, there are several important differences. First, our approach computes LOD models on demand in a view-dependent fashion, rather than precomputing a fixed set of LOD models and using them throughout the walkthrough. Thus, we incur neither the preprocessing nor the storage costs associated with precomputed LOD models. Second, we use a spatial hierarchy rather than an object hierarchy, and our LOD models represent regions of the scene rather than individual objects. Spatial partitioning allows us to correctly depth-sort the LOD models chosen for rendering at each frame, whereas an object hierarchy can suffer from occlusion artifacts where objects overlap.

Image-based modeling and rendering

A different approach for interactive scene display is based on the idea of *view interpolation*, in which different views of a scene are rendered as a pre-processing step, and intermediate views are generated by morphing between the precomputed images in real time. Chen and Williams [13] and McMillan and Bishop [58] have demonstrated two variants of this approach for restricted movement in three-dimensional environments. Although not general purpose, these algorithms provide a viable method of rendering complex environments on machines that do not have fast graphics hardware. Images provide a method of rendering arbitrarily complex scenes in a constant amount of time. This idea is central to both of these papers and to the method we present here.

Another image-based approach, described by Regan and Pose [71], renders the scene onto the faces of a cube centered around the viewer location. Their method allows the

display to be updated very rapidly when the viewer is standing in place and looking about. They also use multiple display memories and image compositing with depth to allow different parts of an environment to be updated at different rates. Only parts of the environment that change or move significantly are re-rendered from one frame to the next, resulting in the majority of objects being rendered infrequently.

Our method can be thought of as a hierarchical extension to the method of Regan and Pose, but with more flexibility: instead of using a fixed number of possible update rates, our method updates each object at its own rate. Another important difference is that instead of simply reusing an object's image over several consecutive frames, we use texture mapping hardware to compensate for motion parallax.

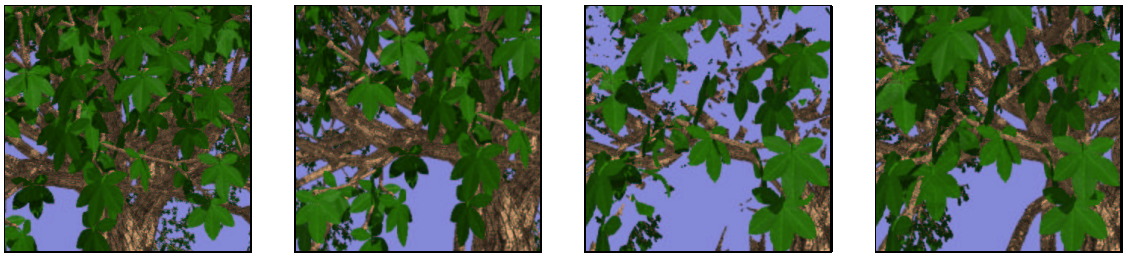
Schaufler and Stürzlinger [75, 76] have concurrently and independently investigated ideas similar to our own. Our approach differs from theirs mostly in the formulation of the error metric and in the cost-benefit analysis that we perform in order to decide whether or not to cache an image.

3.6 Summary

In summary, we have presented a new method for accelerating walkthroughs of complex environments by utilizing path coherence. We have demonstrated speedups of an order of magnitude on a high performance graphics architecture, the Indigo2 Maximum Impact. The speedups increase with the frame rate. While these speedups are significant, we believe they could be made still more dramatic through further optimizations in the underlying graphics hardware and libraries, such as improving the pixel transfer rate from the frame buffer to texture memory, relaxing the existing restrictions on texture map sizes, and providing applications with better control over texture memory management.

Chapter 4

LAYERED DEPTH IMAGES



(a) Recorded Image (b) Image cache (c) Depth image (d) LDI

Figure 4.1: Rendering from image caches, depth images, and LDIs.

This chapter describes the second view-dependent image-based representation presented in this dissertation, layered depth images [80]. Figure 4.1 shows a comparison of rendering a scene using image caches (constant-depth images), depth images and layered depth images. Image caches, presented in Chapter 3, provide a continuous reconstruction of the scene that can be rendered rapidly using standard graphics hardware. However, as Figure 4.1(b) shows, the relative positions of objects in an image cache do not change as they should when the image cache is viewed from an oblique novel viewpoint. In Figure 4.1(c) we can see that depth images remedy this error by storing an explicit depth value at each pixel of the image. This correction has the unfortunate effect of permitting a reconstruction that is no longer continuous: there are holes in the reconstructed image where surfaces that had been occluded in the recorded view are visible to the novel view. This problem can be mitigated by storing multiple samples along each ray of an image, creating a 2.5D representation. Figure 4.1(d) shows that a layered depth image is able to fill in these holes. Layered depth images provide the best of both worlds: correct geometric placement

```
type LayeredDepthImage = record
    Camera: camera
    DepthPixels[0..xres-1,0..yres-1]: array of pointer to DepthPixel
    LayerCounts[0..xres-1,0..yres-1]: array of integer
end record
```

Figure 4.2: Layered depth image data structure.

of samples and continuous reconstruction.

4.1 The LDI Representation

The LDI representation consists of three elements: the camera that defines the parameters of the image (eg. its center of projection and field of view), a compact list of samples (no storage is allocated for portions of the image that have no samples), and an indexing mechanism that allows random access to the samples in the image given a row index, a column index and a layer index. We call samples of geometry *depth pixels*. Whereas a regular image has a 2D array of pixels, an LDI has a 2D array of lists of depth pixels. The *depth pixel array* contains a null-terminated linked-list at each element of the array. Random access is performed by iterating along the lists to the appropriate layer. It is helpful to maintain a second array, the *layer count array*, that holds the length of each list (the number of depth pixels along a ray). In Section 4.5 we will present a technique for efficiently storing depth pixels that discards the linked-list and its overhead in favor of a compacted linear array that takes advantage of the layer count array for fast indexing. Figures 4.2 and 4.3 show the data structures for LDIs and depth pixels.

Distance is measured as the distance between the sample and the center of projection of the LDI camera. Z is distance along the vector that defines the direction of gaze of the camera. Storing both depth and distance from the recording camera is redundant. However,

```
type DepthPixel = record
    red: 32 bit real
    green: 32 bit real
    blue: 32 bit real
    normal x: 32 bit real
    normal y: 32 bit real
    normal z: 32 bit real
    distance: 32 bit real
    z: 32 bit real
    next: pointer to DepthPixel
end record
```

Figure 4.3: Depth pixel data structure.

the splatting reconstruction technique we employ uses distance in determining the screen-space size of a splat. We store distance to avoid incurring the expense of converting depth to distance during rendering. This version of a depth pixel requires a lot of memory; an LDI with one million samples would consume 96 megabytes of memory. In Section 4.5 we will present a quantized depth pixel that uses only 8 bytes of memory.

Our fast software-based renderer takes as input an LDI and a novel view. The renderer uses an incremental warping algorithm to efficiently create an output image using a forward warp that projects depth pixels from the LDI to the novel view frame buffer in scan line order. Reconstruction is accomplished by interpolating between the projected samples using an approximation to splatting [89].

In the rest of this chapter we will: describe the major contribution of this work, the application of McMillan's occlusion-compatible warp ordering to a sparse volume of depth pixels; present our approximation to splatting; discuss several methods for creating LDIs and show examples; describe a set of refinements and optimizations that permit real-time rendering in software on a PC; and lastly, discuss open issues and related work.

4.2 Warping LDIs

The occlusion-compatible warp ordering defined by McMillan [57] provides a means of warping the depth pixels in a depth image in such a way that they arrive at a novel view in a back-to-front order. This is deemed occlusion-compatible because we can use a simple painter's algorithm to display the projected pixels: painting the pixels in a back-to-front order results in the front-most pixels being the last to be painted, and consequently, the ones that are visible once rendering is complete. Thus, the back-most pixels are correctly occluded by the front-most pixels.

McMillan's algorithm is based on an analysis of the epipolar geometry of two images, a recorded view and a novel view. Epipolar geometry is explained in detail in Chapter 2 and is summarized here for convenience. Figure 4.4 illustrates the basic relationships in a epipolar geometry. Two cameras are shown, c_1 and c_2, that see a common point on an object in the scene. The projection of the point into the two cameras is shown along with the projection of each camera center onto the other camera's image plane. These points, labeled $e_{2,1}$ and $e_{1,2}$, are called *epipoles* [26]. The epipoles and a point in the scene define an *epipolar plane*. One such plane is shown in Figure 4.4 as the triangle connecting c_1, c_2, and a point on the object. The intersection of this plane with the image plane of a camera results in an *epipolar line* (shown as dashed lines in Figure 4.4).

McMillan shows that the depth pixels in c_1 can be projected into the view of c_2 in a back-to-front order if the depth pixels are projected along epipolar lines. If c_2 is in front of the image plane of c_1, then the depth pixels in c_1 are warped along epipolar lines starting from the edges of the image going towards the epipole. If c_2 is behind the image plane of c_1, then the order is reversed: we start at the epipole and work outwards towards the edges of the image. Since each of the epipolar lines is independent, we can warp the lines in any order we wish so long as the direction of flow is correct (towards or away from the epipole). This allows us to warp the epipolar lines in parallel by visiting them in scanline order. McMillan enumerates the 18 cases that specify the correct scanline order for performing the warping.

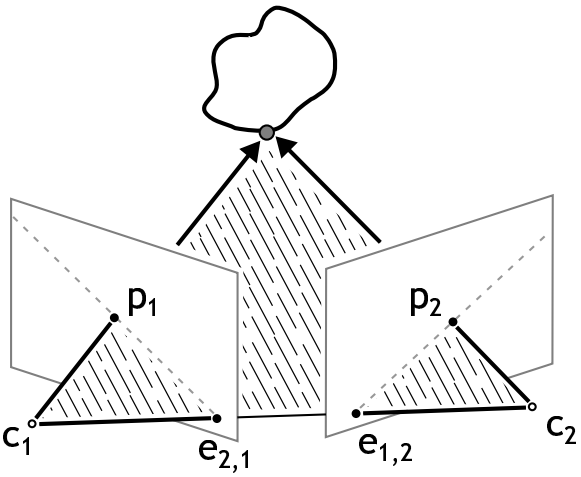


Figure 4.4: Epipolar geometry. The centers of projection of two cameras, c_1 and c_2, and a point on an object, p, form an epipolar plane. The image of one camera's center in the other camera's image plane is called an epipole. The epipole $e_{2,1}$ ($e_{1,2}$) is the projection of c_2 (c_1) onto the image plane of c_1 (c_2). The intersection of an epipolar plane and the image plane of a camera is an epipolar line. The projection into a camera of any point lying on a particular epipolar plane in the scene must lie on the corresponding epipolar line. The points p_1 and p_2 show two such projections.

Our proof that McMillan's warp ordering can be used with LDIs is by inspection. Since each epipolar line can be warped independently, we need only consider a single line of depth pixels. Figure 4.5 shows scenarios of using McMillan's warp ordering for both a depth image and a layered depth image. In this case, the novel view is to the right of the recorded view, so the warp ordering will be a left-to-right traversal of the depth pixels. In addition, for the LDI, the depth pixels in each ray will be warped in a back-to-front order. Let's consider the depth image case first. As the depth pixels from the recorded view are warped to the novel view, the depth pixels warped early in the order that should be obscured will get overwritten by depth pixels warped later in the order. In Figure 4.5(a) we see that depth pixel 4 correctly overwrites depth pixel 3 and depth pixel 5 correctly overwrites depth pixel 1. Rendering the LDI is just as easy. Since the LDI is more dense, we have labeled just a few of the depth pixels. As the warping proceeds from left-to-right, we should see

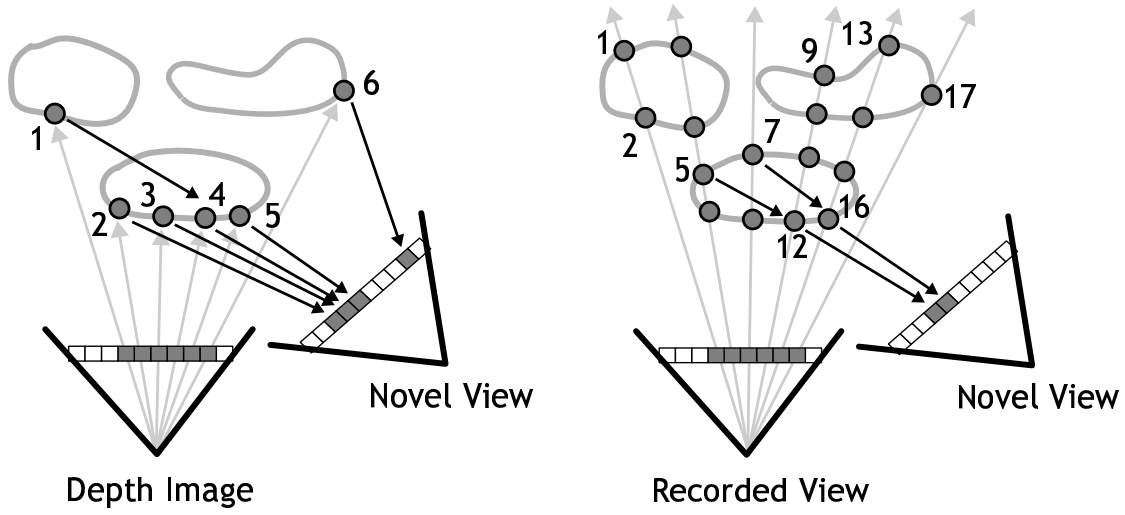


Figure 4.5: Occlusion-compatible warp order. Depth image shown in (a). Layered depth image shown in (b). In both instances, since the novel view is to the right of the recorded view, the warp order is left-to-right over the depth pixels in the recorded view. A selection of the depth pixels are numbered according to their position in the warp ordering. In the instance of the depth image shown in (a) we see that depth pixel 5 correctly overwrites depth pixel 1. In the instance of the layered depth image, we see that a second layer depth pixel, 7, is correctly overwritten by a first layer depth pixel, 16.

the same behavior as in the depth image, but, in addition, we should also see that depth pixels beyond the first layer get correctly occluded in the novel view. In Figure 4.5(b), this behavior is seen when depth pixel 12, a first layer depth pixel, overwrites depth pixel 5, a second layer depth pixel. The same behavior can been seen when depth pixel 16 overwrites depth pixel 7. In fact, if you trace the ray from 16 to 7 back into the scene, you'll see that both of these depth pixels have correctly overwritten depth pixel 1 as well.

This example illustrates one of the drawbacks to using McMillan's warp ordering as a means of computing visibility: in order to reconstruct an image we must warp every depth pixel in an LDI, even if a large percentage of those depth pixels get overwritten. This reveals two fundamental inefficiencies inherent in any forward warping algorithm: wasting computation resources warping depth pixels that never land in the screen window of the

novel camera, and wasting memory bandwidth writing to a pixel in the output image more than once.

We can avoiding warping pixels that we know will never appear in the novel view by culling and clipping. To cull, we subdivide an LDI and compute world-space bounds for all of the depth pixels in that portion of the LDI. If the bounding volume of the subdivision does not intersect the viewing frustum, we do not warp any of the depth pixels it containts. Furthermore, as each depth pixel is warped, we clip it to the view camera's screen window. If the pixel is outside the view, we skip the splat size calculation and splat rasterization procedures. Section 4.5.2 presents the details of these techniques.

Overdraw is the average number of times a pixel in the frame-buffer is written during a single frame of rendering. Since memory bandwidth is a significant factor in determining the performance of a rendering algorithm, limiting overdraw is an important goal. A back-to-front rendering algorithm presents the worst case situation for limiting overdraw. Reversing the order of warping and projecting depth pixels in a front-to-back order can greatly reduce overdraw. However, this requires the use of a z-buffer[10] to determine visibility. A z-buffer stores the depth of each pixel in the frame-buffer. If an incoming pixel has a depth greater than the one in the z-buffer, it is rejected. Otherwise, it is composited into the frame-buffer and the z-buffer is updated with the new pixel's depth. Using a z-buffer requires a read-compare-write operation per pixel. If the z-buffer rejects a warped depth pixel, we execute a read and a compare per pixel. If the z-buffer accepts a warped depth pixel, we execute a read, a compare, and a write per-pixel in the z-buffer and a read, a modify, and a write per-pixel in the frame-buffer. On the other hand, warping back-to-front excutes a read, a modify, and a write per-pixel in only the frame-buffer. But, it does so for every sample. When using a z-buffer, rejection is faster while acceptance is slower. The effectiveness of a software z-buffer implementation depends on the ratio of acceptance to rejection and the cache effects of accessing a second large array. Since the processors we have targeted, consumer-level PCs, have relatively small (256K to 512K) secondary caches, we have speculated that the adverse affects of cache conflicts between the framebuffer and

the z-buffer would result in little performance improvement when using a z-buffer. This, of course, would be dramatically different if a hardware implementation of the warping were available. Hardware systems use dedicated fast memories for the framebuffer and z-buffer and would undoubtedly benefit from a front-to-back rendering order.

4.2.1 Incremental Warping Computation

After determining the order in which scanlines will be warped, we compute per-scanline warping constants and loop over all of the depth pixels in a scanline, projecting them to the novel view. This section describes how to formulate the 3D warping equation as an incremental calculation. The incremental formulation is computationally more efficient because the portion of the computation that is constant across a scanline is factored out and computed just once per scanline.

Let C_1 be the 4×4 matrix for the LDI camera and C_2 be the 4×4 matrix for the novel view. Both camera matrices are composed of an affine transformation matrix (A), a projection matrix (P), and a viewport matrix (V), such that $C = V \cdot P \cdot A$. These camera matrices transform a point from the world-space coordinate system into the camera's projected image coordinate system. For instance, the projected image coordinates (x_1, y_1), obtained after multiplying a point's homogeneous world-space coordinates by C_1 and dividing out w_1, index a screen pixel address. The z_1 coordinate can be used for depth comparisons in a z buffer.

The depth pixels in an LDI are stored using projected image coordinates. The x and y coordinates are implicitly represented by the row and column in which the depth pixel is stored. The z coordinate is stored explicitly with each depth pixel. To warp depth pixels to a novel view, conceptually, we perform two operations. First, we invert the projection of the depth pixel: we transform the depth pixel from projected image coordinates to the world-space coordinate system. This is done by multiplying the point's homogeneous projected coordinates by the inverse of the LDI camera matrix. Second, we use the result of the first step and project the depth pixel from the world-space coordinate system to the projected

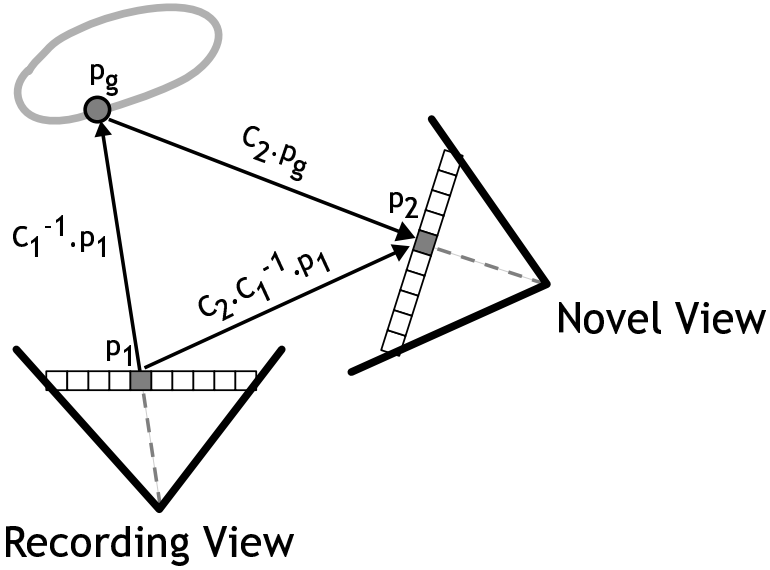


Figure 4.6: Depth pixel warping. Warping a depth pixel consists of two operations: projecting the depth pixel from screen coordinates of the recording camera into world space and projecting from world space into the screen coordinates of the novel camera. The projections can be combined into a single operation that projects directly from the recording camera to the novel camera.

image coordinate system of the novel view and normalize by dividing the projected point by its homogeneous coordinate, *w*. The two projections are illustrated in Figure 4.6 and can be described by the following equations.

$$\begin{aligned} C_1^{-1} \cdot p_1 &= p_g \\ C_2 \cdot p_g &= p_2 \end{aligned} \tag{4.1}$$

We can simplify this process by combining the two projections into a single operation. Replacing p_g in the second equation with the computation from the first equation gives us a single, combined projection that transforms projected image coordinates from one camera image plane to a second camera image plane.

$$C_2 \cdot C_1^{-1} \cdot p_1 = p_2 \tag{4.2}$$

We define this composite matrix to be the *transfer* matrix: $T_{1,2} = C_2 \cdot C_1^{-1}$. Given the

projected image coordinates of some point seen in the LDI camera, this matrix computes the image coordinates as seen in the novel view camera (e.g., the image coordinates of p_2 in camera C_2 in Figure 4.6).

$$T_{1,2} \cdot \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} = \begin{bmatrix} x_2 \cdot w_2 \\ y_2 \cdot w_2 \\ z_2 \cdot w_2 \\ w_2 \end{bmatrix} = \overrightarrow{result}$$

As before, the coordinates (x_2, y_2) obtained after dividing by w_2, index a pixel address in the output camera's image.

Using the linearity of matrix operations, this matrix multiply can be factored to reuse much of the computation from each iteration through the layers of a ray of depth pixels; $\overrightarrow{result}$ can be computed as

$$T_{1,2} \cdot \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} = T_{1,2} \cdot \begin{bmatrix} x_1 \\ y_1 \\ 0 \\ 1 \end{bmatrix} + z_1 \cdot T_{1,2} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \overrightarrow{start} + z_1 \cdot \overrightarrow{depth}$$

To compute the warped position of the next depth pixel along a scanline, the new $\overrightarrow{start}$ is simply incremented.

$$T_{1,2} \cdot \begin{bmatrix} x_1 + 1 \\ y_1 \\ 0 \\ 1 \end{bmatrix} = T_{1,2} \cdot \begin{bmatrix} x_1 \\ y_1 \\ 0 \\ 1 \end{bmatrix} + T_{1,2} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \overrightarrow{start} + \overrightarrow{xincr}$$

The warping algorithm proceeds using McMillan's ordering algorithm [57]. The LDI is broken up into four regions above and below and to the left and right of the epipole. For

```
procedure WarpScanline (row, startColumn, endColumn, start, depth, xincr)
    for col ←startColumn to endColumn
        for layer ←0 to LayerCount(row,col) - 1
            dpix ←GetDepthPixel(row,col,layer)
            result ←start + dpix.z ∗ depth
            //cull if the depth pixel goes behind the novel view camera
            //or if the depth pixel goes out of the novel view camera's frustum
            if result.w > 0 and IsInViewport(result) then
                result ←result / result.w
                size ←ComputeSplatSize(dpix.distance, results.z, dpix.normal)
                RasterizeSplat(dpix.ColorRGBA, results.x, results.y, size)
            end if
        end for // over layers
        // increment for next ray on this scan line
        start ←start + xincr
    end for // over columns
end procedure
```

Figure 4.7: Procedure for warping a depth pixel.

each quadrant, the LDI is traversed in (possibly reverse) scan line order. At the beginning of each scan line, $\overrightarrow{start}$ is computed. The sign of $\overrightarrow{xincr}$ is determined by the direction of processing in this quadrant. Each scan line of depth pixels is then warped to the output image by calling *WarpScanline*. This procedure visits each of the layers in each ray of depth pixels in back-to-front order and computes $\overrightarrow{result}$ to determine the location of a projected depth pixel in the output image. As in perspective texture mapping, a divide is required per pixel. Finally, the depth pixel's projected size is calculated, and it is splatted at this location in the output image. The pseudo code in Figure 4.7 summarizes the rendering algorithm applied to each scanline of depth pixels.

4.3 Splatting

After a depth pixel has been warped, we determine the screen-space area of the sample in the novel view and composite a splat [89] into the frame buffer using the *over* operator [68]. Splats are typically simple shapes that can be rendered easily such as squares or discs with feathered edges. If a scene has been adequately sampled the splats will overlap, creating a continuous reconstruction.

In order to splat a point sample the sample is treated as a geometric object that acts as a proxy for the actual geometry surrounding a sample. By definition, this object projects to a disk one pixel in area in the recording camera. We can construct an object that satisfies this constraint by inverting the projection. A pixel is treated as a screen-space disk on the image plane of the recording camera and projected onto the world-space plane defined by the normal of the point sample. This yields an elliptical disk in world space. Projecting this disk onto the image plane of the novel camera yields an ellipse in screen space. This is the splat shape that should be composited into the framebuffer. This process has much in common with texture mapping algorithms [33].

This mapping is too expensive to compute in real time, so we use an approximation make to make the computation faster. We assume that the projection of the world-space elliptical disk is a disk in the novel view. This allows us to reduce the computation to computing the ratio of the splat area in the novel view to the splat area in the recording view.

The ratio of the area, in pixels, of a splat in the recording view to the area of a splat in the novel view can be computed (differentially) as

$$darea = \frac{(d_1)^2\, cos(\theta_2)\, res_2\, tan(fov_1/2)}{(d_2)^2\, cos(\theta_1)\, res_1\, tan(fov_2/2)}$$

where d_1 is the distance from the sampled surface point to the LDI camera, fov_1 is the field of view of the LDI camera, $res_1 = (w_1 h_1)^{-1}$ where w_1 and h_1 are the width and height of the LDI, and θ_1 is the angle between the surface normal and the line of sight to the LDI camera (see Figure 4.8). The same terms with subscript 2 refer to the output camera. Since splats

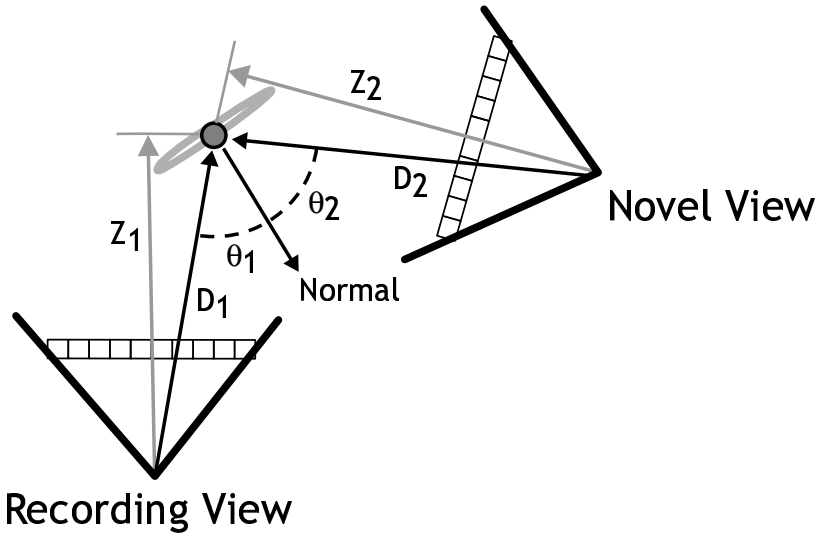


Figure 4.8: Values for size computation of a projected pixel. D is the distance from a camera to the sample. Z is the depth from a camera to the sample. θ is the angle between the normal of the sample and the direction from which the sample is viewed.

in the recording camera always have an area of one pixel, the ratio is area of a splat in the novel view.

The values fov_1, fov_2, res_1, and res_2 are constant, and d_1 is stored in the depth pixel. While warping, only d_2, $cos(\theta_2)$, and $cos(\theta_1)$ need to be computed. The angle between the normal of a depth pixel and the ray of the recording camera, $cos(\theta_1)$, can be computed incrementally as a scan line is being warped. We approximate d_2 by Z_2, the z coordinate of the depth pixel in the output camera's unprojected eye coordinate system. During rendering, we set the projection matrix such that $z_2 = 1/Z_2$. Since warped depth pixels project to essentially random places in the output image, we can't incrementally compute $cos(\theta_2)$.

In practice, we limit the size of a splat to be no less than one pixel and no greater than ten pixels square. The upper limit is necessary because the ratio of projected areas can lead to very large splats (larger than the viewing window of the novel camera). If the normal of a sample is tangential to the ray of the recording camera on which it lies, the recording camera will view a very small projected area. When a novel camera views that same pixel

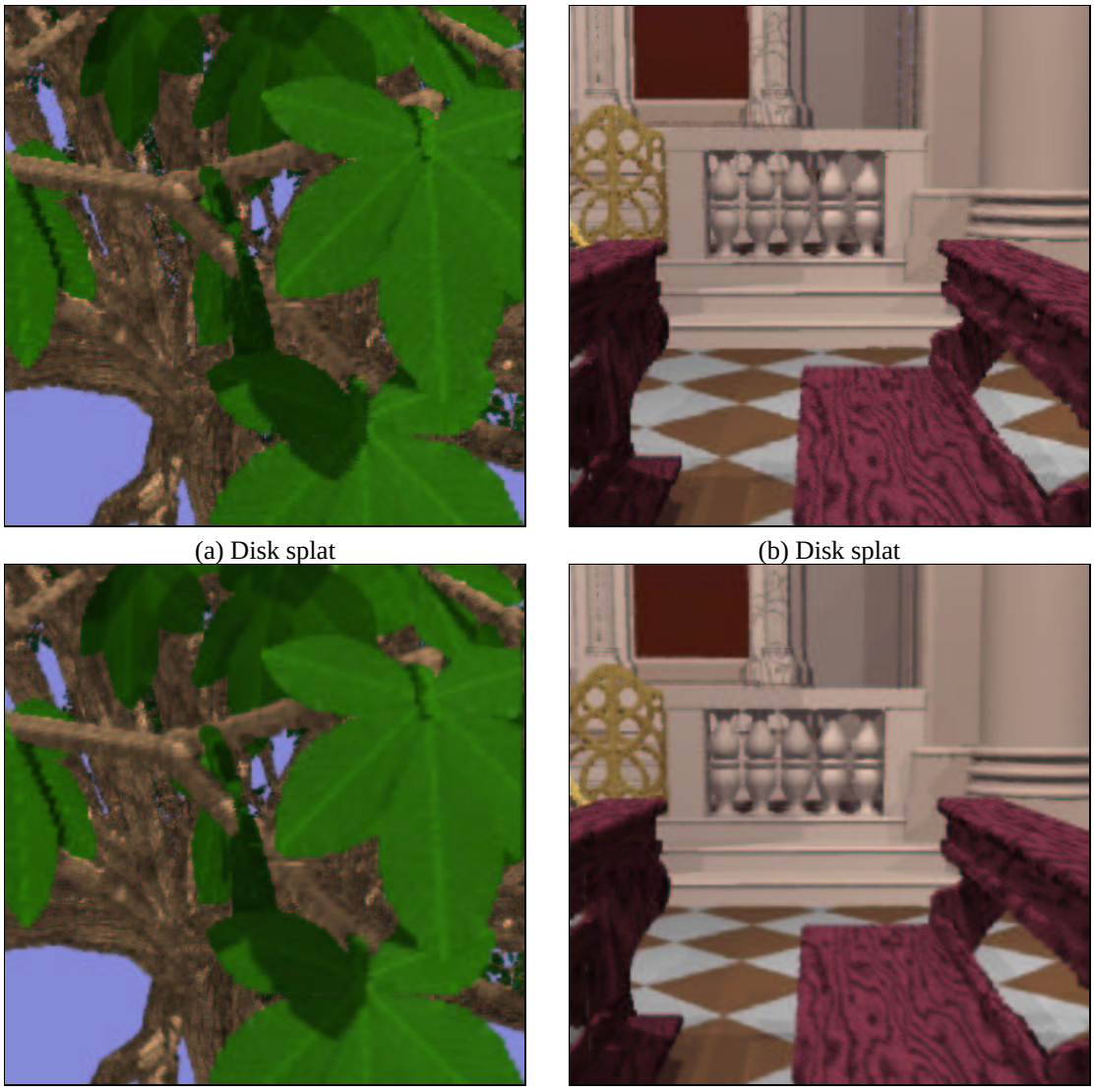
head-on, the ratio of projected areas will become very large.

4.3.1 Splat rasterization

We have experimented with two shapes of splats: a constant-color disk and a disk with an opacity that falls off as a Gaussian function. The Gaussian we use falls off to a value of one half at the radius of the splat. Since the disk is opaque, the rasterizer simply paints the color of the splat into the frame buffer rather than using the *over* operator. The splats are rendered using rasterization functions that have been specialized to particular splat resolutions. We would expect the disk splat to be faster than the Gaussian because of it's simpler rasterization function. In practice, there is little difference between the two. The chestnut tree in Figure 4.9, renders at 12 frames per second using disk splats and at 11 frames per second using Gaussian splats.

The original splatting paper by Westover [89] used high resolution pre-computed tables of floating point numbers to represent sampled Gaussian reconstruction kernels. These tables were used to render warped Gaussian splats. Westover's splatting algorithm computed the elliptical projection of a splat then performed a backwards warp of each pixel of the splat to the normalized space of the sampled Gaussian. Much like texture mapping algorithms, the projection of a pixel after being backward-mapped was used to sample and filter the Gaussian texture.

In contrast, our software renderer relies on a frame buffer of packed integer colors and uses integer operations to implement the *over* compositing operation. The values of sampled reconstruction kernels are quantized to 1, 3/4, 1/2, or 1/4 so that compositing can be done with integer shift and add instructions. In addition, we do not filter the sampled reconstruction kernels at runtime. Instead, we pre-compute a fixed set of sampled reconstruction kernels that match the resolution needed to rasterize a splat. Our system employs kernels that range in resolution from 1×1 pixels to 10×10 pixels. As a consequence of these simplifications, our implementation of splatting is clearly a rough approximation to the algorithm presented by Westover; the benefit of our implementation is that runs in real



(a) Disk splat (b) Disk splat

(c) Gaussian splat (d) Gaussian splat

Figure 4.9: Splat shape comparison. The difference between the two shapes is most noticeable at silhouette edges.

time.

A table-lookup-driven rasterizer would incur significant overhead due to looping and references to table memories. To mitigate this, we have written specialized rasterization functions that have the sampled kernel weights embedded in the code. These functions are

created programmatically from a kernel and a range of desired splat resolutions.

Our rasterization function writer creates two sets of functions: the first assumes that a splat lands at the center of a pixel in the novel image, and the second assumes a splat lands on the border between two pixels in the novel image. During rendering, a simple test determines the sub-pixel position of the splat and chooses the appropriate rasterization function.

Sub-pixel placement of splats alleviates two artifacts that arise from our exact point-sampling of reconstruction kernels. If splats are snapped to pixel centers, they will appear to ”pop” as the LDI is viewed in motion. The splats will jitter and wiggle across the image. In addition, as the LDI is viewed at an angle, adjacent splats that should overlap will be pulled apart and a gap will appear between them creating moire patterns.

4.3.2 Separating visibility and reconstruction

An unfortunate consequence of considering one sample at time is that we cannot, in the strict sense, properly reconstruct the scene. Levoy and Whitted [49] discuss this problem in the context of a point rendering system for curved surfaces. To properly reconstruct a scene from a set of points we should first *add* a set of interpolating reconstruction kernels centered at the sample points of a single surface. After all of the surfaces have been reconstructed, we should then composite them in back-to-front order using the *over* operator. In this way, surface reconstruction and visible surface determination are separated into two steps. One solution to this problem uses a z-buffer. First, the scene is rendered into a z-buffer using opaque splats that write the depth of the sample. Second, the scene is rendered as normal, but as splats are rendered only those pixels with depth within a small epsilon of the value recorded in the z-buffer are rendered. These samples are assumed to be from the same surface and are rasterized using the *add* operator. We have not employed this method because it would require too much computation for a real-time software-based renderer.

4.4 Creating Layered Depth Images

There are a variety of ways to generate an LDI. Given a synthetic scene, we could use multiple images from nearby points of view for which depth information is available at each pixel. This information can be gathered from a standard ray tracer that returns depth per pixel or from a scan conversion and z-buffer algorithm where the z-buffer is also returned. Alternatively, we could use a ray tracer to sample an environment in a less regular way and then store computed ray intersections in the LDI structure. Given multiple real images, we can turn to computer vision techniques that can infer pixel correspondence and thus deduce depth values per pixel. We will demonstrate results from each of these three methods.

4.4.1 LDIs from Multiple Depth Images

We can construct an LDI by warping *n* depth images into a common camera view. If, during the warp from an input camera to the LDI camera, two or more pixels map to the same ray of depth pixels, their Z values are compared. If the Z values differ by more than a preset epsilon, a new depth pixel is inserted in the list of depth pixels for that ray. Otherwise, the values of the two depth pixels are averaged, resulting in a single depth pixel. This preprocessing is similar to the rendering described by Max [56]. This construction of the LDI is effectively decoupled from the final rendering of images from desired viewpoints. Thus, the LDI construction does not need to run at multiple frames per second to allow interactive camera motion.

Figure 4.10 shows two views of a barnyard scene modeled in Softimage. A set of 20 images was pre-rendered from cameras that encircle the chicken using the Mental Ray renderer. The renderer returns colors, depths, and normals at each pixel. The images were rendered at 320 by 320 pixel resolution, taking approximately one minute each to generate. In the interactive system, the 3 images out of the 20 that have the closest direction to the current camera are chosen. The preprocessor (running in a low-priority thread) uses these images to create an LDI in about 1 second. The average depth complexity for these LDIs

Figure 4.10: Barnyard scene. The background in the image on the right is black because the viewer has looked beyond the bounds of the portion of the scene captured in the current LDI, and the LDI-generating thread has not finished creating a new LDI that will fill in the gap.

is only 1.24. Thus the use of three input images only increases the rendering cost by 24 percent. The fast renderer (running concurrently in a high-priority thread) generates images at 300 by 300 resolution. On a Pentium II PC running at 300MHz, we achieved frame rate of 8 to 10 frames per second.

4.4.2 LDIs from a Modified Ray Tracer

By construction, an LDI reconstructs images of a scene well if the novel view is coincident with the recording view (simply display the nearest depth pixels). The quality of the reconstruction from another viewpoint will depend on how closely the distribution of depth pixels in the LDI, when warped to the new viewpoint, corresponds to the pixel density in the new image. Two common events that occur are: (1) disocclusions as the viewpoint changes, and (2) surfaces that grow in terms of screen space. For example, when a surface is edge on to the LDI, it covers no area. Later, it may face the new viewpoint and thus cover some screen space.

When using a ray tracer, we have the freedom to sample the scene with any distribution

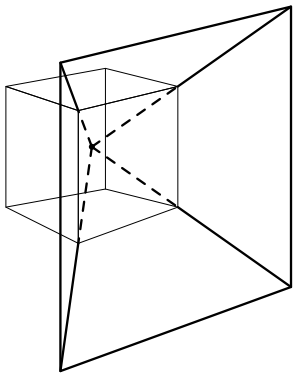


Figure 4.11: An LDI consists of the 90 degree frustum exiting one side of a cube. The cube represents the region of interest in which the viewer will be able to move.

of rays we desire. We could simply allow the rays emanating from the center of the LDI to pierce surfaces, recording each hit along the way (up to some maximum). This would solve the disocclusion problem but would not effectively sample surfaces tangential to a ray.

What set of rays should we trace to sample the scene, to best approximate the distribution of rays from all possible viewpoints of interest? This depends on the resolution of the novel view and the region in which the novel view is allowed to move. In this sense, sampling is output-driven. We set the parameters of sampling to match the parameters of reconstruction.

The resolution of the LDI is matched to the resolution of the novel view. We set the resolution of the LDI so that, when the novel view is coincident with the LDI, the size of a pixel in the novel view is the same as a pixel in the LDI.

For simplicity, we have chosen to use a cubical region of empty space surrounding the LDI center of projection to represent the region of allowed movement for the viewer. Each face of the viewing cube defines a 90 degree frustum which we will use to define a single LDI (Figure 4.11). The six faces of the viewing cube thus cover all of space. For the following discussion we will refer to a single LDI.

Each ray in free space has four coordinates, two for position and two for direction. Since all rays of interest intersect the cube faces, we will choose the outward intersection

Figure 4.12: Examples of LDIs. The top row shows a chestnut tree provided by Radomir Mech. The bottom row shows a sunflower field provided by Oliver Deussen. Both data sets were created using the stratified stochastic sampling algorithm with N and M set to 16 and 64.

to parameterize the position of the ray. Direction is parameterized by two angles.

Given no *a priori* knowledge of the geometry in the scene, we assume that every ray intersecting the cube is equally important. To achieve a uniform density of rays we sample the positional coordinates uniformly. A uniform distribution over the hemisphere of directions requires that the probability of choosing a direction is proportional to the *projected*

Figure 4.13: More LDIs. The top row shows an outdoor landscape provided by Oliver Deussen. The bottom row shows views of the Il Redentore church model provided by Nathan O'Brien. Both data sets were created using the stratified stochastic sampling algorithm with N and M set to 16 and 64.

area in that direction. Thus, the direction is weighted by the cosine of the angle off the normal to the cube face.

Choosing a cosine-weighted direction over a hemisphere can be accomplished by uniformly sampling the unit disk formed by the base of the hemisphere to get two coordinates of the ray direction, say x and y if the z-axis is normal to the disk. The third coordinate is

Table 4.1: Rendering performance. Measurements are given for frames per second, depth pixels flowed per second, depth pixels splatted per second, number of depth pixels in the model, number of depth pixels flowed, and number of depth pixels splatted.

Scene	**FPS**	**PFPS**	**SPS**	**NDP**	**DPF**	**DPS**
Chesnut tree	7.7	3.3M	2.2M	1,725,595	446,986	294,884
Sunflower field	13	3.5M	2.1M	1,361,863	259,567	148,386
Il Rendertore	14	3.2M	2.2M	1,284,615	221,761	152,538
Landscape	9.5	3.6M	2.5M	1,574,193	393,430	264,897
Dinosaur	32	3.5M	3.5M	106,548	106,548	106,548
Flower	45	3.1M	3.1M	67,424	67,424	67,424

chosen to give a unit length ($z = \sqrt{1 - x^2 - y^2}$). We make the selection within the disk by first selecting a point in the unit square, then applying a measure preserving mapping [62] that maps the unit square to the unit disk.

Given this desired distribution of rays, there are several ways to perform the sampling:

Uniform. A straightforward stochastic method would take as input the number of rays to cast. Then, for each ray it would choose an origin on the cube face and a direction from the cosine distribution and cast the ray into the scene. There are two problems with this simple scheme. First, such *white noise* distributions tend to form unwanted clumps. Second, since there is no coherence between rays, complex scenes require considerable memory thrashing since rays will access the database in a random way [67].

Stratified Stochastic. To improve the coherence and distribution of rays, we employ a stratified scheme. In this method, we divide the 4D space of rays uniformly into a grid of $N \times N \times N \times N$ strata. For each stratum, we cast M rays. Enough coherence exists within a stratum that swapping of the data set is alleviated. Typical values for N and M are 16 and 64, generating approximately 8 million rays per cube face.

Once a ray is chosen, we cast it into the scene. If it hits an object, and that object lies in the LDI's frustum, we reproject the intersection into the LDI. If the new sample is within an epsilon tolerance in depth of an existing depth pixel, the color of the new sample is

averaged with the existing depth pixel. Otherwise, the color, normal, and distance to the sample create a new depth pixel that is inserted into the LDI.

Examples of LDIs created using stratified stochastic sampling are shown in Figures 4.12 and 4.13. The LDIs were created using a modified version of the Rayshade [44] raytracer. A server with a 2.8 GHz Intel Xeon processor and 2GB of memory was used to sample the data sets. The chestnut scene took 9 minutes to sample. The sunflower scene took 8 hours. The landscape scene took 105 minutes. The scene of Il Redentore took 20 minutes.

Table 4.1 shows performance measurements taken of our renderer. The primary determinant of performance is the number of depth pixels that get splatted. Many of the depth pixels that are warped fall outside of the view of the novel camera and are clipped. The majority of the computation done for a depth pixel is in splat size calculation and rasterization. The rendering optimizations discussed in the next section were used in all of the examples shown.

4.4.3 LDIs from Real Images

The dinosaur and flower models shown in Figure 4.14 were constructed from 21 photographs of the object undergoing a 360 degree rotation on a computer-controlled calibrated turntable. An adaptation of Seitz and Dyer's voxel coloring algorithm [78] is used to obtain the LDI representation directly from the input images. The regular voxelization of Seitz and Dyer is replaced by a view-centered voxelization similar to the LDI structure. The procedure entails moving outward on rays from the LDI camera center and projecting candidate voxels back into the input images. If all input images agree on a color, this voxel is filled as a depth pixel in the LDI structure. This approach enables straightforward construction of LDI's from images that do not contain depth per pixel. The dinosaur model contains 106,548 depth pixels and renders at 30 frames per second. The flower model contains 67,424 thousand depth pixels and renders at 45 frames per second.

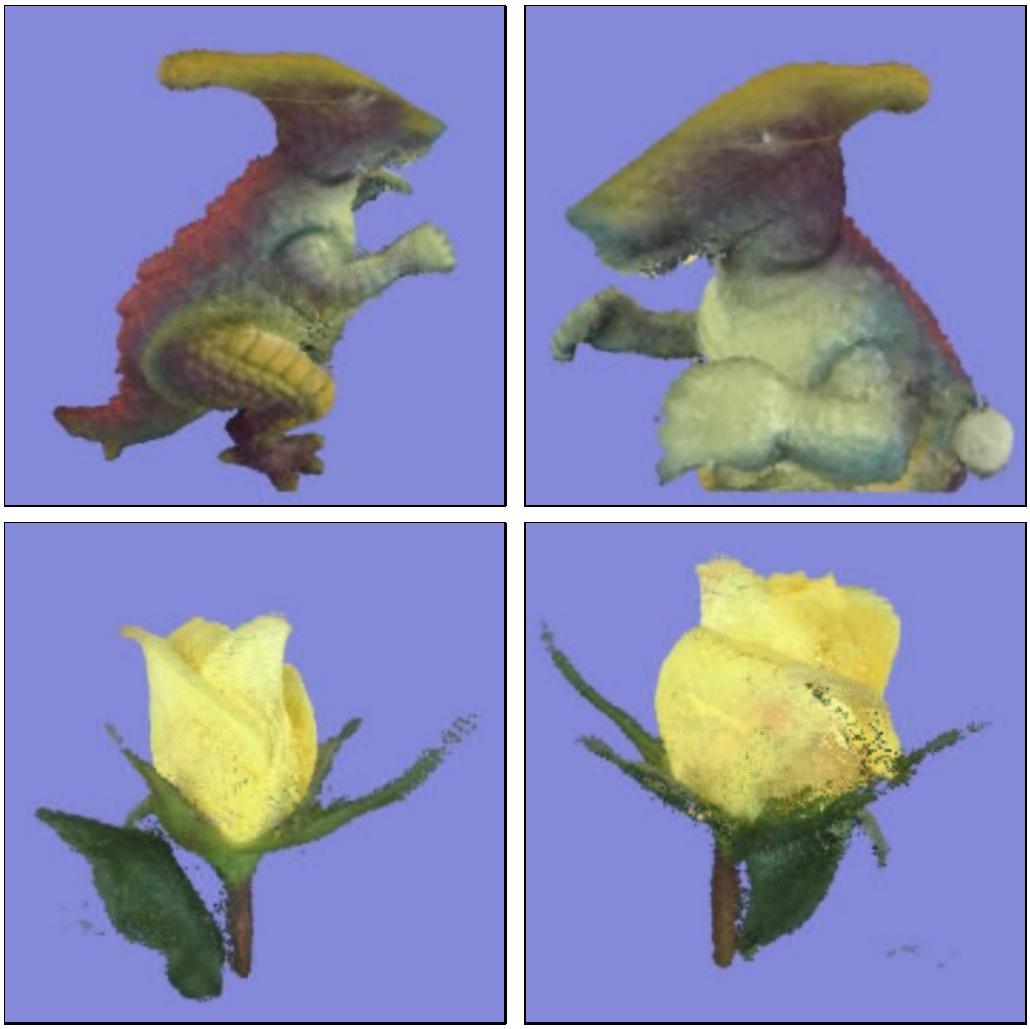


Figure 4.14: LDIs created from real objects. The toy dinosaur and flower models were constructed from 21 photographs. The models were provided by Steve Seitz.

4.5 Optimizations

In this section we will describe optimizations to the LDI data structure and rendering algorithms that permit real-time rendering on PCs using a software-only renderer.

```
type LayeredDepthImage = record
    Camera: camera
    RowOffsets[0..yres-1]: array of 32 bit integer
    ColumnOffsets[0..xres,0..yres-1]: array of 16 bit integer
    DepthPixels[0..xres-1,0..yres-1]: array of DepthPixel
end record
```

Figure 4.15: Run-time LDI data structure.

4.5.1 Space Efficient LDIs

In practice, we implement LDIs in two ways. When creating LDIs, it is important to be able to efficiently insert and delete depth pixels. As illustrated in Figure 4.2, the *DepthPixels* array in the *LayeredDepthImage* structure is implemented as a linked list. When rendering, an LDI is used as a read-only data structure and efficiently accessing the depth pixels is paramount. In the context of an LDI this is accomplished in two ways: direct access to depth pixels instead of traversing a list, and quantizing depth pixels to maximize the use of the second level cache in the CPU [45].

Double Vector Index Array

Traversing a linked list to access a depth pixel is slow for two reasons: traversing the list is an $O(n)$ operation where n is the number of depth pixels, and the overhead of storing a "next" pointer becomes significant when the depth pixels are quantized. Replacing the linked list data structure with an array mitigates both of these problems.

We reorganize the depth pixels into a linear array ordered from bottom to top and left to right in screen space, and back to front along a ray. We also replace the *LayerCounts* array with a double array of offsets into the linear array of depth pixels. Using two arrays allows us to reduce the precision of the integers in the *LayerCounts* array from 32 bits to 16 bits. This halves the size of the array and increases the chance that it will fit into the second (or

```
type DepthPixel = record
    ColorRGB: 24 bit integer
    Normal Index: 8 bit integer
    Z: 16 bit integer
    Distance: 16 bit integer
end record
```

Figure 4.16: Run-time depth pixel data structure.

third) level cache in the CPU. Figure 4.15 shows the run-time LDI data structure:

The number of depth pixels in each scanline is accumulated into a vector of offsets at the beginning of each scanline, *ColumnOffsets*. Within each scanline, for each ray of depth pixels, a total count of the depth pixels from the beginning of the scanline to that location is maintained in *RowOffsets*. Thus to find the beginning address of any ray of depth pixels, one simply offsets to the beginning of the scanline and then further to the first depth pixel at that location. To find the number of depth pixels in the ray, this offset is subtracted from the offset in the ray one entry to the right. This supports scanning in left-to-right and right-to-left order as well as the clipping operations discussed later.

Quantizing the Depth Pixel

When rendering, it is important to make effective use of the second level cache in the CPU. The size of a cache line on current Intel processors, the Pentium family, is 32 bytes. To fit four depth pixels into a single cache line we quantize the floating point *Z* and *Distance* values to 16 bit integers and pack them into a single word. In addition the red, green and blue color values are quantized to 8 bits each. Lastly, the normal vector is replaced by a 7 bit index into a table of 128 full-precision normal vectors that are uniformly distributed over the unit sphere. The color and normal index are packed together in a single word. Figure 4.16 shows the run-time depth pixel data structure. It is redundant to store both

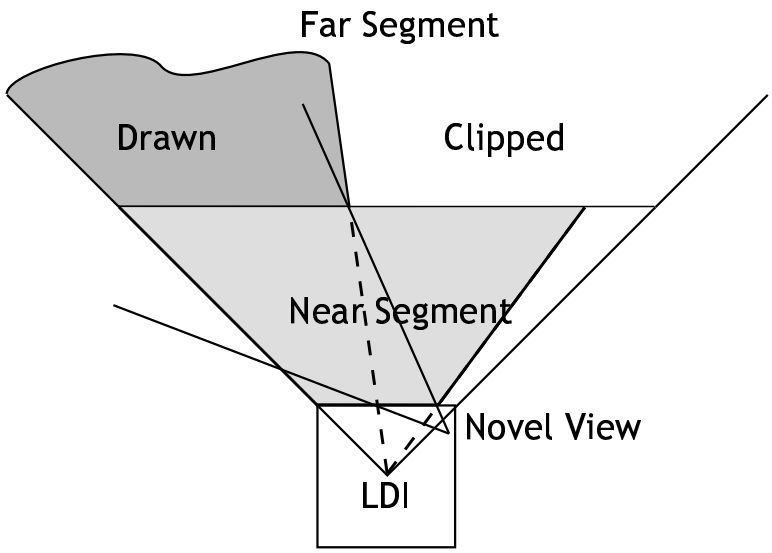


Figure 4.17: LDI with two segments.

depth and distance. It is sufficient to store depth and convert it to distance at runtime, but this conversion is an expensive computation. We have traded storage for speed.

Quantizing d_1 linearly leads to rendering artifacts. The quantization becomes obvious because the transition from larger splat sizes in the near field to smaller splat sizes in the far field is not smooth. To increase the accuracy of the approximation for d_1, we discretize d_1 nonlinearly using a simple exponential function that allocates more bits to the nearby d_1 values, and fewer bits to the distant d_1 values.

There is great lattitude in deciding how to allocate bits among these quantized values, and we settled on this distribution through trial and error. We have intentionally designed this representation so that 4 depth pixels fill a cache line. Data structures that straddle cache lines are likely to incur significant performance penalties. An earlier version of our renderer used depth pixels that were 9 bytes in size. Changing to an 8 byte depth pixel, and making no other change, yielded a 25 percent improvement in rendering speed.

Figure 4.18: A two segment LDI. The left image shows both segments being rendered. The middle image shows the front segment. The right image shows the back segment.

4.5.2 Clipping and Culling

The most effective optimization of a computation is avoiding doing the computation in the first place. Clipping and culling are used to weed out depth pixels that will not be seen in the novel view. Avoiding the warping computation on these pixels provides a significant speedup. We present three techniques for weeding out unseen depth pixels.

Frustum-frustum Clipping

The LDI of the chestnut tree scene in Figure 4.18 is a large data set containing over 1.7 million depth pixels. If we naively render this LDI by reprojecting every depth pixel, we would only be able to render at three frames per second. When the viewer is close to the tree, there is no need to flow those pixels that will fall outside of the new view. Unseen pixels can be culled by intersecting the view frustum with the frustum of the LDI. This is implemented by intersecting the view frustum with the near and far plane of the LDI frustum, and taking the bounding box of the intersection. This region defines the rays of depth pixels that could be seen in the new view. This computation is conservative, and gives suboptimal results when the viewer is looking at the LDI from the side (see Figure 4.17). The view frustum intersects almost the entire cross section of the LDI frustum, but only

Table 4.2: Performance of clipping and culling techniques. The second column shows the rendering performance of an LDI without any clipping or culling. FF refers to frustum-frustum clipping. RC refers to region-culling. SC refers to scanline culling. NDP-F is the number of depth pixels in the front segment, and NDP-B is the number of depth pixels in the back segment.

Scene	**FPS**	**FF**	**RC**	**SC**	**NDP-F**	**NDP-B**
Chesnut tree	3	7.7	7.1	5	301,426	1,424,169
Sunflower field	4.2	12.5	13.5	8	296,632	1,065,231
Il Rendertore	4.5	14.5	13.5	8	233,302	1,051,313
Landscape	3.5	9.5	9	7	353,286	1,220,907

those depth pixels in the desired view need be warped. Our simple clipping test indicates that most of the LDI needs to be warped. To alleviate this, we split the LDI into two segments, a near and a far segment (see Figure 4.17). These are simply two frustra stacked one on top of the other. The near frustum is kept smaller than the back segment. We clip each segment individually, and render the back segment first and the front segment second. Clipping can speed rendering times by a factor of 3. Figure 4.18 shows an LDI rendered in two segments.

Scanline Culling

We can cull (reject) an entire scanline of pixels by constructing a 3D polygon that contains all of the depth pixels in a scanline. This bounding polygon is constructed by recording the maximum and minimum depth values found in a scanline along with the first and last non-empty columns. These four values define a bounding polygon in world space that encloses all of the samples in a scanline. To cull the scanline, we project the polygon to the novel view and use standard view frustum culling techniques to determine if any portion of the polygon intersects the viewing volume of the novel view. Since the LDI is static, we can precompute the per-scanline polygons and store them with the LDI.

Region Culling

The scanline culling technique can be extended to cull 3D sub-volumes of an LDI. We divide the image plane of the LDI into $n \times n$ square regions. Each region represents a sub-volume (a beam) of the LDI that extends from the image plane to the furthest sample found in the region. Like the scanline culling technique, we construtct this polyhedron by recording the closest and furthest depth values found in the region and the first and last non-empty rows and columns in the region. These six values define a bounding polyhedron in world space that encloses all of the samples in a region. Again, using standard view-frustum culling techniques, we project the vertices of the polyhedron to the novel view and cull the region if it does not intersect the viewing volume of the novel view.

Scanline culling and region culling are conservative: the bounding geometry may not provide a tight fit to the distribution of samples in the scanline or region. A more accurate bounding volume such as Kay-Kajiya bounding slabs [43] would result in a region being culled more often, but at a higher computational cost in the culling test.

Table 4.2 shows timings for the clipping and culling techniques. Frustum-frustum clipping was the most effective in three of the tests and region culling was most effective in one test. Scanline culling can be used in conjunction with frustum-frustum clipping and region culling, but there is a negligible benefit from doing so. While frustum-frustum clipping is typically faster than region culling, it is very difficult to implement. Region culling is trivial to implement.

4.6 Summary

There have been many papers on image based rendering that are related to LDIs. The incremental warping computation is similar to the ones used for texture mapping operations [38, 77]. The geometry of this computation was analyzed by Mcmillan [57], and an efficient computation for the special case of orthographic input images is given in [19].

In [49], Levoy and Whitted present a system that renders geometry as a collection of

points. They discretize geometry by distributing points over the surface of an object, and render using an algorithm similar to splatting. The discussion of separating visibility and reconstruction in Section 4.3.2 draws heavily from a similar discussion in this paper.

Chen and Williams presented the idea of rendering from images [13]. McMillan and Bishop extended this work to images using cylindrical projections [58]. Both systems present techniques for synthesizing a view from more than one source image. Laveau and Faugeras discuss IBR using a backwards map [46]. Seitz and Dyer describe a system that allows a user to correctly model view transforms in a user controlled image morphing system [79].

Max uses a representation similar to an LDI [56], but for a purpose quite different than ours; his purpose is high quality anti-aliasing, while our goal is efficiency. Max reports his rendering time as 5 minutes per frame while our goal is multiple frames per second. Max warps from *n* input LDIs, each with different camera information; the multiple depth layers serve to represent the high depth complexity of trees. We warp from a single LDI, so that the warping can be done most efficiently. For output, Max warps to an LDI. This is done so that, in conjunction with an A-buffer, high quality, but somewhat expensive, anti-aliasing of the output picture can be performed.

Mark et al.[54] and Darsa et al.[20] create triangulated depth maps from input images with per-pixel depth. Darsa concentrates on limiting the number of triangles by looking for depth coherence across regions of pixels. This triangle mesh is then rendered traditionally taking advantage of graphics hardware pipelines. Mark et al.describe the use of multiple input images as well. In this aspect of their work, specific triangles are given lowered priority if there is a large discontinuity in depth across neighboring pixels. In this case, if another image fills in the same area with a triangle of higher priority, the latter is used instead. These papers offer refinements to the triangulated depth image representation presented in Chapter 2.

All of these representations, however, do not explicitly account for certain variations of scene appearance with viewpoint, e.g., specularities, transparency, etc. View-dependent

texture maps [21], and 4D representations such as lightfields or Lumigraphs [48, 32], have been designed to model such effects. These techniques can lead to greater realism than LDIs, but usually require more effort (and time) to render.

The feature that distinguishes LDIs from these other works is the combination of points with an image-based parameterization. Extending McMillan's occlusion-compatible warp ordering to LDIs permits software-based real-time rendering of complex scenes. LDIs are very good at rendering nearby objects with high depth complexity. The advantages of the LDI representation decrease for objects that are far from the camera. In contrast, our first image-based representation, image caches, works very well for objects in the distance but relies on standard polygon rendering for nearby objects. These two ideas are synergistic and reinforce a fundamental premise of image-based rendering: put effort where effort is due. Objects that change in appearance slowly can be represented by simple approximations, such as image caches, while objects that change rapidly need to be represented by more complex approximations, such as LDIs.

Chapter 5

TILING LAYERED DEPTH IMAGES

Modeling and rendering scenes that capture the complexity of the real world is not an easy task. The manual effort required to model natural environments and the computational cost required to render them are both very high. This chapter presents a technique that simplifies the modeling and rendering of realistic terrains. Combining layered depth images with a novel scheme for creating non-periodic tilings of the plane, we show how a small set of prototype 3D textures can be combined to produce an expansive, naturalistic scene [16].

In this system LDIs are used to represent very detailed, small portions of the vegetation that covers a terrain. By properly stitching together the LDIs with a tiling scheme we are able to produce a vast expanse of vegetation that has the appearance of being globally unique. There are no repetitive patterns that expose the fact that only 8 unique tiles are being used to give the illusion of a vast expanse of globally unique texture. By caching and reusing the geometry in an LDI we have reduced the time needed to model a complex scene to the time needed to model a very small scene. Figure 5.1 shows an example rendering.

5.1 Tiling

The inspiration for this work was the image in Figure 5.2(a). This scene consists of about seven thousand randomly placed sunflowers. There are only eleven unique sunflower models; instancing is employed to avoid modeling each flower individually. We first constructed a standard layered depth image of the scene. This provides a very satisfying result and allows camera motion near the LDI center, but the quality of the rendering breaks down quickly away from this point, as seen in Figure 5.2(b).

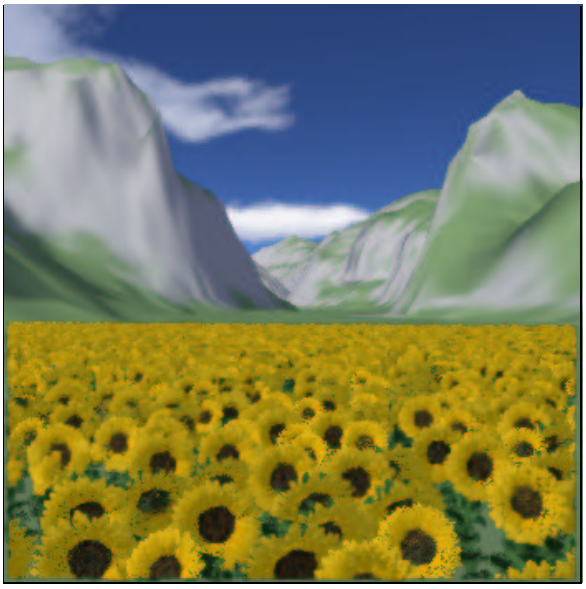
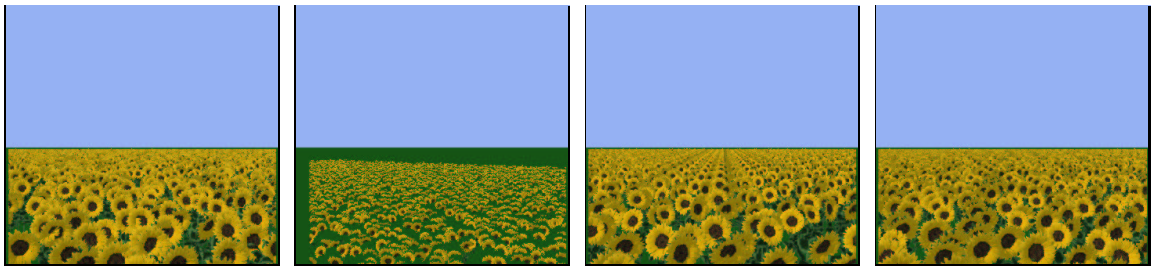


Figure 5.1: Yosemite valley covered in sunflowers.

Our first attempt at modeling this scene using tiles was to use just one tile. A tile in this case consists of a square plot of terrain with about 20 sunflowers in it. This type of tile has toroidal edge constraints: the north side matches the south side and the east side matches the west side. Figure 5.10, at the end of the chapter, shows sets of tiles similar to this one. Figure 5.2(c) shows a rendering of the tiled scene. There is a pronounced "corn row" effect in the image, and the periodicity in the tiling is obvious. After this initial failure, we turned to the study of tilings to help us create tiled sunflowers that would look as natural as Figure 5.2(a).

A *tiling of the plane* is a countable family of tiles $\mathcal{T} = \{T_1, T_2, \ldots\}$ which cover the plane without gaps or overlaps [36]. In other words, every point on the plane must be a member of some T_i for some i, and the intersection of any two tiles must be empty. Tiles can take many different shapes, from triangles to squares to polygons. In this paper we use a simple class of square tiles called Wang tiles [86, 87].

A recent development in the theory of tilings has demonstrated the existance of sets of prototiles which admit infinitely many tilings of the plane, none of which are periodic.



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

Figure 5.2: Modeling with LDI tiles. (a) A Layered Depth Image of the scene that inspired this work, (b) the same LDI from a different view, (c) modeling the scene using a single tile, (d) modeling the scene using the 8 Wang tiles introduced in this paper.

The first such set of prototiles discovered was comprised of tiles known as the Wang tiles. Wang tiles are square tiles with colored edges. The edges of any two adjacent tiles in a tiling must match, and the tiling must consist only of translations of the prototiles. Rotations and reflections are not allowed. In the 20^{th} century, Wang had conjectured that no aperiodic sets existed, where an *aperiodic* set was one which admitted only *non-periodic* tilings of the plane (i.e. no valid tiling of the plane is periodic). The first known set of aperiodic Wang tiles was discovered by R. Berger in 1966 [8]. Berger's original set had 20,426 prototiles. He later reduced this number to 104, and until recently, the smallest known aperiodic set had 16 prototiles [72].

Wang tiles are interesting theoretically, because it is possible to find sets of Wang tiles that mimic the behavior of any Turing machine. They are interesting to us because sets of Wang tiles have been discovered with as few as 13 prototiles [39, 41]. Having fewer tiles is desirable because the geometry in each tile is very detailed and requires a lot of storage space.

After our initial excitement regarding the idea of using Wang tiles, we found that although the tilings are aperiodic, at least the small Wang tile sets display a marked structure (see Figure 5.4(c)) which is exactly what we wanted to avoid. Rather than give up on this path we tried to create a small set of Wang tiles that could tile the plane simply and at least

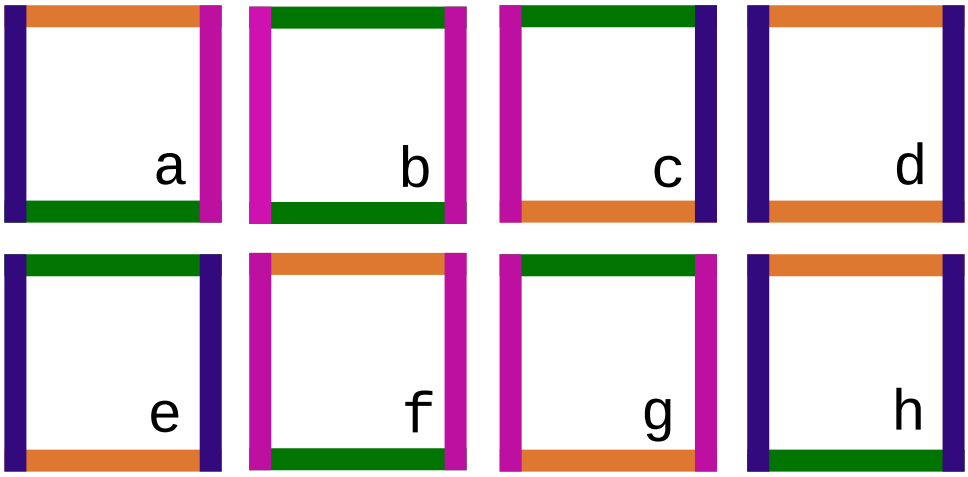


Figure 5.3: The set of 8 proposed prototiles.

not appear periodic and not display any obvious structure. We found one such set of 8 tiles shown in Figure 5.3. A tiling is created with a very simply algorithm:

1. Choose a tile at random and place it in the lower left corner

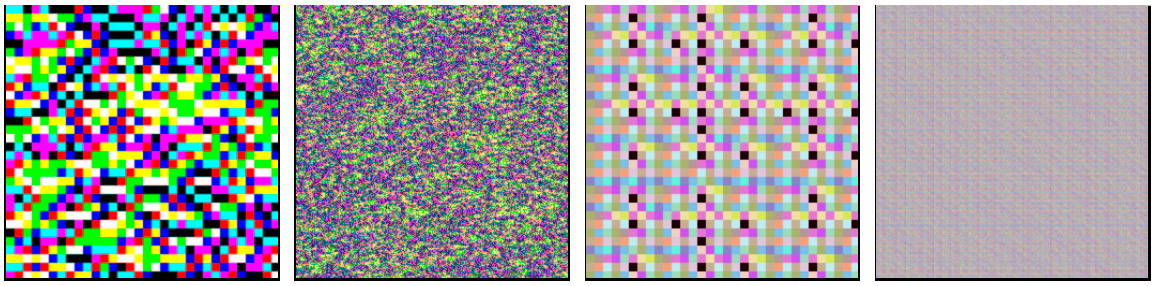
2. For the bottom row, choose compatible tiles from left to right (i.e., the west edge must match the previous east edge). If more than one choice is possible, choose randomly amongst compatible tiles.

3. For each row above the bottom row

    1. Choose the first tile to be compatible with the one below it (i.e., the south edge must match the north edge from below)

    2. Complete the row with tiles that match both the west and south edges, to tiles on the left and below.

Somewhat to our surprise this worked. A tiling of the plane with a random choice of color assigned to each prototile is shown in Figure 5.4(a). Larger scale tilings using our prototiles and Robinson's 16 are shown in Figure 5.4 (b) and (d) respectively. The large scale tiling of



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

Figure 5.4: A comparison of tilings. Small scale (32x32) and large scale (256x256) tilings using (a and b) the 8 Wang tiles proposed in this paper and (c and d) Robinson's set of 16 aperiodic Wang tiles.

the Robinson tiles shows a marked plaid-like structure with strong horizontal and vertical features, while the large scale tiling of our tiles looks similar to white noise. Figure 5.2(d) shows a rendering of the sunflower scene modeled using this tile set.

This set of eight tiles is clearly not strictly aperiodic as one could create a tiling of the plane using only one of the tiles that has the same color on its north and south edges and on its east and west edges. However, our stochastic construction procedure prevents such a degenerate tiling from appearing in practice. We have generated valid tilings of ten thousand tiles on a side using our stochastic tiles.

5.2 Modeling

In our demonstrations, we model the terrain surface as a set of objects such as sunflowers, dandelions, and blades of grass in a random-close-packed arrangement. For visual variety, each type of terrain object has a number of versions. The sunflower scene is made up of 11 versions of the flower, while the grassy scene is made up of 15 versions of grass, 9 versions of the dandelion and 15 versions of the yellow flower. Each type of object has a radius that determines how densely they will be distributed.

We approximate this terrain model with an aperiodic arrangement of a small number of

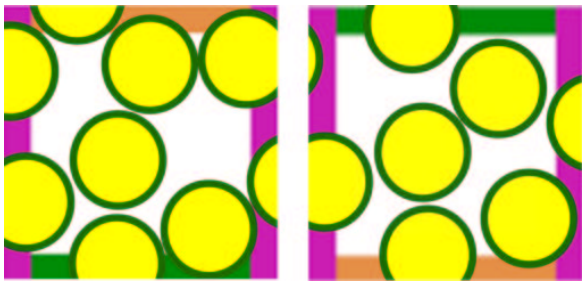


Figure 5.5: Poisson disk distributions for possible neighboring tiles.

square tiles. For each of those tiles, we must construct a geometrical model made up of an arrangement of terrain objects that will be consistent with the tile's edge-color boundary condition. If one tile has a purple east edge, it could appear next to any one of several other tiles having a purple west edge (Figure 5.5). We would like the random-close-packed arrangement (i.e., a Poisson-disk distribution) to extend across the tile boundary no matter which of the purple-west-edge tiles happens to be adjacent. If the terrain object distributions in each tile do not mesh with neighboring tiles, the result is a highly visible, periodic disruption of the terrain along the grid lines between tiles.

To achieve consistency with the edge-color boundary conditions, we use a stochastic dart-throwing process to produce a set of points (terrain-object locations) in and around each tile. The dart-throwing process visits each tile in round-robin order and attempts to add one new object to that tile, until a sufficient density of objects have been placed in each. This round-robin processing insures that the tiles have almost identical object density. During each visit to a tile, up to 10,000 attempts are made to insert a new object at a uniformly distributed random location.

An attempt succeeds if the new object's radius does not overlap any other object's radius either within the tile or in potentially neighboring tiles. If this is true, it is added to the tile and we may move on to the next tile. When the new object's radius lies completely within the tile, Figure 5.6(a), the new object's radius need only be check against other points associated with that tile. If the new object's radius extends beyond an edge of the tile (e.g.,

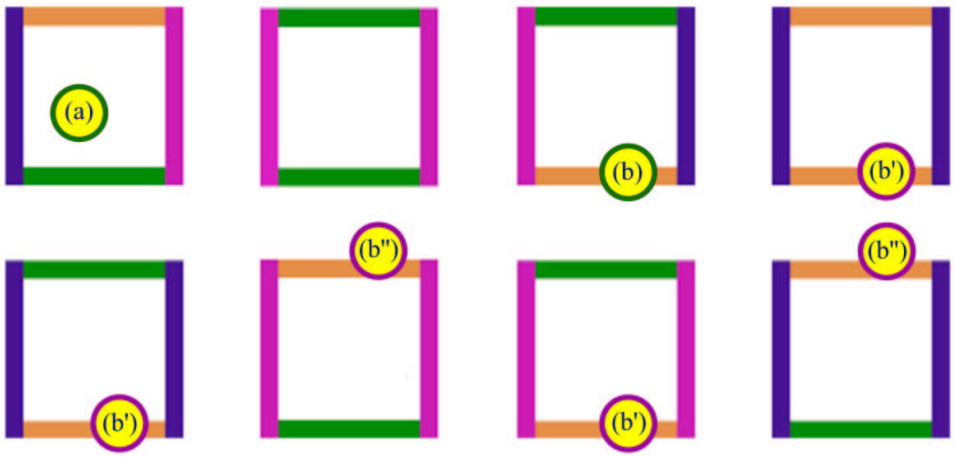


Figure 5.6: Placing objects in tiles.

an orange south edge), Figure 5.6(b'), then an attempt must be made to add this new object into

- all other tiles that have the same edge color for the same edge (e.g., all orange south edges), Figure 5.6(b') and

- and all potential neighbors across the edge (e.g., all orange north edges), Figure 5.6(b").

In the latter check, the new object's location is outside the tile as if it were in the neighboring tile (e.g., across the blue south edge). Note that tiles can now have points associated with them that will lie just outside their boundary; however only the portion of the corresponding terrain object that protrudes into the tile will be rendered. All of the above conditions must be true for the new object to be accepted.

This treatment of the edge-color boundary condition means that the distribution of terrain objects is identical near each colored edge. Since the tiles are aperiodic, these edges are randomly distributed and do not appear to create a periodic visual artifact in the final scene. However, there is one remaining problem. If a terrain object protrudes beyond a

corner of a tile, it could overlap any or all other tiles in the tile set. That might force us to place a terrain object in the same location near the corner of all tiles, and that would produce a periodic artifact in the scene. To address this problem, we assign two radii to each terrain object. One radius tightly bounds the extent of the object's geometry, while a slightly larger radius is actually used to control the packing density, giving a slight buffer zone. We do not allow the tight radius of any terrain object to protrude across a corner of a tile, but we do allow the buffer to cross a corner. The result is that objects crowd in slightly to avoid creating a periodic void in the distribution at each tile corner.

We have used a simple terrain model based on a Poisson-disk distribution of plant species. Deussen et al. [23] present several more advanced models, based on plant population dynamics and terrain topography (elevation, slope, closeness to water, etc.). Modeling these phenomena presents an interesting, unsolved challenge to a real-time tile-based approach. Possible future work would experiment with using larger numbers of tiles and edge colors, and associtate a range of plant densities with each edge color. This may afford enough lattitude to adapt the tiling to local topographical conditions.

5.3 Representation

Once we have finished placing instances of the plant models in the tiles, we are faced with the task of creating a run-time representation of each tile. There are several characteristics we want of such a representation. It should

- render as realistically as possible,
- look good at many different screen space resolutions,
- look good from many different angles, and
- be able to render at interactive rates.

Our solution is to build multiresolution multi-view layered depth images (MRMVLDI), a collection of LDIs sampled at varying resolutions and from varying directions. When rendering a tile, we choose the best LDI from this set based on the direction from which the novel camera views the tile and the distance from the novel camera to the tile.

LDIs are a natural fit for creating a view-dependent level-of-detail hierarchy. By construction, LDIs store only those samples that are necessary for a limited range of views. Using a collection of LDIs covering the entire range of views surrounding an object allows us to turn a model that is too detailed to render in real time into a collection of sampled representations, each of which *can* be rendered in real time. This segmentation and sampling effectively precomputes visible surface determination. In addition, LDIs fit nicely with the need to create lower resolution representations for objects that are far from the camera. To create a lower resolution model we simply reduce the resolution of the LDI.

The LDIs we built for this system use an orthographic camera configured to look top-down at a square plot of the solid texture we wish to tile across a terrain. The orthographic frustum fits naturally with square tiles. Figure 5.10 shows top-down orthographic and off-to-the-side perspective views of the eight tiles that were sampled and tiled to create the sunflower scene.

Each MRMVLDI is parameterized first by distance to the camera (resolution) and then by viewing direction (range of views used to sample the tile). We divide the range of viewing directions into eight evenly sized wedges that encircle the tile. Using the stochastic sampling strategy presented in Chapter 4 we sample the tile with rays that cover the views of a camera that sweeps out one eighth of a circle surrounding the tile. As the level of detail decreases, the radius of the circle increases and the resolution of the LDI decreases.

The resolution chosen for each level of detail is driven by the parameters of the run-time system. At run time the viewer is allowed to wander around a terrain at a fixed height above the terrain. To ensure that the rendering does not become overly blurry, we require that splats are never larger than five pixels in area. Along with the output camera resolution and field of view, these two parameters completely specify the resolution needed for a tile.

(a) (b) (c)

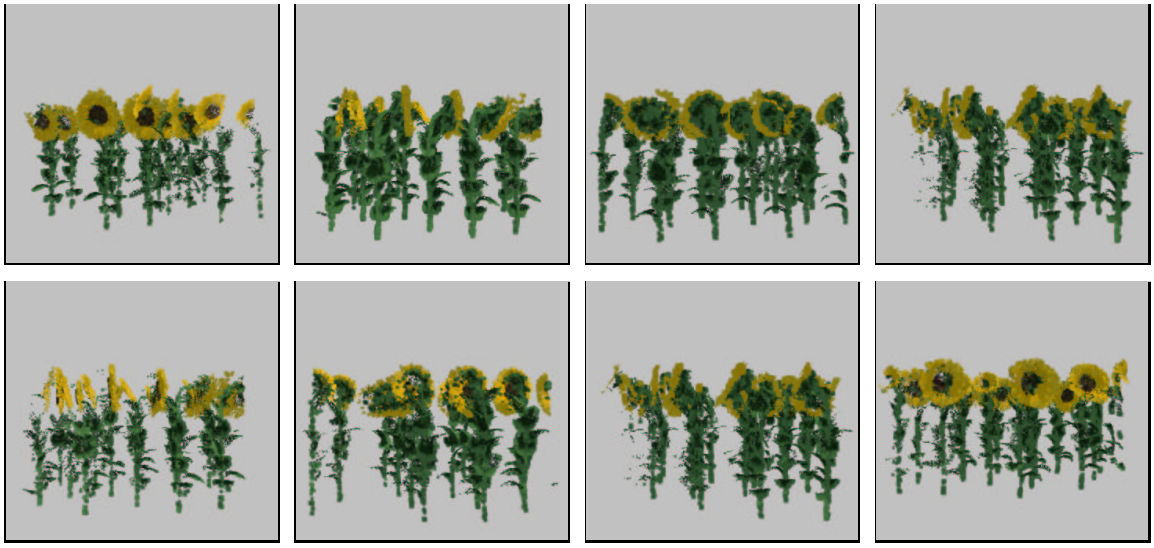
Figure 5.7: Multiresolution LDI. (a) A tile sampled at a low resolution rendered at the correct distance. (b) The same resolution LDI rendered close to the camera. (c) The proper resolution LDI for the close-by position.

The closest a tile will ever come to the viewer is the difference between the height of the viewer above the terrain and the height of the geometry in a tile. If the highest resolution tile is placed five times this distance from the viewer, the splats will be one pixel in area. This is the distance we use to sample the tile. Each subsequent tile can be placed twice as far away and still satisfy the constraint.

Our LDI sampling algorithm proceeds as follows. We position a camera with the field of view and resolution used at run time at "head height" above the origin. For each tile, we determine the resolution of a level of detail by placing the bounding box of the tile's geometry at the appropriate distance from the camera. The camera is tilted to look at the center of the top of the bounding box (approximately the gaze we expect the viewer to have at run time). The screen space projection of the top of the bounding box is then computed, telling us the resolution of the LDI required to ensure a splat size of one. The eight directional LDIs for that level are then created.

Figure 5.7 shows an illustration of rendering an LDI at multiple resolutions. On the left is a tile far in the distance. If we render this LDI close to the camera, the low resolution of the LDI becomes apparent. On the right we show the proper resolution LDI for the close-by position. The results shown in this paper used five levels of detail with LDIs of resolution

253, 127, 63, 32, and 16 pixels square.



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

Figure 5.8: View-dependent sampling. The preferred view direction is: (a) from the south (b) from the west (c) from the north (d) from the east.

Figure 5.8 shows the effect of view-dependent sampling of the tiles. In each column, the top picture is a view from the direction of the sampling rays. The bottom picture is a view rotated ninety degrees counter-clockwise about the center of the tile. So, the bottom row images are rendered from the same direction as the images one column to the right in the top row. By comparing the top row images to the images one down and to the left, you can see the effects of view-dependent sampling. The side of the tile closest to the sampling rays has more depth pixels.

5.4 Real-time Rendering

We have implemented an interactive renderer that combines a software-based LDI renderer with an OpenGL-based polygon renderer. The system is fast enough to allow the user to move around in real time. Rendering proceeds in three stages:

- LDI rendering of the tiles producing an image with alpha plus a z-buffer,
- polygon rendering of the terrain or other standard graphics objects with OpenGL, and finally
- z-compositing of the LDI image over the OpenGL frame buffer.

5.4.1 Initialization

The input to the interactive renderer is: a set of prototiles, a set of MRMVLDIs (one set per prototile), a terrain height field, and the initial position of the viewer. The first thing the system does is compute a tiling that covers the entire height field. Not every tile in the tiling has to be instantiated. For instance, in the examples we show, only tiles that lie on parts of the terrain below a threshold height are instantiated. This is done to prevent the system from trying to put tiles of flowers on tops of mountains. Instead, one could decide to place tiles based on the gradient of the height field or use a hand-painted image mapped over the terrain that defines where tiles can be instantiated [23].

In preparation for rendering, our system finds the object space bounding boxes of all of the tiles in a scene. The footprint and height of the tiles are determined in the modeling phase. The world space placement of a tile's bounding box is determined by the mapping of the tiling onto the height field.

5.4.2 LDI Rendering

For each frame, the first stage of rendering uses an LDI renderer to create an image of all of the visible tiles. To render the visible set of tiles, we step through the tiling in a back-to-front order (as determined by the viewer's position and orientation). Each tile's object space bounding box is tested for inclusion in the view frustum. If this test succeeds, the resolution needed for the tile is computed by finding the distance between the tile and the viewer.

Lastly, the most appropriate directional LDI is chosen. This LDI is then transformed so that it coincides with the position of the tile and warped into the LDI frame buffer.

Since the result of LDI rendering will be combined with a hardware-based rendering of the terrain, we compute a write-only z-buffer as a side effect of LDI rasterization. This is a straightforward extension to the LDI rendering algorithm. We render using a back-to-front order of the tiles, and within each tile, use McMillan's occlusion-compatible warp ordering [57]. The projective z of every pixel already computed for the splatting calculation is splatted into a software z-buffer at the end of the warping function. Since the z-buffer is not used to determine visibility in this phase, it is never read by the LDI renderer.

5.4.3 Terrain Rendering and Z-compositing

Once all of the tiles have been warped, the terrain is rendered using OpenGL. The only acceleration technique we use on the terrain is to render it using triangle strips. The z-buffer produced by the LDI code is then written into the z-buffer produced by the terrain rendering. At every pixel where the LDI z value is closer than the terrain z value, a bit is set in the stencil planes. We then composite the color buffer from the LDI rendering into the hardware frame buffer using the over operator. OpenGL is configured to modify the color buffer only at the pixels where the stencil buffer has been set. This properly z-composites [24] the depth image from the LDI rendering over the depth image created by the OpenGL rendering.

5.4.4 Shear Warping LDIs

In order to mold the LDIs to the terrain, we add an affine shear warp to the standard LDI rendering algorithm. To facilitate this, we render the LDI in two triangular sections. The shear warp is straightforward to compute: we define a frame with the world up direction mapped to the up vector of the frame and each of two sides of a terrain triangle mapped to the other two vectors. The frame defines the matrix used to do the shearing. Lastly, we

add a separating plane to each tile that runs along the diagonal of the LDI. At runtime we use this plane and the viewer's position to determine a back-to-front drawing order of the two halves of the LDI. Shearing is just an approximation to the true deformation that would exactly mimic placing the objects in the tiles on the terrain. If the terrain is very steep or has a sharp feature, the LDIs can be bent in unnatural ways. As our results show, for a gently rolling terrain, the shear warp provides an adequate approximation.

5.5 Results

We demonstrate our system using two sets of tiles, a field of sunflowers and a field of grass. Each of the tile sets is mapped onto a synthetic terrain of gently rolling hills. Figures 5.10 and 5.11 show top-down orthographic and off-to-the side perspective views of each of the tiles. The orthographic views are rendered with shadows turned off in order to make it clear where every object in the tile is being placed. The off-to-the side views were rendered with only the objects assigned to each tile. Objects from adjacent tiles that may overlap the edges are not shown.

Figure 5.9 shows screen shots from our interactive renderer. The top 6 pictures show the sunflower tiles being mapped over the terrain, while the bottom six show the grassy tiles being mapped over the terrain. These are very complex data sets. Each sunflower tile holds on average 20 sunflowers, and each sunflower is comprised of 35,000 triangles. At any point in time, there are approximately 7,000 flowers in the view frustum. This means that our system is reconstructing a view of a database equivalent to one of 245 million triangles. Using the stratified stochastic sampling strategy given in Chapter 4, the 40 LDIs for each tile were sampled using a modified version of the Rayshade [44] raytracer. On a server with a 2.8 GHz Intel Xeon processor and 2 GB of memory, the sampling process took 5 and one-half hours. The multiresolution multi-view LDIs for this scene require 57 MB of storage. Our viewer can render this scene at 3 to 7 frames per second. The system used to do the timings was a PC with a single 1.3 GHz AMD Athlon processor, 512 MB of

memory, and an nVIDIA GeForce2 MX graphics card.

The tiles in the grassy scene are comprised of grass, dandelions and yellow flowers. Each tile in this scene has approximately 62,000 triangles, and the view frustum intersects a portion of the scene equivalent to one with 25 million triangles. Sampling the multiresolution multi-view LDIs for this scene took 46 hours, and the LDIs occupy 200 MB of memory. While there are fewer triangles in the scene, sampling results in 4 times as many depth pixels. There is very little occlusion among the leaves of grass. For this reason, there is a good chance a sampling ray will hit an object on the far side of the tile. View-dependent sampling has a lesser effect on this type of data. We get run time performance of only 2 to 3 frames per second for this scene.

Lastly, Figure 5.1 shows a rendering using one degree digital elevation data distributed by the USGS. While this data set does not demonstrate the shear warping of the tiles, it does answer the question: What would Yosemite valley look like if it was covered in sunflowers?

5.6 Related Work

Kajiya and Kay [40] built a system for rendering fur that used deformed volumes to represent a volumetric texture. Their system is akin to ours in that they deformed the volumes to get local variation. However their system was not interactive, it was rendered using a raytracing algorithm. Neyret [64] extended Kajiya's work to use multiresolution volumes as a technique for anti-aliasing the animation of raytraced volumetric textures. In later work, Meyer and Neyret [60] showed how to use graphics hardware to accelerate volume textures tiled on a surface. The tiles they used had toroidal edge constraints (north matches south and east matches west). So, their tiling is inherently periodic. To add variation to the texture they deform the volume elements according to a height field mapped over an object.

Aperiodic texture mapping of surfaces was the subject of Neyret and Cani [65]. In this paper, they show how to tile a surface aperiodically using triangular tiles. They map two dimensional triangualr textures onto their surfaces. Using the techniques we present,

it may be possible to extrude their triangluar tiles to model solid textures over an object. Stam explored using Robinson's set of 16 Wang tiles to texture map the plane aperiodically [83]. Stam's tilings were small, 6x8, so the structure apparent in the Wang 16 tilings seen in Figure 5.4(d) is not visible in his images. The stochastic tile set introduced in this paper would enhance Stam's work, allowing large non-periodic tilings to be computed easily.

There is a large body of work dealing with the problem of accelerating the display of very complex scenes [76, 81, 3, 5]. These systems are typically geared toward optimizing the use of polygon rendering hardware used in conjunction with image caching. Weber and Penn [88] developed a method for multiresolution modeling of realistic looking trees. These models were employed in a system that generated images of realistic looking terrains, although not in real time. Lastly, Deussen [23], recently presented a system that uses approximate instancing to model expansive natural looking scenes. Approximate instancing is similar to tiling in that it attempts to make non-perioidic imagery by repeating a small set of prototypical objects throughout the scene.

At the other end of the spectrum, a lot of effort has been devoted to realistically modeling plants and terrain. Animatek's World Builder [6] and Bryce from Meta Creations [18] are two commercial software packages that can model and render realistic outdoor scenes. The output from both packages is an image; the user cannot view the scene interactively. Mech [59] has recently shown extremely detailed and natural looking models of trees. These models are so complex, however, that rendering them in real time using polygons is not possible. However, as we saw in Chapter 4, an LDI can be used to render a very complex chestnut created using Mech's technique.

5.7 Discussion

Several aspects of our work merit further discussion. The first is modeling. Figures 5.11 shows top-down views of the grassy tiles. An obvious artifact in these tiles is that that corners are unnaturally sparse. This is inherent in two dimensional tilings. Even with edge

constraints, the tile that is across a corner is unconstrained. Therefore, no object can cross two borders of a tile. Neyret recognized this same problem [65] and solved the problem by making the corner of every tile the same color. We could do the same thing here by manually placing objects in the corner before stochastic object placement.

Second, our run-time system does not render every tile that intersects the view frustum. It only renders those tiles within a radius of 40 tiles from the viewer. There are two reasons for this: our system could not render the scene in real time if we went all the way to the horizon; and, there is no reason to do so. The parallax in tiles that far away is minimal, and would be best rendered as view-dependent texture maps.

Lastly, this work indicates that sparse volume rendering representations are a viable alternative to geometry. Hardware support for a sparse volume rendering primitive would allow scenes like the ones we show here to be rendered at higher frame rates. Even though the data sets we've created are quite large, only a small subset of the multi-view LDIs are needed for any frame. Furthmore, the set of LDIs needed changes smoothly and predictably as the viewer moves through the scene. It is conceivable that streaming the LDIs from system memory to the graphics card as needed would be possible.

5.8 Summary

We have presented a technique that enables real-time rendering of highly complex three dimensional scenes. By combining a result from the field of two dimensional tiling with a view-dependent image-based representation, we have provided a solution to a long sought after problem: adding realistic three dimensional texture to terrains. This work shows that image-based rendering primitives like LDIs can be successfully combined with polygon rendering to solve a problem not easily solved with either technique in isolation.

Figure 5.9: Screenshots of our real-time renderer in action.

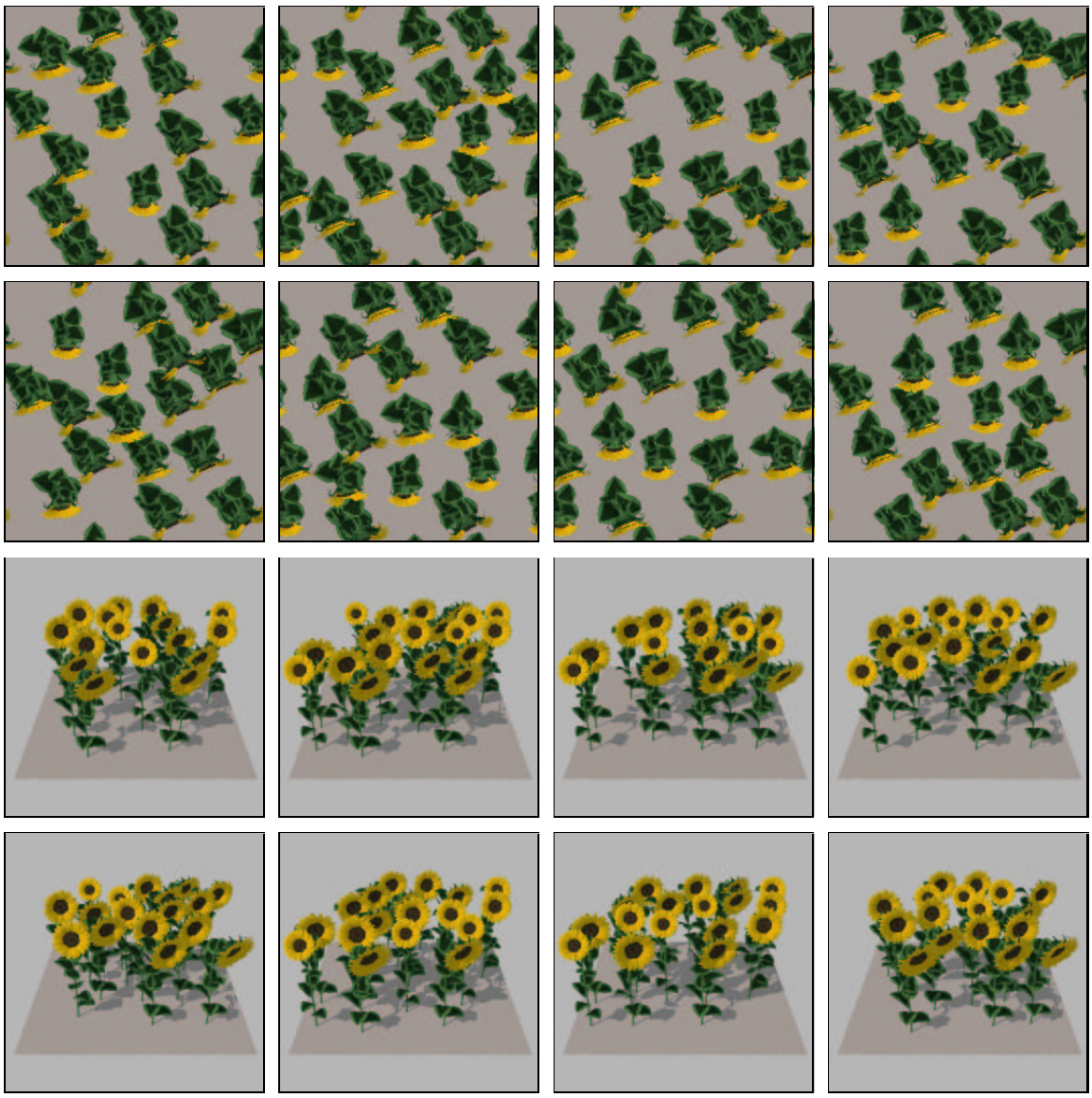


Figure 5.10: The 8 tiles used for the sunflower terrain in top-down and perspective views.

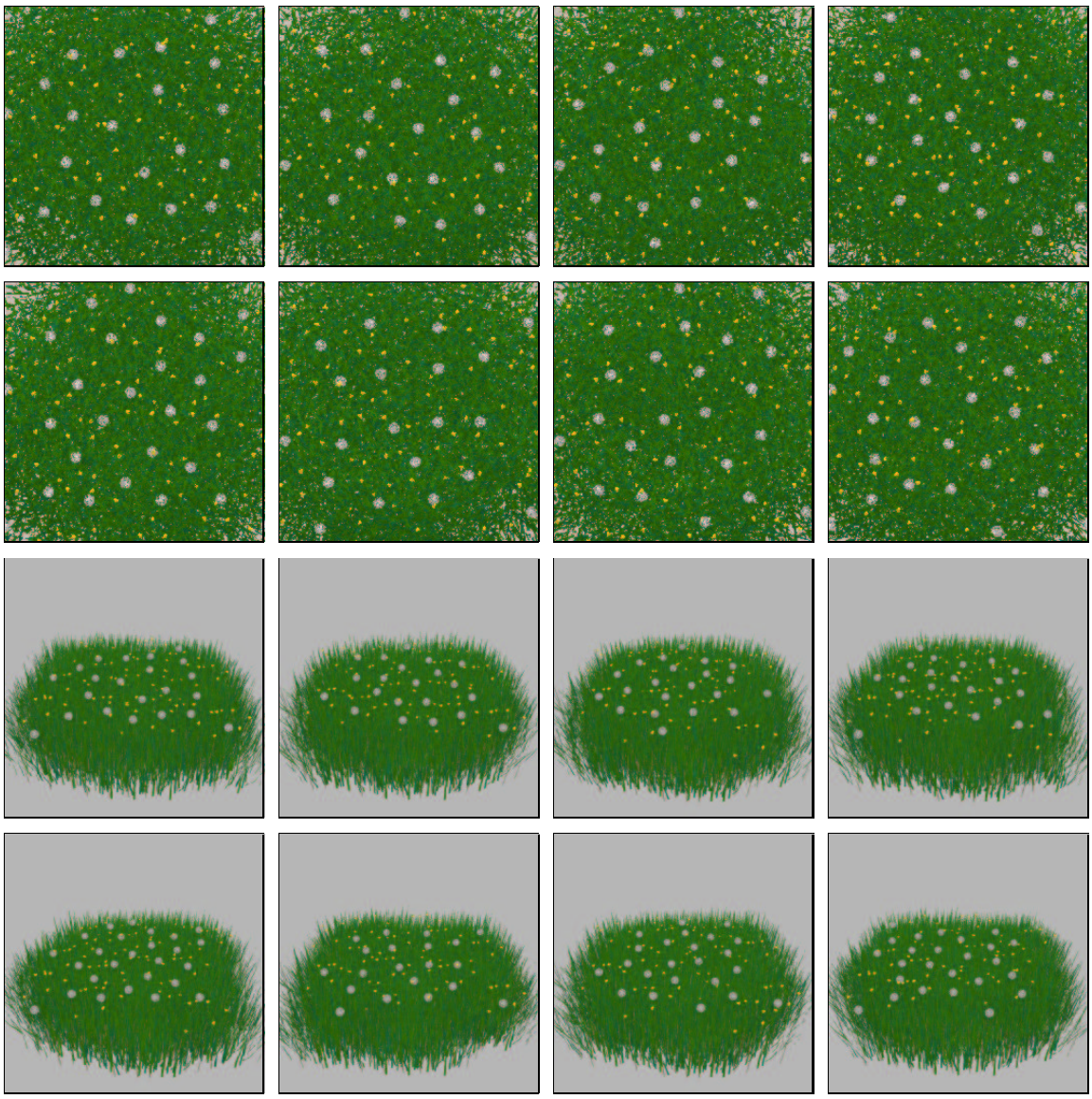


Figure 5.11: The 8 tiles used for the grassy terrain in top-down and perspective views.

Chapter 6

CONCLUSION

This dissertation has presented a novel classification of image-based representations, and described three novel image-based rendering systems: Hierarchical Image Caching, Layered Depth Images, and Tiling Layered Depth Images.

All image-based representations are sampled approximations of the plenoptic function. In order to produce data sets that can be stored in a manageable amount of memory, only a portion of the full plenoptic function can be sampled. Image-based approximations can be classified by four techniques used to choose which part of the plenoptic function is sampled. The first technique is taking a subset of the function to allow for a finite set of samples. The second technique is reducing the dimension of the function by throwing away one or more degrees of freedom. This effectively reduces the size of the representation, but at the cost of also reducing the space of views that can be reconstructed. The third technique is sparsely sampling one of the dimensions. This can be a versatile technique. Instead of throwing away a whole dimension, we throw away most of it but keep an interesting or useful part. Fourth, the amount of data in the plenoptic function can be reduced by finding and eliminating redundancy.

View-dependent representations sample a scene from a particular point of view with the intention that the samples in the recorded view will be projected into a nearby novel view. An entire scene is typically represented by a collection of these view-dependent representations, with those closest to the novel view being used to render the scene. The representations described in this dissertation couple the sampling of shape and shading. This simplification has allowed us to create systems that greatly increase the rendering speed of complex scenes.

6.1 Hierarchical Image Caching

Hierarchical image caching shows that the rendering speed of a very complex scene can be accelerated by replacing the geometry in portions of the scene with images. In this representation sampling shape and shading are coupled, but the image caches are created dynamically. Both shading and shape are updated periodically. While the appearance is static locally, it is dynamic globally. There are many ways to extend this work:

Animation. Although our method is currently applicable only to static scenes, it should be easy to extend it to handle a few small moving objects or animated sprites. A more challenging problem for further research is to allow scenes where many objects are capable of moving and/or deforming their geometry.

Speculative caching. Our algorithm should be extended to cache images not only for nodes already in the view frustum, but also for nodes that should come into view in the next few frames. This extension would help alleviate temporary degradations in rendering performance that occur as a user travels into an area of the scene that is more complex. Speculative caching could be particularly effective if the caching computations are done in parallel by a separate thread.

Geometric LOD modeling. Many of the objects drawn while creating cached images occupy only a small number of pixels in the image. Thus, instead of drawing such objects in full detail, we could draw a coarser model of the same object, using a multi-resolution representation such as the one by Eck et al. [25] or Chamberlain et al. [11]. Using a multi-resolution representation could also accelerate rendering of objects for which no cached images were created.

Persistent caches. As regions of the scene pass out of the view frustum, the image caches for the newly culled nodes are invalidated, and the memory is released. In the case that the

viewer is simply looking around and not changing position, these culled caches are still valid representations of their regions. Suspending invalidation of image caches in this case could potentially save a great deal of computation, in much the same way as the method of Regan and Pose [71].

6.2 Layered Depth Images

Layered depth images extend the concept of depth images to account for disocclusions. By storing multiple samples along each ray of an image a scene can be reconstructed that accurately reproduces the original geometry. There are two essential properties of LDIs. First, McMillan's occlusion-compatible warp ordering for depth images can be extended to LDIs, facilitating an efficient software-based implementation of warping. Second, LDIs store only those samples that are visible from a small range of views near the recording camera. This sparse precomputation of visibility reduces the number of samples of a very complex scene to a manageable level. A scene containing over a billion polygons can be sampled and rendered in software at interactive rates. In contrast to hierarchical image caching, LDIs are static, recording just one sampling of shape and shading. Extensions to this work include:

Dynamic lighting. Dynamic lighting can be added to LDIs by calculating the color of a sample as it is viewed from many directions. At runtime, the gaze direction of the novel camera is used as an index into this table of precomputed colors. To avoid a large increase in the size of an LDI, this function can be sampled sparsely and interpolated at runtime.

Hardware-based rendering. An obvious extension to LDIs is to implement the z-buffer-based rendering algorithm presented in Section 4.3.2. If opaque splats are acceptable, a very fast one-pass LDI renderer can be implemented by warping front-to-back using a hardware-based z-buffer. One of the primary performance constraints in hardware-based renderers is fill rate: the rate at which pixels can be written to the frame-buffer. Rendering front-to-back

conserves fill rate because the closest splats are rendered first, and all splats further from the camera are rejected by the z-buffer test.

Animation. Animation can trivially be added to LDIs by creating an LDI movie. Unfortunately, creating a smooth animation would require a very large number of LDIs for even a short animation. Since the LDI representation is already compressed, a method of compressing motion is needed. Any two adjacent frames of an animation are likely to have many samples in common, but in different locations in the LDI. LDIs could be modified to store a set of unique depth pixels and a set of shared depth pixels. The shared depth pixels store a vector that gives an offset to its position in the next LDI. At runtime, an LDI is created by gathering the unique and shared depth pixels for each ray and sorting them by depth.

6.3 Tiling LDIs

Tiling layered depth images contributes a novel technique for tiling the plane. A set of eight square prototiles can be seamlessly and non-periodically tiled using a stochastic algorithm to place tiles. This two dimensional tiling scheme can be used to render landscapes covered in realistic three dimensional models of plants. Interactive rendering is made possible by leveraging the property of precomputed visibility inherent in LDIs. This shows that LDIs can be used as a basic modeling primitive and not simply a device for rendering a view of an entire scene. LDIs fit within the paradigm of mixed-mode rendering: combining software and hardware rendering. There are many ways to extend this work:

Directional textures. A seamless texture that extends to the horizon can be made by rendering eight directional texture maps in a manner similar to the directional LDI creation. For tiles far from the eye, instead of rendering an LDI, a directional texture is mapped to the top face of the bounding box surrounding a tile. Since these texture maps will only be used for very distant parts of the terrain they can be very low resolution, say 32x32 pixels.

Blending LDIs. Transitions between levels of detail and between different directional LDIs can result in popping. A way to combat this would be to blend between the two LDIs as the transition is made. In addition, blending would enable us to handle directional lighting, albeit at a coarse resolution.

Fairing the edges of the tiling. The transitions from textured to non-textured parts of the terrain (like the tops of hills) is rather abrupt. One solution is to add special tiles whose textures don't cover the entire tile. This, of course, would increase the size of the tile data set.

Adding variations on tiles. Another modeling option is to add more tiles that have the same edge constraints as one of the eight prototiles, but uses a different texture on the interior. This would allow us to put other objects in the tiles.

Automatic creation of dungeons or mazes. An obvious application to games is using tiling to automatically create dungeon-like mazes. A tile would consist of several levels in the dungeon. Edge constraints would require that passages or rooms along the edges match. There is great flexibility in the granularity of the tile in this situation. If the tiles are made large enough, there could be variations on the interiors of the tiles. In other words, having several versions of every tile, all with the same edges but with unique interiors.

Approximation of three dimensional simulations. Any simulation that is expensive to compute could be approximated on a large scale by tiling smaller scale simulations. An example is computing a fluid flow simulation. The simulation could be solved locally inside each tile, taking care every few time steps to make sure the boundaries between the tiles agree. This would certainly not produce a correct fluid flow simulation, but it may produce a plausible one with modest computational resources.

6.4 Final Thoughts

Image-based representations can be effectively used to accelerate rendering of complex, naturalistic scenes. Image caching is well-suited to the current trend in graphics hardware towards rendering partial results of a computation into a texture that is reused later. Since hardware is being optimized for such render-to-texture algorithms, the overhead of caching images will become less significant, increasing the amount of acceleration possible. The average size of a triangle is rapidly approaching one pixel in area. This trend reinforces the idea of using points as a modeling primitive. The view-dependent sampling strategies of LDIs will be an important tool in building point-based representations that are efficient to store and render. Lastly, the richness of textures that can be reproduced with stochastic Wang tilings is compelling. The applications of this technique are broad, and I expect it will become a standard modeling paradigm when creating large-scale highly-detailed scenes.

BIBLIOGRAPHY

[1] E. H. Adelson and J. R. Bergen. The plenoptic function and the elements of early vision. In M. Landy and J. A. Movshon, editors, *Computational Models of Visual Processing*, chapter 1. MIT Press, Cambridge, MA, 1991.

[2] John M. Airey, John H. Rohlf, and Jr. Frederick P. Brooks. Towards image realism with interactive update rates in complex virtual building environments. *1990 Symposium on Interactive 3D Graphics*, 24(2):41–50, March 1990. ISBN 0-89791-351-5.

[3] Daniel Aliaga, Jonathan Cohen, Andrew Wilson, Hansong Zhang, Carl Erikson, Kenneth Hoff, T. Hudson, Wolfgang Strzlinger, Eric Baker, Rui Bastos, Mary Whitton, Frederick Brooks, and Dinesh Manocha. Mmr: An interactive massive model rendering system using geometric and image-based acceleration. *1999 ACM Symposium on Interactive 3D Graphics*, pages 199–206, April 1999. ISBN 1-58113-082-1.

[4] Daniel G. Aliaga. Visualization of complex models using dynamic texture-based simplification. *IEEE Visualization '96*, pages 101–106, October 1996. ISBN 0-89791-864-9.

[5] Daniel G. Aliaga and Anselmo Lastra. Automatic image placement to provide a guaranteed frame rate. *Proceedings of SIGGRAPH 99*, pages 307–316, August 1999. ISBN 0-20148-560-5. Held in Los Angeles, California.

[6] Animatek. *World Builder Web Site*. http://www.animatek.com, 1999.

[7] Thaddeus Beier and Shawn Neely. Feature-based image metamorphosis. *Computer*

Graphics (Proceedings of SIGGRAPH 92), 26(2):35–42, July 1992. ISBN 0-201-51585-7. Held in Chicago, Illinois.

[8] R. Berger. The undecidablility of the domino problem. In *Memoirs Amer. Math. Soc.*, page 72, 1966.

[9] Robert C. Bolles, Harlyn H. Baker, and David H. Marimont. Epipolar-plane image analysis: An approach to determining structure from motion. *International Journal of Computer Vision*, 1(7):7–55, 1987.

[10] Ed Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, 1974.

[11] Bradford Chamberlain, Tony DeRose, Dani Lischinski, David Salesin, and John Snyder. Fast rendering of complex environments using a spatial hierarchy. In *Proceedings of Graphics Interface '96*, May 1996.

[12] Shenchang Eric Chen. Quicktime vr - an image-based approach to virtual environment navigation. *Proceedings of SIGGRAPH 95*, pages 29–38, August 1995. ISBN 0-201-84776-0. Held in Los Angeles, California.

[13] Shenchang Eric Chen and Lance Williams. View interpolation for image synthesis. *Proceedings of SIGGRAPH 93*, pages 279–288, August 1993. ISBN 0-201-58889-7. Held in Anaheim, California.

[14] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, October 1976.

[15] Michael F. Cohen and Donald P. Greenberg. The hemi-cube: A radiosity solution for complex environments. *Computer Graphics (Proceedings of SIGGRAPH 85)*, 19(3):31–40, August 1985. Held in San Francisco, California.

[16] Michael F. Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. Wang tiles for image and texture generation. *ACM Transactions on Graphics*, 22(3):287–294, July 2003.

[17] Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press, San Diego, 1993.

[18] Meta Creations. *Bryce Web Site*. http://www.metacreations.com/products/bryce4, 1999.

[19] William Dally, Leonard McMillan, Gary Bishop, and Henry Fuchs. The delta tree: An object centered approach to image based rendering. AI technical Memo 1604, MIT, 1996.

[20] Lucia Darsa, Bruno Costa Silva, and Amitabh Varshney. Navigating static environments using image-space simplification and morphing. *1997 Symposium on Interactive 3D Graphics*, pages 25–34, April 1997. ISBN 0-89791-884-3.

[21] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 11–20. ACM SIGGRAPH, Addison Wesley, August 1996.

[22] Paul E. Debevec, Yizhou Yu, and George D. Borshukov. Efficient view-dependent image-based rendering with projective texture-mapping. *Eurographics Rendering Workshop 1998*, pages 105–116, June 1998. ISBN 3-211-83213-0. Held in Vienna, Austria.

[23] Oliver Deussen, Patrick Hanrahan, Bernd Lintermann, Radomír Mech, Matt Pharr, and Przemyslaw Prusinkiewicz. Realistic modeling and rendering of plant ecosys-

tems. *Proceedings of SIGGRAPH 98*, pages 275–286, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[24] Tom Duff. Compositing 3-d rendered images. *Computer Graphics (Proceedings of SIGGRAPH 85)*, 19(3):41–44, July 1985. Held in San Francisco, California.

[25] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis for arbitrary meshes. In *Computer Graphics* Proceedings, Annual Conference Series, pages 173–182, August 1995.

[26] O. Faugeras. *Three-dimensional computer vision: A geometric viewpoint*. MIT Press, Cambridge, Massachusetts, 1993.

[27] Henry Fuchs, Z. M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. *Computer Graphics (Proceedings of SIGGRAPH 80)*, 14(3):124–133, July 1980. Held in Seattle, Washington.

[28] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Computer Graphics* Proceedings, Annual Conference Series, pages 247–254, August 1993.

[29] B. Garlick, D. Baum, and J. Winget. Interactive viewing of large geometric databases using multiprocessor graphics workstations. SIGGRAPH ‘90 Course Nores: Parallel Algorithms and Architectures for 3D Image Generation, 1990.

[30] Andrew Glassner. Cubism and cameras: Free-form optics for computer graphics. Technical Report MSR-TR-2000-5, Microsoft Corporation, 2000.

[31] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modelling the interaction of light between diffuse surfaces. *Computer Graphics (Pro-*

ceedings of SIGGRAPH 84), 18(3):213–222, July 1984. Held in Minneapolis, Minnesota.

[32] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 43–54. ACM SIGGRAPH, Addison Wesley, August 1996.

[33] N. Greene and P. Heckbert. Creating raster Omnimax images from multiple perspective views using the elliptical weighted average filter. *IEEE Computer Graphics and Applications*, 6(6):21–27, June 1986.

[34] Ned Greene and Michael Kass. Approximating visibility with environment maps. Technical Report 41, Apple Computer, Inc., 1994.

[35] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *Computer Graphics* Proceedings, Annual Conference Series, pages 231–238, August 1993.

[36] Branko Grunbaum and G. C. Shephard. *Tilings and Patterns*. W. H. Freeman and Company, 1987.

[37] Michael W. Halle. Holographic stereograms as discrete imaging systems. In *Practical Holography VIII*, volume 2176, pages 73–84, Bellingham, WA, 1994. SPIE - The International Society for Optical Engineering.

[38] Paul S. Heckbert and Henry P. Moreton. *Interpolation for Polygon Texture Mapping and Shading*, pages 101–111. Springer-Verlag, 1991.

[39] Karel Culik II. An aperiodic set of 13 wang tiles. *Discrete Mathematics*, 160:245–251, 1996.

[40] James T. Kajiya and Timothy L. Kay. Rendering fur with three dimensional textures. *Computer Graphics (Proceedings of SIGGRAPH 89)*, 23(3):271–280, July 1989. Held in Boston, Massachusetts.

[41] Jarkko Kari. An small aperiodic set of wang tiles. *Discrete Mathematics*, 160:259–264, 1996.

[42] Akihiro Katayama, Koichiro Tanaka, Takahiro Oshino, and Hideyuki Tamura. A viewpoint dependent stereoscopic display using interpolation of multi-viewpoint images. *Stereoscopic Displays and Virtual Reality Systems II(SPIE)*, 2409:11–20, 1995.

[43] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *Computer Graphics (Proceedings of SIGGRAPH 86)*, 20(4):269–278, August 1986. Held in Dallas, Texas.

[44] Craig E. Kolb. *Rayshade User's Guide and Reference Manual*. http://graphics.stanford.edu/ cek/rayshade, 1992.

[45] Anthony G. LaMarca. *Caches and Algorithms*. PhD thesis, University of Washington, 1996.

[46] S. Laveau and O. D. Faugeras. 3-d scene representation as a collection of images. In *Twelfth International Conference on Pattern Recognition (ICPR'94)*, volume A, pages 689–691, Jerusalem, Israel, October 1994. IEEE Computer Society Press.

[47] Jed Lengyel and John Snyder. Rendering with coherent layers. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 233–242. ACM SIGGRAPH, Addison Wesley, August 1997.

[48] Marc Levoy and Pat Hanrahan. Light field rendering. In Holly Rushmeier, editor,

SIGGRAPH 96 Conference Proceedings, Annual Conference Series, pages 31–42. ACM SIGGRAPH, Addison Wesley, August 1996.

[49] Mark Levoy and Turner Whitted. The use of points as a display primitive. Technical Report 85-022, University of North Carolina, 1985.

[50] A. Lippman. Movie-maps: an application of the optical videodisc to computer graphics. *Computer Graphics*, 14(3):32–42, July 1980.

[51] Dani Lischinski and Ari Rappoport. Image-based rendering for non-diffuse synthetic scenes. *Eurographics Rendering Workshop 1998*, pages 301–314, June 1998. ISBN 3-211-83213-0. Held in Vienna, Austria.

[52] Daivid Luebke and Chris Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *1995 Symposium on Interactive 3D Graphics*, pages 105–106, April 1995.

[53] Paulo W. C. Maciel and Peter Shirley. Visual navigation of large environments using textured clusters. In *1995 Symposium on Interactive 3D Graphics*, pages 95–102, April 1995.

[54] William R. Mark, Leonard McMillan, and Gary Bishop. Post-rendering 3D warping. In *Proc. 1997 Symposium on Interactive 3D Graphics*, pages 7–16, 1997.

[55] William R. Mark, Leonard McMillan, and Gary Bishop. Post-rendering 3d warping. *1997 Symposium on Interactive 3D Graphics*, pages 7–16, April 1997. ISBN 0-89791-884-3.

[56] Nelson Max. Hierarchical rendering of trees from precomputed multi-layer Z-buffers. In Xavier Pueyo and Peter Schröder, editors, *Eurographics Rendering Workshop 1996*, pages 165–174, New York City, NY, June 1996. Eurographics, Springer Wein.

[57] Leonard McMillan. Computing visibility without depth. Technical Report 95-047, University of North Carolina, 1995.

[58] Leonard McMillan and Gary Bishop. Plenoptic modeling: An image-based rendering system. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 39–46. ACM SIGGRAPH, Addison Wesley, August 1995.

[59] Radomír Mech and Przemyslaw Prusinkiewicz. Visual models of plants interacting with their environment. *Proceedings of SIGGRAPH 96*, pages 397–410, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[60] Alexandre Meyer and Fabrice Neyret. Interactive volumetric textures. *Eurographics Rendering Workshop 1998*, pages 157–168, June 1998. ISBN 3-211-83213-0. Held in Vienna, Austria.

[61] Gavin S. P. Miller, Steven Rubin, and Dulce Ponceleon. Lazy decompression of surface light fields for precomputed global illumination. *Eurographics Rendering Workshop 1998*, pages 281–292, June 1998. ISBN 3-211-83213-0. Held in Vienna, Austria.

[62] Don P. Mitchell. *personal communication*. 1997.

[63] Jackie Neider, Tom Davis, and Mason Woo. *OpengGL Programming Guide*. Addison Wesley, 1993.

[64] Fabrice Neyret. A general and multiscale model for volumetric textures. *Graphics Interface '95*, pages 83–91, May 1995. ISBN 0-9695338-4-5.

[65] Fabrice Neyret and Marie-Paule Cani. Pattern-based texturing revisited. *Proceedings of SIGGRAPH 99*, pages 235–242, August 1999. ISBN 0-20148-560-5. Held in Los Angeles, California.

[66] Tomoyuki Nishita and Eihachiro Nakamae. Continuous tone representation of three-dimensional objects taking account of shadows and interreflection. *Computer Graphics (Proceedings of SIGGRAPH 85)*, 19(3):23–30, July 1985. Held in San Francisco, California.

[67] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. *Proceedings of SIGGRAPH 97*, pages 101–108, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[68] Thomas Porter and Tom Duff. Compositing digital images. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, volume 18, pages 253–259, July 1984.

[69] Kari Pulli, Michael Cohen, Tom Duchamp, Hugues Hoppe, Linda Shapiro, and Werner Stuetzle. View-based rendering: Visualizing real objects from scanned range and color data. *Eurographics Rendering Workshop 1997*, pages 23–34, June 1997. ISBN 3-211-83001-4. Held in St. Etienne, France.

[70] Paul Rademacher and Gary Bishop. Multiple-center-of-projection images. *Proceedings of SIGGRAPH 98*, pages 199–206, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[71] Matthew Regan and Ronald Pose. Priority rendering with a virtual reality address recalculation pipeline. *Proceedings of SIGGRAPH 94*, pages 155–162, July 1994. ISBN 0-89791-667-0. Held in Orlando, Florida.

[72] R. M. Robinson. Undecidable tiling problems in the hyperbolic plane. In *Inventiones Math.*, pages 259–264, 1978.

[73] Jarek Rossignac and Paul Borrel. Multi-resolution 3D approximations for rendering complex scenes. Research Report RC 17697 (#77951), IBM, Yorktown Heights, New York 10598, 1992. Also appeared in the *IFIP TC 5.WG 5.10*.

[74] Graham Saxby. *Practical Holography*. Prentice Hall, second edition, 1994.

[75] Gernot Schaufler. Exploiting frame to frame coherence in a virtual reality system. In *Proceedings of VRAIS '96*, pages 95–102, April 1996.

[76] Gernot Schaufler and Wolfgang Stürzlinger. A three dimensional image cache for virtual reality. *Computer Graphics Forum*, 15(3):227–236, August 1996. ISSN 1067-7055.

[77] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul E. Haeberli. Fast shadows and lighting effects using texture mapping. *Computer Graphics (Proceedings of SIGGRAPH 92)*, 26(2):249–252, July 1992. ISBN 0-201-51585-7. Held in Chicago, Illinois.

[78] Steven M. seitz and Charles R. Dyer. Photorealistic scene reconstruction by voxel coloring. In *Proc. Computer Vision and Pattern Recognition Conf.*, pages 1067–1073, 1997.

[79] Steven M. Seitz and Charles R. Dyer. View morphing: Synthesizing 3d metamorphoses using image transforms. *Proceedings of SIGGRAPH 96*, pages 21–30, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[80] Jonathan Shade, Steven J. Gortler, Li wei He, and Richard Szeliski. Layered depth images. *Proceedings of SIGGRAPH 98*, pages 231–242, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[81] Jonathan Shade, Dani Lischinski, David Salesin, Tony DeRose, and John Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. *Proceedings of SIGGRAPH 96*, pages 75–82, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

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

[83] Jos Stam. Aperiodic texture mapping. Technical Report R046, European Research Consortium for Informatics and Mathematics (ERCIM), 1997.

[84] Seth J. Teller. *Visibility Computations in Densely Occluded Polyhedral Environments*. PhD thesis, Computer Science Division (EECS), UC Berkeley, Berkeley, California 94720, October 1992. Available as Report No. UCB/CSD-92-708.

[85] Jay Torborg and Jim Kajiya. Talisman: Commodity Real-time 3D graphics for the PC. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 353–364. ACM SIGGRAPH, Addison Wesley, August 1996.

[86] Hao Wang. Proving theorems by pattern recognition. *Bell system Tech. J.*, 40:1–42, 1961.

[87] Hao Wang. Games, logic and computers. *Scientific American*, pages 98–106, November 1965.

[88] Jason Weber and Joseph Penn. Creation and rendering of realistic trees. *Proceedings of SIGGRAPH 95*, pages 119–128, August 1995. ISBN 0-201-84776-0. Held in Los Angeles, California.

[89] Lee Westover. Footprint evaluation for volume rendering. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 367–376, August 1990.

[90] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980.

[91] Daniel N. Wood, Daniel I. Azuma, Ken Aldinger, Brian Curless, Tom Duchamp, David H. Salesin, and Werner Stuetzle. Surface light fields for 3d photography. In Kurt Akeley, editor, *to appear in SIGGRAPH 2000 Conference Proceedings*, Annual Conference Series. ACM SIGGRAPH, Addison Wesley, August 2000.

[92] Daniel N. Wood, Adam Finkelstein, John F. Hughes, Craig E. Thayer, and David H. Salesin. Multiperspective panoramas for cel animation. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 243–250. ACM SIGGRAPH, Addison Wesley, August 1997.

VITA

Jonathan Shade was born in 1970 in Indianapolis. Grew up in Memphis. Spent a moment in Sacramento on the way to San Diego. Met Erin. Whiled away a summer colliding particles under Sand Hill. Earned a Bachelor of Science in Computer Engineering at UCSD. Married Erin. Moved to Seattle. Earned a Master of Science in Computer Science and a Doctor of Philosophy in Computer Science at the University of Washington. And now makes pretty pictures for a living.