LDI TREE: A SAMPLING RATE PRESERVING AND HIERARCHICAL DATA REPRESENTATION FOR IMAGE-BASED RENDERING

by

Chun-Fa Chang

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements of the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

2001

Approved by:

_____

Advisor: Gary Bishop

_____

Reader: Anselmo Lastra

_____

Reader: Lars S. Nyland

_____

Nick England

_____

Russell M. Taylor II

ABSTRACT

Chun-Fa Chang

LDI TREE: A SAMPLING RATE PRESERVING AND HIERARCHICAL DATA REPRESENTATION FOR IMAGE-BASED RENDERING

(Under the direction of Dr. Gary Bishop)

Multiple reference images are required to eliminate disocclusion artifacts in image-based rendering based on McMillan's 3D image warping. Simply warping each individual reference image causes the rendering time to grow linearly with the number of reference images. This calls for a new data representation for merging multiple reference images.

In this dissertation I present a hierarchical data representation, the LDI Tree, for merging reference images. It is distinguished from previous works by identifying the sampling rate issue and preserving the sampling rates of the reference images by adaptively selecting a suitable level in the hierarchy for each pixel. During rendering, the traversal of the LDI Tree is limited to the levels that are comparable to the sampling rate of the output image. This allows the rendering time to be driven by the requirements of output images instead of the number of input images. I also present a progressive refinement feature and a hole-filling algorithm implemented by pre-filtering the LDI Tree.

I show that the amount of memory required has the same order of growth as the 2D reference images in the worst case. I also show that the rendering time depends mostly on the quality of the output image, while the number of reference images has a relatively small impact.

To

my wife,

my parents,

and

my grandparents.

ACKNOWLEDGEMENTS

Special thanks to my advisor, Gary Bishop, for being a mentor, a friend and a role model, for his guidance and encouragement, and for many discussions during which he patiently followed every detail of my work while making sure that I did not lose sight of the big picture.

Thanks to the other members of my dissertation committee: Anselmo Lastra, Lars Nyland, Nick England and Russell Taylor for their comments and encouragement.

I would like to also thank Turner Whitted for spending precious time and sharing his research philosophy with me during my early exploration of image-based rendering.

I would like to acknowledge the following people who provided help during my project: Nathan O'Brien for creating the Rayshade model of Il Redentore, David McAllister, Voicu Popescu, Chris McCue, Lars and Anselmo for providing the reading room model, and the entire UNC ImageFlow project team for many inspiring discussions.

Thanks to Norman Jouppi for the wonderful internship at Western Research Laboratory during the summers of 1996 and 1997.

Thanks to Mu-Yu Yang for helping me settle when I first arrived in the Triangle area, and Ta-Ming Chen, Heng Chu, Gentaro Hirota, Wei-Chao Chen, William Jiang, and many other fellow graduate students for many joyful years since then.

Thanks to many friends in King's Park International Church and Churches Serving Internationals, especially John and Yunhee Gray, Todd and Rachel Schwartz, and Wayne and Sharon Mitchell, for their support and prayers during several inevitable difficult times in life.

Thanks to Yao-Wen Chang for persuading me to pursue a doctoral degree and for his assistance during my job search.

Finally, my deepest appreciation to my wife, Chia-Lin Yang, my parents and my grandparents for their everlasting and unconditional love.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

2D — two-dimensional

3D — three-dimensional

LDI — layered depth image

MB — megabyte; equivalent to 2^{20} bytes

MCOP — multiple-center-of-projection

CHAPTER 1 – INTRODUCTION

1.1 Image-Based Rendering

In computer graphics, photorealistic rendering refers to producing images that are indistinguishable from real photographs. Traditionally it has been accomplished by meticulously modeling the geometry and material properties of the objects to be rendered and correctly computing the subtle interaction of lighting between those objects. Most of the scenes encountered in our daily life contain such complexity that models containing tens or hundreds of millions of polygons are necessary to convey all the details to make a photorealistic rendering. With the recent advances in processor performance and in specialized graphics hardware subsystems, rendering those complex models might no longer be a formidable task. However, building such complex models is still time consuming.

Recently, image-based rendering has gained popularity and provides an alternative to the traditional polygon-based modeling and rendering processes. In image-based rendering, no three-dimensional (3D) model is explicitly provided as the input data. Instead, images are acquired from a limited number of positions in the scene. When the user moves beyond those positions, the image-based rendering program attempts to reconstruct the scene and render the output images using the information that is available in the input images.

1.2 3D Image Warping

There are two main branches of image-based rendering, depending on whether the input images contain depth information. In this work, I only consider input images that contain depth information. The rendering algorithms described here are derived from the 3D Image Warping algorithm of McMillan and Bishop [McMillan97]. In contrast, some image-based rendering methods such as the Light Field Rendering [Levoy96] and the

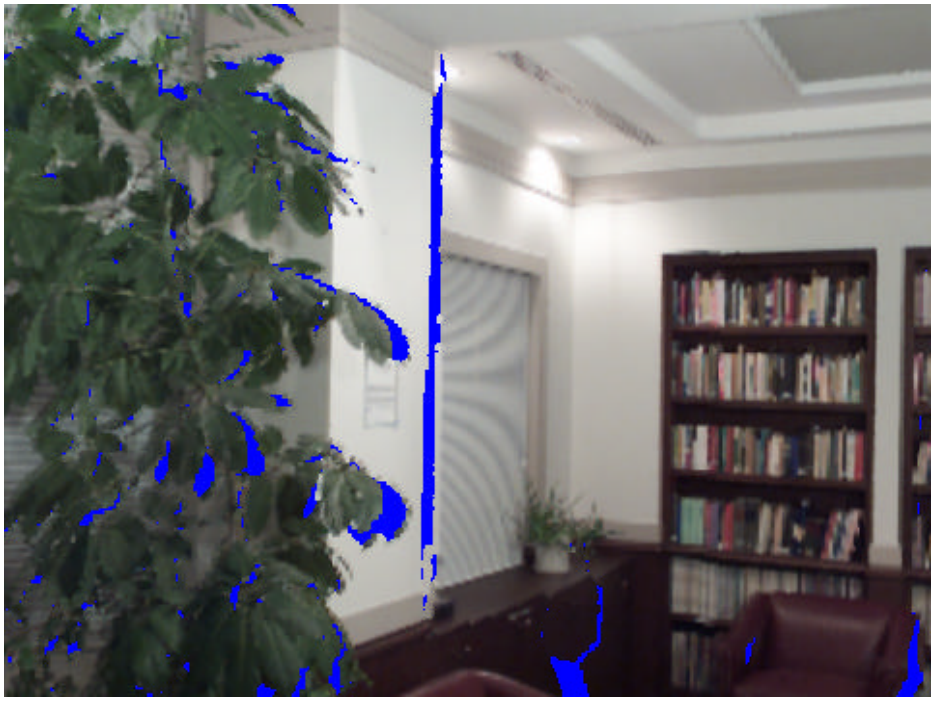


Figure 1-1: Disocclusion artifacts of 3D Image Warping (shown in the blue background color.

Lumigraph [Gortler96] use input images that do not require depth information, but those methods need significantly larger amount of input data.

Given an image and the camera parameters of that image, 3D image warping can generate images for novel viewpoints by moving the pixels of the input image to their locations on the output images.

McMillan's 3D Image warping algorithm uses regular single-layered depth images (which are called *source images* or *reference images*) as the initial input. One of the major problems of the 3D image warping is the disocclusion artifact that is caused by the areas that are occluded in the original reference images but visible in the current view. Those artifacts appear as holes or gaps in the output image. Figure 1-1 shows an example. In Mark's Post-Rendering Warping [Mark97], the techniques of splatting and meshing are proposed to deal with the disocclusion artifacts. Both splatting and meshing are adequate for post-rendering warping where the current view does not deviate much from the view of the reference image. However, the fundamental problem of the

disocclusion artifact is that the information of the previously occluded area is missing in the reference image. By using multiple reference images taken from different viewpoints, the disocclusion artifacts can be reduced because an area that is not visible from one view may be visible from another. When multiple source images are available, we expect the disocclusion artifacts that occur while warping one reference image to be eliminated by one of the other reference images.

An advantage that image-based rendering has over traditional polygon-based rendering is that the rendering cost does not grow with the scene complexity. While using multiple reference images can attenuate the disocclusion artifacts, it also increases the rendering cost. If too many reference images are used, then we lose an advantage of image-based rendering.

The problem comes from the redundant information among the reference images. While the additional reference images provide information on previously occluded surfaces, they also introduce redundancy to those surfaces that are already visible. If we render the new views by warping every reference image, then most of the rendering time is wasted in processing the redundant information.

Another problem with multiple reference images is the difficulty in arbitrating between the redundant information. When a surface is visible in multiple reference images, it may look like multiple overlapping surfaces if its redundancy is undetected. This is a typical problem for graphics hardware that use Z-buffer for hidden-surface removal because small numerical errors could cause different ordering of those overlapping surfaces, which can lead to noticeable artifacts.

Therefore it is desirable to merge multiple reference images into an intermediate data structure for rendering. By merging the depth images into a new representation, we are able to eliminate the redundancy and render the output images more efficiently. However, merging multiple reference images and eliminating the redundant information remains a challenging problem.

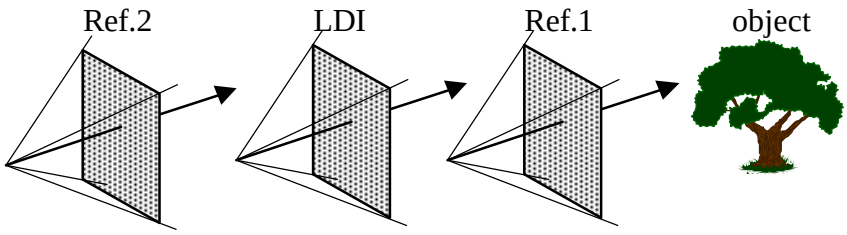


Figure 1-2: The LDI does not preserve the sampling rates of the reference images.

1.3 Layered Depth Image

Recently, the Layered Depth Image (LDI) was proposed by Shade et al [Shade98] to merge many reference images under a single center of projection. It tackles the disocclusion problem by keeping multiple depth pixels per pixel location, while still maintaining the simplicity of warping a single reference image. Its limitation is that the fixed resolution of the LDI may not provide an adequate sampling rate for every reference image. Figure 1-2 shows two examples of such situations. Assuming the two reference images have the same resolution as the LDI, the object covers more pixels in reference image 1 than it does in the LDI. Therefore the LDI has a lower sampling rate for the object than reference image 1. Similar analysis shows the LDI has a higher sampling rate than reference image 2. If we combine both reference images into the LDI and render the object from the center of projection of reference image 1, the insufficient sampling rate of the LDI will cause the object to look blurrier than it looks in reference image 1. When we render the object from the center of projection of reference image 2, the excessive sampling rate of the LDI might not hurt the quality of the output. However, processing more pixels than necessary slows down the rendering.

1.4 Hierarchical Representation

Let us look at the sampling rate issue in more detail. During rendering, we would like to efficiently extract samples that are adequate for the desired viewpoints from this intermediate data structure that we created from the reference images. This sample extraction problem is in fact a view dependent problem. That is because every output image poses a different sampling rate requirement for all the visible surfaces in the

rendered scene. Mismatch in the sampling rate between the extracted samples and the output pixels will result in either a poorly rendered output image or wasted processing resources. This suggests a hierarchical (or multiresolution) data structure.

Using multiresolution representations to control the level of detail of models in order to match the requirements of output images is a common practice in rendering of polygonal models [Erikson00, Lee98]. In fact, much effort has been devoted to polygonal simplification [Cohen96, Luebke97] to automatically create object models in various polygon counts. In contrast, building a multiresolution representation for image-based rendering is much simpler because the objects are already represented as discrete samples, which are amenable to the operations of filtering, resampling, and reconstruction.

In this dissertation, I propose to use 3D hierarchical representations for merging multiple depth images. My approach has the following two properties that distinguish it from the other methods:

1. It preserves the sampling rates of each depth image (unlike the LDI).
2. It sorts and merges the samples from the depth images in 3D, while allocating resources (in terms of storage space and rendering time) comparable to the original 2D depth images.

This leads to the following thesis statement.

1.5 Thesis Statement.

By considering depth images as samples of surfaces in a 3D scene and carefully preserving their sampling rates using a hierarchical representation, we can synthesize new views in time that is dependent mostly on the output quality, instead of the number of reference images.

Therefore as many images as necessary may be used to solve the disocclusion problem without increasing the rendering cost unduly.

1.6 Framework

In this dissertation, I demonstrate the benefits of the hierarchical representations for combining multiple depth images by implementing a hierarchical representation, which is called the LDI Tree.

Currently, acquiring source images with precise depth information from the real world is still a difficult task. Because this thesis focuses on the rendering aspects, not the modeling and acquisition aspects of image-based rendering and 3D image warping, I demonstrate my system using source images from synthetic scenes. I also show preliminary results of using images acquired from the real world and suggest ways to deal with the associated artifacts due to the imperfection of the data set.

1.7 Outline

The rest of this dissertation is organized as follows:

- Chapter 2 describes the background and related work. It includes various image-based rendering techniques and 3D Image Warping on which my work is based.
- Chapter 3 describes, in detail, hierarchical representations and the LDI Tree, in particular how they are built from reference images and how to render output images from them.
- Chapter 4 analyzes the memory usage and the rendering time of the LDI Tree.
- Chapter 5 concludes the dissertation and discusses the issues of occlusion culling, hardware support, and image acquisition, as well as possible future work.

CHAPTER 2 – BACKGROUND AND RELATED WORK

2.1 Image-Based Rendering

Image-based rendering is a broad term for all rendering techniques that use images instead of or in addition to geometric models as the input data. The theoretical background of the image-based rendering is the Plenoptic Function introduced by Adelson [Adelson91]. The Plenoptic Function is a 7-dimensional function:

$$P(q, f, l, t, V_x, V_y, V_z)$$

where

- (q, f) represents the viewing angle.
- (V_x, V_y, V_z) represents the location in the three-dimensional space.
- l represents the wavelength of light.
- t represents the time.

For static scenes (i.e., t is constant), the Plenoptic Function may be rewritten as a 5-dimensional function[1]:

$$color = P(q, f, V_x, V_y, V_z)$$

2.2 Light Field and Lumigraph

It is almost impossible to represent the Plenoptic Function in analytic form. Therefore we rely on discrete representations of the function by sampling and reconstruction of the continuous function. One such representation is the Light Field

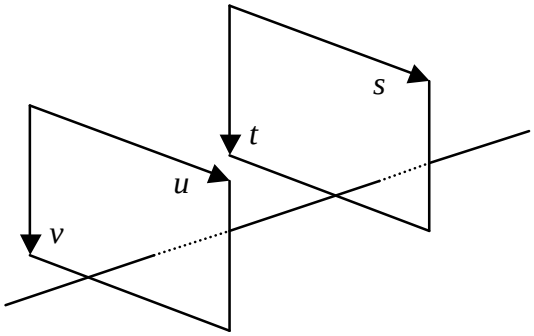


Figure 2-1: The two-plane parameterization of line space.

representation proposed by Levoy and Hanrahan [Levoy96]. The Lumigraph is a similar representation that was independently discovered by Gortler et al [Gortler96] around the same time.

The Light Field is a 4D function that is defined for a line in the object space. A line may be parameterized in many ways, one of which is to use the two-plane parameterization, which is depicted in Figure 2-1. If the *u*-*v* plane and the *s*-*t* plane are infinite, then any line[2] in the object space may be parameterized by (*u, v, s, t*) according to where the line intersects the two planes. Then the Light Field function can be written as:

$$color = L(u, v, s, t)$$

The Light Field has one fewer dimension than the Plenoptic Function by recognizing the fact that the Plenoptic Function along a ray remains constant in an open space.

An interesting relationship between regular 2D images and the Light Field is the following. When a camera is placed on the *u*-*v* plane, an image taken by that camera contains samples of the Light Field function of the scene with the same (*u, v*) parameters

---

[1] Note that the l parameter is now embedded in the color. Depending on the color representation, a color channel may be calculated as the integral (along l) of the plenoptic function multiplied with a weighting function $C(l)$ specific to that color channel.

[2] Except lines that are parallel to the *u*-*v* plane and the *s*-*t* plane.

but different (s, t) at each pixel. This means that a Light Field function can be built from 2D images without any knowledge of the depth of each pixel or other 3D geometric information of the scene. However the Light Field must be densely sampled to avoid blurriness in the reconstructed output images. In Levoy and Hanrahan's system [Levoy96] the u-v plane contains at least 16×16 sample points and the s-t plane contains at least 128×128 sample points, which results in raw data size of 50 to 1608 MB without compression.

2.3 Plenoptic Modeling and 3D Image Warping

McMillan and Bishop introduced Plenoptic Modeling [McMillan95b] as a method to sample the Plenoptic Function with input images at some given viewpoints and to generate novel views by reconstructing the Plenoptic Function at new viewpoints. The reconstruction of the Plenoptic Function is made possible by obtaining the disparity (a quantity inversely proportional to depth or range) associated with each pixel of the input images. The details about how the disparity values are obtained are in their paper [McMillan95b]. With the per-pixel disparity values, we may apply the 3D Image Warping to move each pixel of an input image to its new pixel location on a new image at a different viewpoint. The following warping equation shows how we can calculate the coordinates on the new image (x_2, y_2) for a pixel on the input image that has the coordinates (x_1, y_1) and the disparity d_1.

$$(x_2, y_2) = \left(\frac{k_1 x_1 + k_2 y_1 + k_3 + k_4 d_1}{k_9 x_1 + k_{10} y_1 + k_{11} + k_{12} d_1}, \frac{k_5 x_1 + k_6 y_1 + k_7 + k_8 d_1}{k_9 x_1 + k_{10} y_1 + k_{11} + k_{12} d_1}\right)$$

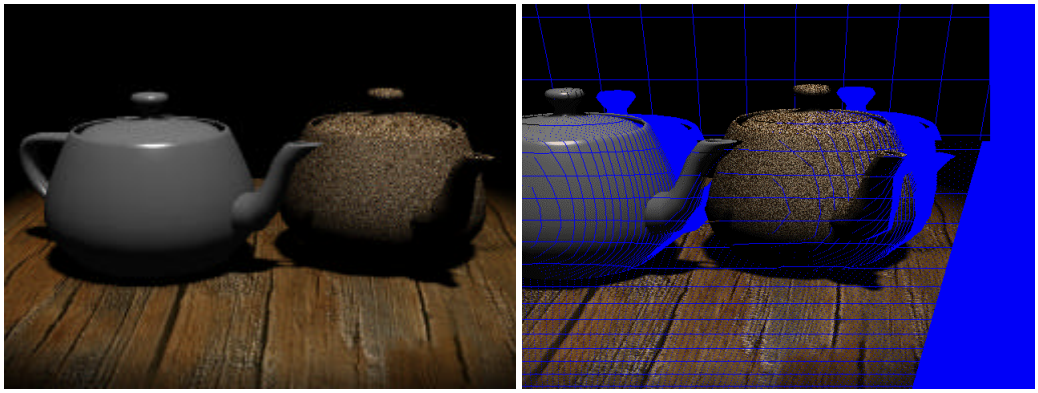
The coefficients k_1 through k_{12} are calculated from the viewing parameters of the input image and the output image. They remain constant until the view changes in the subsequent frame. Appendix A shows the detailed derivation of the warping equation.

McMillan also introduced the concept of *occlusion compatible order* for 3D Image Warping [McMillan95a], which allows the pixels on the output image to be drawn in back-to-front order similar to the painter's algorithm[3].

[3] An intuitive view of the occlusion compatible order is described in Appendix B.

A major difference between 3D Image Warping and Light Field Rendering is that 3D Image Warping requires a disparity value for each pixel of the input images. With per-pixel disparity values, 3D Image Warping may reconstruct output images for viewpoints within a wide range. Therefore a typical scene can now be represented by only a few images.

2.4 3D Image Warping with a Large Set of Depth Images



(a) **(b)**

Figure 2-2: An example of 3D Image Warping. **(a)** A reference image. **(b)** A new view where the viewpoint has shifted to the right. The artifacts are shown in the blue background color.

3D Image Warping may produce artifacts such as those shown in Figure 2-2. Figure 2-2(a) shows a reference image and Figure 2-2(b) shows the results of 3D Image Warping where the viewpoint has shifted to the right. There are two types of artifacts in Figure 2-2(b). The first type of artifacts are those thin cracks on the teapots and the floor. They occur because 3D Image Warping does not move the pixels uniformly. Fortunately, those thin cracks may be resolved by better surface reconstruction methods. The second type of artifacts are caused by the exposure of previously occluded areas. These disocclusion artifacts can only be resolved by other images that can provide the missing information of those areas.

Therefore we need multiple reference images to work with 3D Image warping in order to reduce the disocclusion artifacts. The question is: how many reference images are enough? It is in fact a difficult question that I will discuss further in Section 5.4. Generally speaking, if we have a nontrivial scene and want to render new views from a wide range of possible viewpoints, then we may need a large set of depth images. For example, the scene of a cathedral that is described in Section 3.4 and Figure 3-9 still shows noticeable disocclusion artifacts with 36 reference images that are taken from 9 different positions.

However using more reference images increases the rendering cost of 3D Image Warping. If we naively apply the warping equation to every pixel in all available reference images, then the rendering time will grow linearly with the number of reference images. Clearly we need either a better rendering algorithm or a better data representation to work with a large set of depth images.

2.5 Inverse Warping

In [McMillan97] McMillan also proposed an inverse warping algorithm. For each pixel in the output image, searches are performed in all reference images to find the pixels that could be warped to the targeted pixel in the output image. If matches are found in more than one reference image, then the one representing the front-most surface wins.

Since the mapping from the pixels in reference images to the pixels in an output image is many-to-one, we cannot avoid searching all reference images (unless each reference image represents only a portion of the scene, such as the special case of displacement maps in [Schaufler99].) Although epipolar geometry (see Appendix B) limits the search space to a one-dimensional line or curve in each reference image and a quadtree-based optimization has been proposed in [Marcato98], searching through all reference images is still time consuming.

2.6 Layered Depth Image

Another way to deal with the disocclusion artifacts of image warping is to use the Layered Depth Image (LDI) [Shade98]. The main difference between an LDI and a

regular depth image is that an LDI may have a list of pixels at each pixel location. Given a set of reference images, one can create an LDI by warping all reference images to a carefully chosen camera setup (e.g. center of projection and view frustum), which is usually close to the camera of one of the reference images. When more than one pixel is warped to the same pixel location of the LDI, some of them may be occluded. Although the occluded pixels are not visible from the viewpoint of the LDI, they are not discarded. Instead, separate layers are created to store the occluded pixels. Those extra pixels are likely to reduce the disocclusion artifacts. Its limitation is that the fixed resolution of the LDI may not provide an adequate sampling rate for every reference image.

Lischinski and Rappoport used three parallel-projection LDIs to form a Layered Depth Cube [Lischinski98]. Max's hierarchical rendering method [Max96] uses the precomputed multi-layer Z-Buffers that are similar to the LDIs. It generates the LDIs from polygons and the hierarchy is built into the model.

2.7 Multiple-Center-of-Projection Image and Other Static Sets

In this section, I will discuss the other representations for combining multiple depth images. Like the layered depth image, the shortfall of these representations is that they do not recognize the different requirement of sampling rate when the viewpoint changes.

The assumption of having only one center of projection for a whole depth image contributes much to the disocclusion artifacts of 3D image warping. The Multiple-Center-of-Projection (MCOP) image proposed by Rademacher and Bishop [Rademacher98] breaks free from that assumption. Conceptually, a different camera may be associated with each pixel in an MCOP image. In their paper, they describe a particular instance of MCOP image that is acquired using a stripe-camera type of device. Each vertical column of the image is acquired from a different center of projection when the camera moves along a path.

Note that MCOP images may not prevent disocclusion artifacts. The path on which the camera is moving during the acquisition determines how completely the surfaces in the scene are sampled and how many disocclusion artifacts may still remain.

Also, combining regular planar depth images into MCOP images still remains a difficult problem.

The other method for combining multiple images is to break the images into tiles or regions and then identify the redundant regions. The main problem, again, is that it lacks the multi-resolution feature for matching the sampling rate requirement of the output image.

2.8 Volumetric Methods

Image-based rendering is not the only method in computer graphics that uses discrete samples as the primitive. In volume rendering, the scene is described by a 3D array of points which are called voxels. The volumetric data can represent not only the surfaces but also the interiors of objects by using partially transparent voxels.

The Layered Depth Image (LDI) and the hierarchical representations that are proposed in this thesis have similar structures to volumetric data. They may be considered a compressed form of volumetric data because they describe only the surfaces but not the interiors of the models.

In the following, I will discuss the most related work in volumetric methods, which are those that construct volumetric representations from range images, and the technique of Hierarchical Splatting.

2.8.1 Isosurface Extraction from Range Images

Curless and Levoy presented a volumetric method to extract an isosurface from range images [Curless96]. The goal of their work, however, was to build highly detailed models made of triangles. The volume data used in that method is not hierarchical and it relies on a run-length encoding for space efficiency.

2.8.2 Octree Generation from Range Images

There has also been work related to octree generation from range images [Chien88][Connolly84][Li94]. However the octree that is generated by those methods is

used to encode the space occupancy information. Each octree cell represents either completely occupied or completely empty parts of the scene.

2.8.3 Hierarchical Splatting

The multi-resolution volume representation in the Hierarchical Splatting work [Laur91] by Laur and Hanrahan can be considered as a special case of the LDI tree in which the LDIs are of 1×1 resolution. Hierarchical Splatting also uses the octree to control the level of detail. The octree is fully expanded initially (which is called a pyramid in their paper). Smaller children cells are combined into a larger parent cell if the children are homogeneous or if they are so similar that combing them into a larger cell results in only a small error within a pre-specified tolerance. By using various error tolerances, the same data may be represented at different levels of detail by different octrees. Therefore, an octree with higher error tolerance may be used to achieve interactive rendering while an octree with lower error tolerance may be used to produce rendering of better quality.

In hierarchical splatting, the octree to be traversed does not change with the viewpoint during rendering since it is determined by the error tolerance alone. In contrast, the octree cells of an LDI Tree that are traversed during rendering change with the viewpoint because the requirements of sampling rate for an output image change with the viewpoint as well.

2.9 Image Caching for Rendering Polygonal Models

The image caching techniques of Shade et al [Shade96] and Schaufler et al [Schaufler96] use a hierarchical structure similar to the LDI tree. Each space partition has an image (or imposter in terms of Schaufler et al) instead of an LDI. The imposter can be generated rapidly from the objects within the space partition by using hardware acceleration. However, the imposter has to be frequently regenerated whenever it is no longer suitable for the new view. The imposters may be decomposed into multiple layers to reduce the frequency of updating the imposters [Schaufler98], but the need to regenerate them is still there.

In contrast, the information stored in the LDI tree is valid at all times and does not need to be regenerated from frame to frame. By generating the LDI tree from the reference images instead of the objects within the space partitions, the LDI tree can be used for non-synthesized (i.e. acquired) scenes as well.

2.10 Summary

This dissertation builds upon McMillan's 3D Image Warping method. The previous works that are based on 3D Image Warping have been limited to two-dimensional data structures. For example, the inverse warping described in Section 2.5 and the static sets described in Section 2.7 work directly on reference images. The LDI described in Section 2.6 and the MCOP image described in Section 2.7 both provide alternative representations to the original reference images. But they are still extensions of two-dimensional images and do not consider the sampling rate issues.

Since the objects in the scene reside in a three-dimensional (3D) space, using a 3D data structure such as the one I propose in this dissertation can produce simpler or more natural algorithms. For example, merging samples and detecting redundancy between reference images become straightforward. However, using a 3D data structure brings concerns about its cost in terms of memory usage and rendering time.

I believe that the hierarchical representations proposed in this dissertation provide a solution. By considering the sampling rate issues, a hierarchical representation maintains many benefits of a 3D data structure while adaptively allocating the memory or rendering cost based on the sampling rates that are represented in the reference images and the output images. As we will see in the next two chapters, the memory and rendering costs are in fact asymptotically comparable to those of two-dimensional images.

CHAPTER 3 – HIERARCHICAL REPRESENTATION AND THE LDI TREE

One of the most challenging problems of 3D Image Warping is to minimize the disocclusion artifacts while keeping the rendering efficient; to reduce the disocclusion artifacts, we need more input depth images, but warping more depth images takes more time.

The difficulty arises from the lack of a suitable data representation. There are many redundant samples within a set of depth images since most of the objects that are visible in one image are visible in the other images. Obviously, a new data representation is required for merging the depth images. There are several desired features for this new data representation:

1. It should detect and remove the redundancy among the depth images.
2. It should be memory (space) efficient, which means the empty space should incur only little overhead in the amount of memory.
3. It should preserve the sampling rate or the level of detail in the original input images.

The Layered Depth Image (LDI) [Shade98] meets the requirements of 1 and 2, but not 3. Feature 3 is important for rendering the output image efficiently, especially when the viewpoint for the output image is far from the viewpoint of an input image because the surfaces may be sampled at very different rates (or levels of detail).

In this chapter, I will describe a hierarchical representation for merging multiple depth images. It differs from the other solutions such as the LDI by considering the sampling rates of the depth images, which are carefully preserved through the use of the hierarchy. During rendering, we only need to traverse the hierarchy to the levels that are comparable to the sampling rate of the output image.

I will first describe the details of the hierarchical representation in Section 3.1, the process of constructing the hierarchical representation from depth images in Section 3.2, and the rendering algorithm in Section 3.3. Some rendering results are shown in Section 3.4. In Section 3.5, I will discuss various issues such as the surface orientation, and the challenges of using the depth images acquired from the real world environment.

3.1 Hierarchical Representations

The goal of the hierarchical representation is to combine multiple reference images, detect redundancy across images, and most importantly preserve their sampling rates. A hierarchical representation consists of two parts: The first part is a space partitioning scheme that divides the object space into cells. The second part is a cell representation for storing the samples within a cell.

Initially, the hierarchy consists of only the top-level cell that encompasses the whole scene. As reference images are added, the hierarchy is expanded adaptively by subdividing the existing cells. Therefore the final shape of the hierarchy is determined by the sampling rates represented in the reference images. (Figure 4-2 shows an example.) In the regions where the surfaces are finely sampled by a reference image, there will be more levels of subdivided cells in the hierarchy. This process will be described in detail in Section 3.2.

The samples that are stored in the hierarchical representations are prefiltered. This means that a parent cell has all the information in its children cells, albeit at a lower sampling rate. This allows the rendering algorithm (discussed in Section 3.3) to stop at the levels with sufficient sampling rates without traversing the children cells at lower levels.

There are many possible implementations of the hierarchical representation in its generic form. The choice for the space partitioning scheme is independent of the choice for the cell representation. In this dissertation, I use a particular instance of the hierarchical representation, which is called the LDI Tree, to illustrate the detail of the data structure and various algorithms.

3.1.1 The LDI Tree

The LDI tree is an octree with an LDI attached to each octree cell (node). The octree is chosen for its simplicity but can be replaced by the other space partitioning schemes. Each octree cell also contains a bounding box and pointers to its eight children cells. The root of the octree contains the bounding box of the scene to be rendered[1]. The following is pseudo code representing the data structure:

```
LDI_tree_node = {
    Bounding_box[{X,Y,Z}, {Min,Max}]: array of real;
    Children[1..8]: array of pointer to LDI_tree_node;
    LDI[1..6]: array of Layered_depth_image;
}
```

All LDIs in the LDI tree have the same image resolution[2], which can be set arbitrarily. The height (or number of levels) of the LDI tree will adapt to different choices of resolution. In general, a lower resolution results in more levels in the LDI tree. Ultimately, we can make the resolution of the LDI be 1×1 which makes the LDI tree resemble the volume data in the Hierarchical Splatting [Laur91]. However, using a low resolution increases the memory overhead of the octree structure, which is discussed later in Section 4.1.3. We do not want the resolution to be too high either because it makes the LDI Tree look more like the regular LDI. In my implementation, I find that the resolution of 64×64 works well.

Note that each LDI in the LDI Tree contains only the samples from objects within the bounding box of the cell. I use the LDI simply as a representation to store multiple layers of surfaces within a cell. Its purpose is different from the LDI originally proposed by Shade et al, which is intended to store the samples from all visible surfaces.

For simplicity, each face of the bounding box is the projection plane of one of the 6 LDIs within a cell. Orthographic projection is used and the projection direction is perpendicular to the projection plane. The depth values that are stored in an LDI range

---

[1] For outdoor scenes, background textures can be added to the faces of the bounding box. The bounding box can be extended with little overhead if most of the space is empty.

[2] In other words, the LDIs have the same number of grid points. However, the *spatial* resolution may vary, depending the size of the cell.

from 0.0 to 1.0, where a sample on the projection plane receives depth value 0.0 and a sample on the opposite face of the bounding box receives depth value 1.0.

In my earlier work [Chang99] I used only one LDI per octree cell. The reason why I switched to the 6-LDI representation is to deal with the resampling issues arising from surfaces that are nearly perpendicular to the projection plane of the LDI. This will be explained in more detail in Section 3.5.1. Using 6 LDIs per cell also allows me to distinguish the front side and back side of a thin surface, and to implement a back-face culling algorithm that I will explain further in Section 3.3.4.

3.1.2 Memory Efficiency

Conceptually, a layered depth image (LDI) can be described by the following pseudo code:

```
Layered_depth_image = {
    Nx, Ny: integer;  // image resolution
    Pixel[1..Nx, 1..Ny]: array of pointer to
         Layered_pixel;
}
Layered_pixel = {
    Next: pointer to Layered_pixel;
    Color: color_type;  // e.g., R, G, B, and Alpha
    Depth: real;
    // Other flags or tags may be added here.
}
```

It must be pointed out here that an LDI in the LDI Tree may have many empty pixels in most or all of its grid points. This is especially true in cells that contain no objects or only small objects. To prevent the LDI Tree from using too much memory, it is important to use a data structure that is suitable for a sparse LDI.

My choice is to associate with each pixel its coordinates in the LDI, which leads to the following modified pseudo code:

```
Packed_layered_depth_image = {
    Count: integer;
    Pixel[1..Count]: array of Packed_layered_pixel;
}
Packed_layered_pixel = {
    X, Y: integer;  // coordinates
```

```
    Color: color_type;  // e.g., R, G, B, and Alpha
    Depth: real;
    // Other flags or tags may be added here.
}
```

This is different from the “Space Efficient Representation” described in the original LDI paper [Shade98], which is more suitable for a dense LDI. As in the original LDI paper, the pixels are stored in a linear array during rendering to maintain the spatial locality in memory. But a pointer-based data structure for the LDI is used when creating the LDI Tree.

3.1.3 The Generic Form

We may construct other forms of hierarchical representation by choosing either a different space partitioning scheme or a different cell representation from the one that is used in the LDI Tree. For example, we may use the k-d-tree or the BSP-tree [Fuchs80] instead of the octree as the space partitioning scheme. We may also use only one LDI per cell in the cell representation as I did in my earlier work [Chang99]. It is easy to verify that the construction and rendering algorithms that I will describe later in this chapter are portable to hierarchical representations other than the LDI Tree.

The only caveat is that the cell representation must be memory efficient. This means that its memory size must be compact and should grow only when samples are added. This excludes cell representations such as the volumetric representation.

3.2 Construction of the LDI Tree from Source Images

The initial input to a 3D image warper is a set of depth images, which are also often referred to as source images or reference images. We call them depth images because they are color images enhanced with per-pixel depth values. Each depth image also contains camera information, which may be represented as vectors that define a pin-hole camera model (as illustrated in Figure 3-1). It may also be considered as the basis of its own coordinate system, which allows each pixel of the image to be projected back to the 3D coordinates of the object that the pixel samples.

In summary, a typical depth image will contain the following parameters:

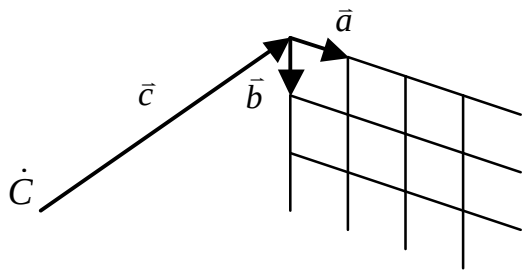


Figure 3-1: A pinhole camera model.

1. N_x, N_y: the size of the image.
2. $\vec{a}$, $\vec{b}$, $\vec{c}$, $\dot{C}$: the camera information (Figure 3-1).
3. *color[1..N_x, 1..N_y]*: an array of colors. Each color may be stored as multiple components such as the RGB values.
4. *depth[1..N_x, 1..N_y]*: an array of depth values. They are typically stored as floating point or fixed point real numbers.
5. *normal[1..N_x, 1..N_y]*: an array of normal vectors. They may be derived from the depth values if they are not directly provided as input. My implementation uses four adjacent pixels to compute the slopes in X and Y directions, then takes their cross product to obtain the normal vector of each pixel. If a discontinuity between adjacent pixels is detected from the sudden change in the slopes along either X or Y direction, then the greater slope is discarded.

The sampling rate or the quality of a pixel of a reference image depends on its depth information. For example, if (part of) a reference image represents a surface that is far away, then those pixels that describe that surface do not provide enough detail when the viewer zooms in or walks toward that surface. Conversely, warping every pixel of a reference image taken near an object is wasteful when the object is viewed from far away.

The LDI Tree is constructed from reference images by selecting an octree cell for each pixel of the reference images to store the pixel in an LDI of that cell, then filtering the affected LDI pixels to the LDIs of all ancestor cells in the octree. In the following

subsections, I will first describe the construction process with a single reference image (Section 3.2.1). Then I will discuss how the samples are merged and the redundancy removed when multiple reference images are involved (Section 3.2.2). After all reference images are added, the samples are filtered to complete the LDI Tree construction (Section 3.2.3).

3.2.1 Storing a Reference Image into the LDI Tree

In 3D image warping, each pixel of the reference images contains depth information that is either stored explicitly as a depth value (d) or implicitly as a disparity value (δ). This allows us to project the center of the pixel to a point in the space where the scene described by the reference images resides.

To be consistent with the notations in [McMillan97], I use the disparity value to represent the depth information in the following discussion. The disparity value (δ) is closely related to the depth information (d). Their relationship is shown in the following equation:

$$d = f / d$$

$$f = \vec{c} \cdot \frac{(\vec{a} \times \vec{b})}{\left\| (\vec{a} \times \vec{b}) \right\|}$$

where f is the distance from the center of projection to the projection plane.

When a pixel is projected to the 3D object space, we get a point representing the center of the projected pixel and a "stamp size." The center for the pixel (u,v) is computed as:

$$\dot{C} + (u\vec{a} + v\vec{b} + \vec{c}) / d \qquad (1)$$

where $\dot{C}$ is the viewpoint and $\vec{a}$, $\vec{b}$, $\vec{c}$ define the basis of the non-orthogonal coordinate space of the camera, which was previously defined in Section 3.2. The stamp size S is calculated by:

$$S = S_X \times S_Y \qquad (2)$$
$$S_X = |\vec{a}| / d$$
$$S_Y = |\vec{b}| / d$$

To simplify the discussion, I do not consider the orientation of the object surface from which the pixel is taken. I also ignore the slight variation of stamp size at the edges of the projection plane (for example, if we are interested in the solid angle that is covered by a pixel).

An octree cell is then selected to store this pixel. The center location determines which branch of the octree to follow. The stamp size determines which level (or what size) of the octree cell should be used. The level is chosen such that the stamp size approximately matches the pixel size of the LDI in that cell.

After an octree cell has been chosen, the pixel can then be warped to one of the LDIs in that cell. The details of the warping are described in [Mark97]. The normal vector associated with the pixel determines which one of the 6 LDIs is used. By simply checking which of the X, Y, and Z coordinates of the normal is the largest in magnitude, we can narrow down our choices to two of the 6 LDIs. Then its sign determines which one of the two LDIs is chosen.

Usually, the center of the pixel will not fall exactly on the grid of the LDI, so resampling is necessary. This is done by splatting [Westover91] the pixel to the neighboring grid points. In this system I use a bilinear kernel. Usually four LDI pixels are updated for each pixel of a reference image. More specifically, the alpha values that result from the splatting are computed by:

$$\text{alpha} = W_X W_Y \tag{3}$$

$$W_X = \begin{cases} Kernel(|X_i - X_c|, \frac{S_X}{P_X}), & S_X > P_X \\ Kernel(|X_i - X_c|, 1) * \frac{S_X}{P_X}, & S_X \le P_X \end{cases} \tag{3a}$$

$$W_Y = \begin{cases} Kernel(|Y_i - Y_c|, \frac{S_Y}{P_Y}), & S_Y > P_Y \\ Kernel(|Y_i - Y_c|, 1) * \frac{S_Y}{P_Y}, & S_Y \le P_Y \end{cases} \tag{3b}$$

$$P_X = B_X / N_X$$

$$P_Y = B_Y / N_Y$$

$$Kernel(d, s) = \begin{cases} 1 - \frac{d}{s}, & d \le s \\ 0, & d > s \end{cases}$$

where B_X and B_Y are the physical sizes of the LDI projection plane (which is a face of the bounding box) in object space. N_X and N_Y are the resolutions of the LDI. P_X and P_Y represent the physical size of the pixel that is being splatted. S_X and S_Y represent the stamp size as defined in Equation 2. (X_c, Y_c) is the center of splatting in the selected LDI and (X_i, Y_i) is one of the grid points covered by the splatting. The conditions in Equations 3a and 3b guarantee that the splat size will not be smaller than the LDI grid size, which represents the maximal sampling rate of the LDI.[3]

The reader is encouraged to verify that the alpha values sum to 1.0 for a planar surface that is uniformly sampled in a reference image and is parallel to the LDI. It is true not only for the octree cell we have chosen, but also for the ancestor cells where the same pixel is splatted to a smaller area (because more pixels will be splatted). Therefore Equation 3 is also used during the filtering process, which will be described in Section 3.2.3.

The above are summarized in the following pseudo code:

[3] It is similar to how the subpixels are prefiltered in supersampling for antialiasing.

```
 AddDepthImage(Image) {
1.   For each pixel of Image {
2.       Compute Pixel_Center and Stamp_Size;
3.       AddSample(Root_Cell, Pixel_Center, Stamp_Size);
     }
 }

 AddSample(Octree, Pixel_Center, Stamp_Size) {
1.   If Pixel_Center is not inside Octree,
2.   then return;
3.   If Stamp_Size is too big for Octree then {
4.       Subdivide Octree if it has no children cells;
5.       Child = the child cell of Octree that
         Pixel_Center is inside.
6.       AddSample(Child, Pixel_Center, Stamp_Size);
     }
7.   else {
8.       Splat the pixel to the LDIs in the Octree cell;
     }
 }
```

3.2.2 Sample Merging and Redundancy Removal

An LDI pixel may get contributions from many pixels of the same surface. When only one reference image is involved, they may only come from the overlapping splats of neighboring pixels in the reference image. When multiple reference images are involved, an LDI pixel may get contributions from pixels in different reference images that sample the same surface. The contributions from those pixels must be merged and blended together. Figure 3-2(a) shows an example of those cases. When we merge the pixels from different reference images that sample the same surface, we have detected and removed the redundancy between those reference images.

An LDI pixel can also get contributions from many pixels of different surfaces. In those cases, we assign them to different layers of the LDI pixel. Figure 3-2(b) shows an example of those cases. To determine whether they are from the same surface or not, we check the difference in their depth value against a threshold. We select the threshold to be slightly smaller than the spacing between adjacent LDI pixels, so that the sampling rate of a surface that is perpendicular to the projection plane of the LDI can be preserved.

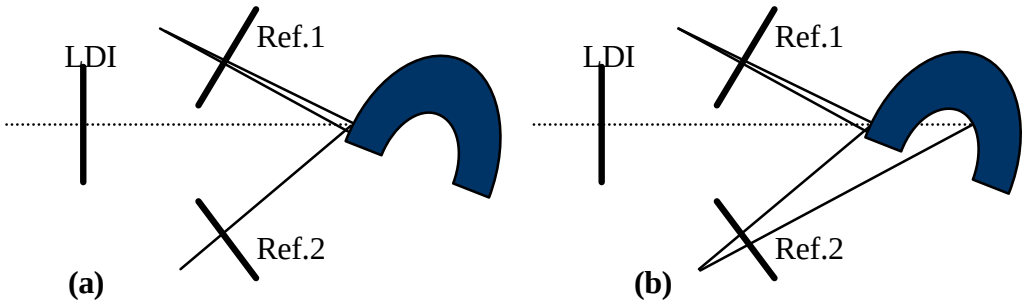


Figure 3-2: Examples of pixels that are warped to the same pixel location in an LDI. **(a)** Two pixels from reference image 1 and a pixel from reference image 2 are taken from the same region of a surface. Blending is used to combine their contribution to the LDI pixel. **(b)** One of the pixels from reference image 2 is taken from a different surface. A separate layer in the LDI is created to accommodate its contribution to the same LDI pixel.

Note that we may encounter an interesting situation where multiple reference images sample the same surface but at different sampling rates. In that case, the sample merging will happen after the sample from a lower level is filtered to the higher levels during the filtering process that will be discussed next.

3.2.3 Filtering

The splatting process described in Section 3.2.1 stores each pixel of the reference images into only one cell initially. After the filtering process, each pixel also contributes to the parent cell and all ancestor cells of the initially chosen cell. The result is that a cell also contains the samples of its descendants, even though those samples are filtered down to the lower sampling rate of the current cell. Therefore, later in the rendering process we need not traverse the children cells if the current cell already provides enough detail.

The filtering is done by splatting each pixel to the LDIs of all ancestor cells. Note that the same splatting described in Section 3.2.1 may be used for filtering as well, even though the stamp size is smaller than the pixel size of the LDIs in ancestor cells. The filtering may be performed either on the fly while each pixel of a reference image is being added or in batch mode after all reference images have been added to the LDI Tree.

I classify the pixels in the LDI Tree into two categories: *unfiltered* and *filtered*. The unfiltered pixels are those that come from splatting to the octree cell that was

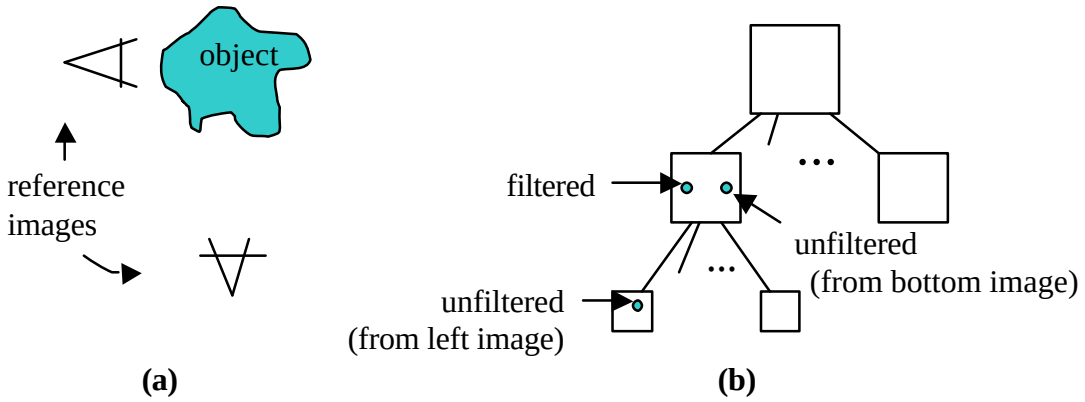


Figure 3-3: When an object is sampled from multiple reference images, its samples might not be merged until the filtering process. **(a)** shows the object and two reference images. **(b)** shows the structure of the octree and where the samples are stored. The filtered samples from the left image and the unfiltered samples from the bottom image might be merged if they are from the same area of the object.

initially chosen for a reference image pixel. Those pixels that come from the splatting to the ancestor cells during filtering are classified as filtered. Note that an unfiltered pixel may be merged with a filtered pixel during the filtering process. The merged pixel is considered filtered because better-sampled pixels exist in the LDIs of some children cells of the current cell.

An example is shown in Figure 3-3. Figure 3-3(a) shows an object and two reference images that sample the object at different distances. Figure 3-3(b) shows the schematic or structural view of the octree cells that contain the samples. Because the upper left image sees more details of the object at a closer distance, its pixels are splatted to the cell in the lower level (which represents the higher sampling rate) and stored as unfiltered samples. The pixels from the bottom image are splatted to the middle level initially because the bottom image samples the object at a lower sampling rate.

A common mistake is to think that unfiltered samples exist only in leaf cells of an LDI Tree. Figure 3-3 shows a counter-example, where one of non-leaf cells in Figure 3-3(b) contains both filtered and unfiltered samples. Those unfiltered samples in non-leaf cells must be taken into account in the design of rendering algorithm, which I discuss next.

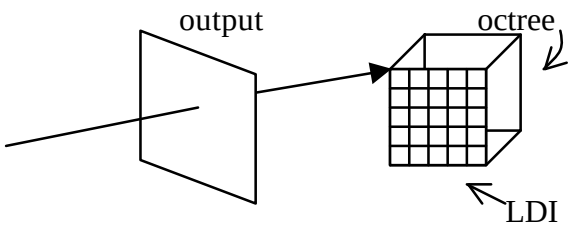


Figure 3-4: To estimate the range of stamp size for all pixels in the LDI, the corners of the bounding box are warped to the output image.

3.3 Rendering from the LDI Tree

A new view of the scene is rendered by warping the LDIs in the octree cells to the output image. The advantage of having a hierarchical model is that we need not render every LDI in the octree. We can render those cells that are farther away at less detail by using the filtered samples that are stored in the LDIs higher in the hierarchy.

To start the rendering, I traverse the octree from the top-level cell (i.e. the root). At each cell, I first perform view frustum culling, then check whether it can provide enough detail if its LDI is warped to the output image. If the current cell does not provide enough detail, then its children are traversed. An LDI is considered to provide enough detail if the pixel stamp size covers about one output pixel. Therefore the traversal of the LDI tree during the rendering will adapt to the resolution of the output image. Note that we do not calculate the pixel stamp size for each individual pixel in an LDI. Because all the pixels in the LDI of an octree cell represent samples of objects that are within its bounding box (as shown in Figure 3-4), we can estimate the range of stamp size for all pixels of the LDI by warping the LDI pixels that correspond to the corners of the bounding box. The corners of the bounding box are obtained by placing the maximal and minimal possible depth at the four corner pixel locations of the LDI. We use Equation 2 to compute the stamp size with the vector $\vec{a}$ and $\vec{b}$ of the output image and the disparity value δ obtained from the warping. Note that a special case exists if the new viewpoint is within the octree cell. When this happens we traverse the children.

The pseudo code for the octree traversal follows:

```
 Render (Octree) {
1.   If outside of view frustum,
     then return;
2.   Estimate the stamp size of the LDI pixels;
3.   If LDI stamp size is too large or the viewer is
     inside the bounding box then {
4.       Call Render() recursively for each child;
5.       Warp the unfiltered pixels in LDI to the Output
         buffer; }
6.   else {
7.       Warp both unfiltered and filtered pixels in LDI
         to the output buffer; }
 }
```

Note the difference in step 5 and step 7 of the pseudo code. As mentioned in Section 3.2.3, each LDI in the octree may contain both unfiltered and filtered pixels. When we warp both the LDI in a parent cell and the LDI in a child cell, the filtered pixels in the parent cell should not contribute to the output image because the unfiltered pixels in the child cell already provide better sampling for the same part of the scene.

Let us revisit the example in Figure 3-3. If the octree traversal during the rendering reaches the bottom level, we will render all the unfiltered samples but none of the filtered samples in the middle cell. However the unfiltered samples in the middle cell (if they are not merged during filtering) will be rendered. Here we see why it is necessary to classify filtered and unfiltered samples. Without the classification, we would not be able to tell which samples in the middle cell are those unfiltered samples from the bottom reference image and need to be rendered.

One feature of the original LDI is that it preserves the occlusion compatible order in McMillan's 3D warping algorithm [McMillan95a][McMillan95b]. However this feature is lacking in the LDI tree. Although the back-to-front order can still be obtained within an LDI and across LDIs of sibling cells of the octree, I cannot obtain such order between LDIs of a parent cell and a child cell. This causes problems when unfiltered samples exist in both parent and child cells. In addition, the warped pixels are partially transparent due to the splatting process (see Equation 3 in Section 3.2.1). Therefore, I need to keep a list of pixels for each pixel location in the output buffer. I implement the output buffer as an LDI, which resembles the A-Buffer [Carpenter84]. At the end of the

rendering, each list is composited to a color for display. The details of the compositing are discussed next.

3.3.1 Compositing in the Output Buffer

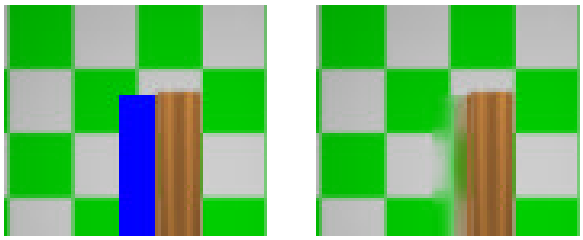
As discussed above, each pixel location of the output image may have a list of semi-transparent pixels at the end of the rendering process. To obtain the final color for each pixel location, I sort the list of pixels in depth and then use alpha blending starting from the front of the sorted list. An exception is that two pixels with similar depth should be merged first and their alpha values summed together before they are alpha-blended with the other pixels. That is because they are likely to represent sampling of the same surface.

Therefore, the pixel merging is also performed in the output LDI, which is similar to the pixel merging in the LDI of the octree cell as discussed in Section 3.2.2. The difference is that a single threshold value of depth difference does not work anymore because the pixels can come from different levels of the LDI tree. This difficulty is solved by attaching the level of octree cell which the pixel comes from to each pixel in the output LDI. The threshold value that is used for that level of octree is then used to determine whether two pixels in the output LDI should be merged.

Because the objects in the rendered scene are not uniformly sampled, some surfaces may look translucent. If that is not desired, we may loosen the definition of opaqueness to include pixels with slightly lower alpha values and allow the alpha blending of the sorted list to stop when the desired opaqueness is reached.

3.3.2 Hole Filling

When we construct the LDI tree from many reference images, chances are we have eliminated most of the disocclusion artifacts. However, it is possible that some disocclusion artifacts still remain. I propose a two-pass algorithm that uses the filtered pixels in the LDI tree to fill in the holes in the output image. The algorithm consists of the following steps:



(a) (b)

Figure 3-5 An example of hole filling. **(a)** The blue hole is the disocclusion artifact when the viewpoint moves to the left. **(b)** The hole is filled by the algorithm described in Section 3.3.2. Note the contribution from the part of checkerboard that is hidden under the wood.

1. Render the output image from the LDI tree as discussed in Section 3.3. This is the first pass.
2. A stencil (or coverage of pixels) is then built from the output image.
3. Render the output image from the LDI tree again. But in this pass, splat only the filtered pixels that were skipped during the first pass, i.e. step 5 of the pseudo code in Section 3.3 is changed to splat only filtered pixels and step 7 is changed to do nothing.
4. Use the stencil from step 2 to add the image from step 3 to the output image rendered from step 1.

The stencil from step 2 allows the filtered pixels to draw only to the holes in the output image from step 1. This assumes that the output image would be completely filled if no disocclusion artifact occurred.

The two-pass hole-filling algorithm is not completely accurate. For example, the filtered samples in higher levels of the hierarchy are not needed during hole filling if the filtered samples at slightly lower levels already fill all holes. This means that the second pass of the hole-filling algorithm may actually fill the holes with an image that is blurrier than necessary. However, the two-pass hole-filling algorithm is still a practical implementation since a slightly blurrier result is acceptable for hole filling purposes and an accurate implementation may require too many passes to be efficient.

What distinguishes my hole-filling method from the other work is the fact that occluded surfaces may contribute to the hole filling if they are near boundary of a hole. That is because my method is based on the filtered samples that reside in the 3D domain. Figure 3-5 shows such an example. In contrast, the other hole-filling methods often utilize only the information on the output image that comes from visible surfaces.

3.3.3 Progressive Refinement

Another feature of the hierarchical representation is that it is easy to produce output images at reduced resolutions. The rendering algorithm (described in Section 3.3) will adapt to the reduced resolutions and fewer octree cells will be traversed. This provides opportunities for progressive refinement during the rendering of output images.

The simplest method to create the effect of progressive refinement is to render the LDI tree to a low-resolution output image first, then increase the resolution gradually. However, this method does not utilize the coherence between the renderings of two different resolutions.

To utilize the coherence between two renderings, we can tag the octree cells that are traversed in the previous rendering and skip them in the current rendering. Note that some filtered pixels may have been warped to the output buffer if they are from the leaf nodes of the subtree traversed of the previous rendering[4]. Those pixels must also be tagged so they can be removed from the output buffer if the leaf nodes in the previous rendering become interior nodes of the current rendering.

3.3.4 Back-Face Culling

Since each octree cell of the LDI Tree contains 6 LDIs, it is likely that at least one of the LDIs is backward facing and does not need to be warped during the rendering. Note that culling a backward facing LDI is different from simply culling a back face of an octree cell because a face of octree cell represents only the projection plane of an LDI. The actual samples that are stored in an LDI may come from surfaces that are still

[4] See line 7 of the pseudo code in Section 3.3.

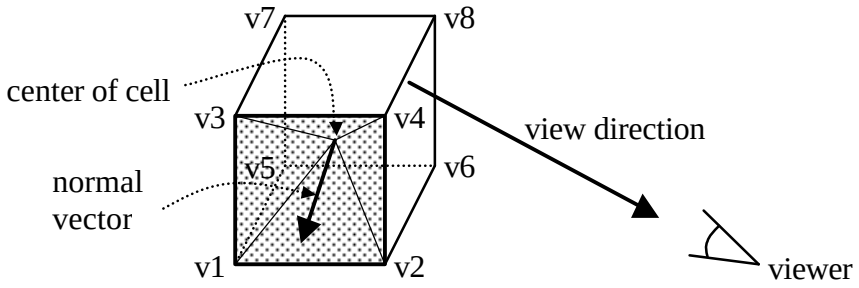


Figure 3-6: How to identify a possibly forward-facing LDI. The back-face culling algorithm culls away those LDIs that cannot be identified as forward facing.

forward facing even when the projection plane of the LDI is backward facing. The following describe my back-face culling algorithm to detect those LDIs.

Figure 3-6 explains the thinking behind my back-face culling algorithm. Instead of finding the backward facing LDIs, I identify the LDIs that may contain samples from forward facing surfaces. If we mark all those LDIs, then the backward facing LDIs are simply those that are left unmarked.

Assume that the rectangle (v1, v2, v3, v4) is the projection plane of an LDI. The LDI may contain samples from surfaces whose normal vectors are within the range represented by the pyramid formed by v1, v2, v3, v4 and the center of the cell[5]. To determine whether any normal vector within that range may be facing the viewer, we need to consider all possible viewing directions. This can be achieved by comparing the normal vectors with 8 viewing directions, each of which is formed between the viewer and one of the cell vertices. For example, if we form the first vector from the cell center to any one of v1-v4 and the second vector from any of v1-v8 to the viewer, then the LDI may contain forward-facing samples if the dot product of the two vectors is positive. The first vector represents any possible normal vector and the second vector represents any possible viewing direction.

[5] As discussed in section 3.2.1, a sample is assigned to one of the 6 LDIs of the cell based on the magnitudes and signs of the coordinates of its normal vector. If the cell does not have equal sides, then this pyramid is computed with a unit cube instead.

The above observations lead to a back-face culling algorithm, which is shown in the following pseudo code:

```
 Back_Face_Culling (Octree) {
1.   Mark all 6 LDIs as back-faced;
2.   for (vi = v1 to v8) {
3.       normal = vi - center;
4.       for (vj = v1 to v8) {
5.           viewDir = viewer – vj;
6.           if dot(normal, viewDir) > 0 then {
7.               mark the 3 LDIs containing vi as front-
                 faced;
8.               exit early from the vj loop;
             }
         }
     }
9.   Cull away the LDIs that are still marked back-
     faced;
 }
```

3.4 Results

I demonstrate the results of the LDI Tree algorithms using a model of the interior of a cathedral, Palladio's Il Redentore in Venice [O'Brien93]. The reference images are generated by ray-tracing using the Rayshade program [Kolb94]. Each reference image has 512×512 pixels and a 90-degree field of view. The reference images are taken from up to 25 different positions, with four images taken at each position to complete a 360-degree horizontal field of view.

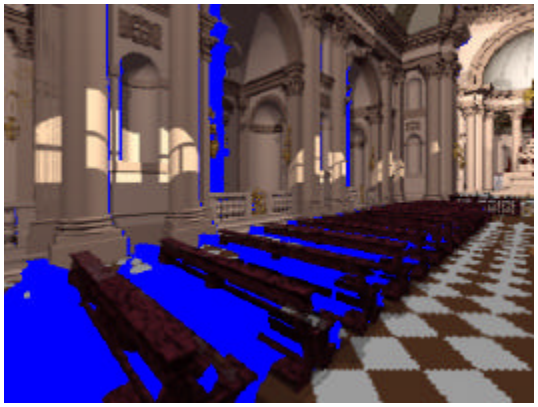


Figure 3-7: A new view from four reference images (taken at the same position).

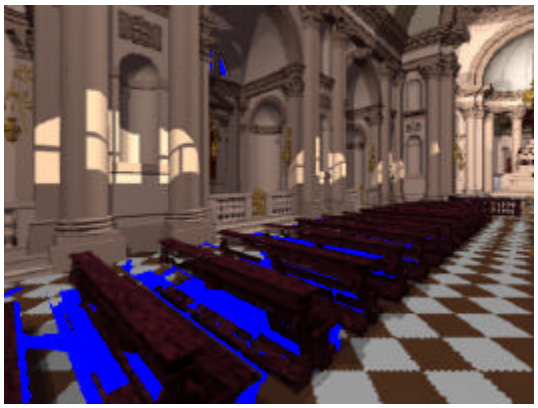


Figure 3-8: A new view from 12 reference images (taken at three different positions).

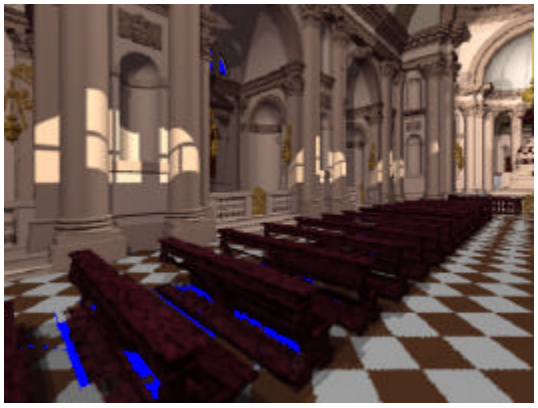


Figure 3-9: A new view from 36 reference images (taken at 9 different positions).

Figure 3-10: A new view from 36 reference images (taken at 9 different positions). Hole filling is enabled.

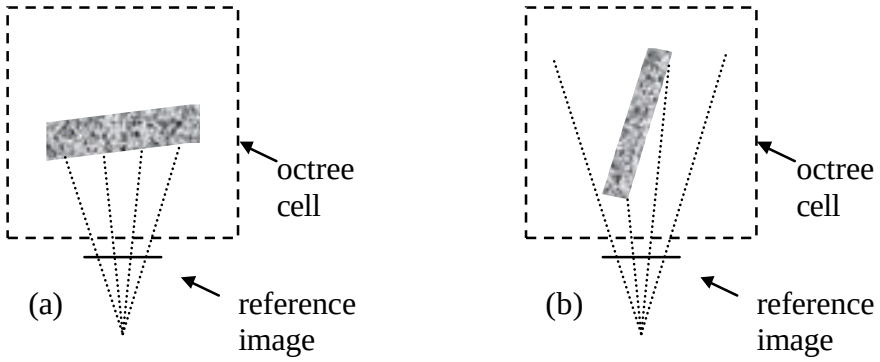


Figure 3-11: Surface orientation may affect how well a surface is sampled. **(a)** A surface is well sampled when it is facing the viewpoint. **(b)** A surface is poorly sampled when it is viewed at a grazing angle.

Figure 3-7 to Figure 3-9 show the effects of disappearing disocclusion artifacts when more reference images are added. I disabled the gap filling to let the disocclusion artifacts appear in blue background color. Figure 3-7 has severe disocclusion artifacts because only four reference images from the same viewpoint are used. Figure 3-8 and Figure 3-9 show the same view but with 12 and 36 reference images (from 3 and 9 viewpoints) respectively. Figure 3-10 is generated from the same reference images as Figure 3-9 but with hole filling enabled.

3.5 Discussion

3.5.1 The Problem of Near-Perpendicular Surfaces

When a reference image is stored in an LDI Tree using the method described in Section 3.2, the calculation of sampling rate is based on depth information only. However, as shown in Figure 3-11, the surface orientation also affects how well a surface is sampled by a reference image. When a reference image sees a surface at a grazing angle, the reference image samples the surface with fewer pixels than it would if the surface is facing it.

It is possible to include the surface orientation in the computation of sampling rates by an approach similar to the splat size computation in the original LDI paper [Shade98]. However, finding how the surface orientation changes the splat size alone is

not sufficient either, because the surface orientation also affects the shape of the splatting. For example, a vertical surface that is almost perpendicular to the reference image may be poorly sampled in the horizontal direction of the reference image but still well sampled in the vertical direction. The splatting is thus stretched in one direction only. This problem is similar to the filtering in anisotropic texture mapping [Williams83] [McCormack99].

A possible extension to my algorithm is to include the surface orientation in the computation of sampling rates and the shape of splatting. However, such an extension is not as important as it seems for the LDI Tree because the LDI Tree can handle the poorly sampled surfaces in a similar fashion to the occluded surfaces. More specifically, the other reference images may look at the same surface from a better angle and eliminate the problem. If no other reference image has a better view of the surface, then it is a limit of the input data and we simply do not have sufficient samples for that surface. So from a different perspective, we may consider the artifact that is caused by the surfaces viewed from grazing angles as a special case of the disocclusion artifact.

The surface orientation poses a different problem when the surface is almost perpendicular to an LDI. Figure 3-11(a) shows an example if the right wall or left wall of the octree cell is the projection plane of an LDI. Even though the surface is well sampled in the reference image, the samples are splatted to only a few pixels when the surface is resampled to the LDI. This will result in long lists at a small number of pixels of the LDI. It is indeed a limitation of the LDI which was mentioned by Shade et al in the discussion section of their LDI paper [Shade98].

By using 6 LDIs per octree cell, I avoid the above problem because a different wall of the octree cell will be chosen as the projection plane.

3.5.2 Real-World Environment

So far, I have assumed that the depth information from reference images is accurate. However, acquiring source images with precise depth information from real-world environment is still a difficult task. Does the LDI Tree still work when errors are present in the depth values?

The answer is yes if the errors are small. The sample merging and redundancy removal scheme of the LDI Tree (described in Section 3.2.2) requires depth information to the precision that is approximately equal to the distance between two adjacent samples. This means that the LDI Tree can tolerate small errors in the depth. Furthermore, we may consider using a lower sampling rate for samples with errors in depth that is only slightly larger than the tolerance.

I believe the above requirement of small error rates is reasonable because larger errors will likely cause difficulty in registration of reference images that are taken from different centers of projection. The registration problem between reference images taken from real-world environment is not within the scope of my work. (Had the LDI Tree been able to handle larger errors, it would have solved the registration problem by transforming the registration problem into an LDI Tree problem.)

I have tested my algorithms on the reading room model which was acquired by Nyland et al using a time-of-flight laser range-finding device [Nyland99]. Unfortunately the errors are beyond the tolerance of the LDI Tree. Therefore, the samples that are from the same surface but taken at different centers of projection are not merged. Most of the errors are in fact due to the misregistration of scans across different centers-of-projection. However, the LDI Tree can still be useful for the purpose of hole filling.

Figure 3-12 and Figure 3-13 show the results when it is applied to the reading room model. The reference images are taken from the same center of projection. The hole filling algorithm is disabled in Figure 3-12 but is enabled in Figure 3-13.

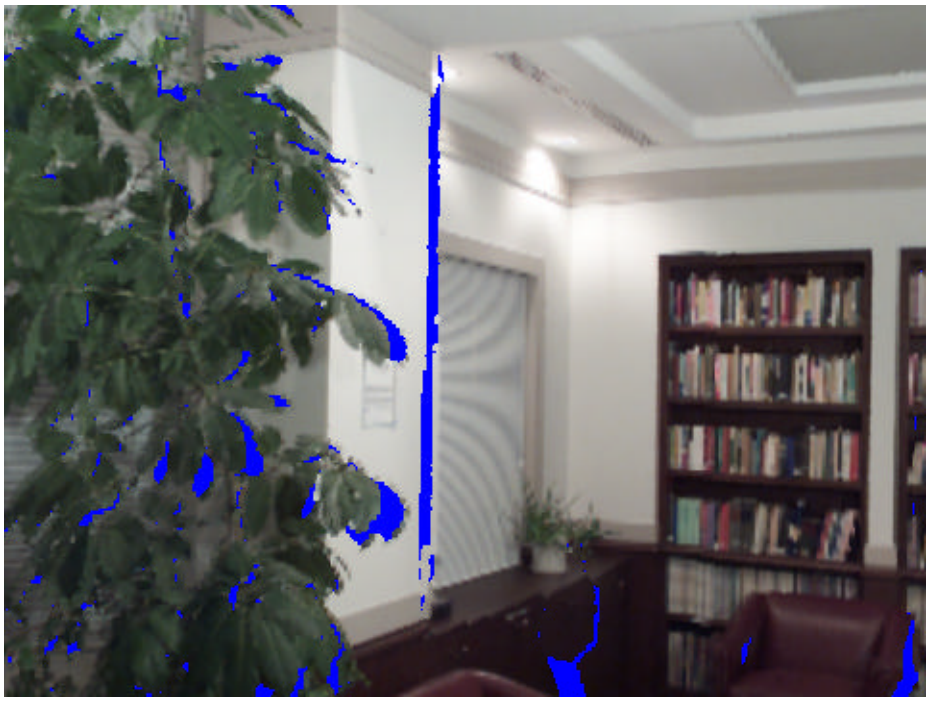


Figure 3-12: A new view from the reading room model which is captured from real-world environment.

Figure 3-13: The same view as Figure 3-12 with hole filling enabled.

CHAPTER 4 – COST ANALYSIS

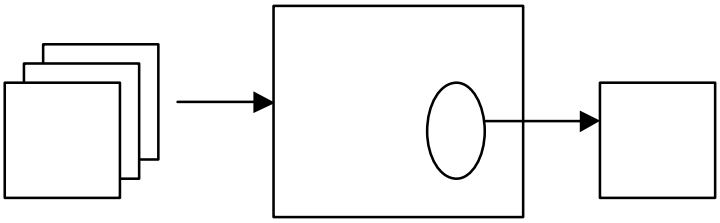


Figure 4-1: The data flow of an LDI Tree.

The hierarchical representations and the LDI Tree are three-dimensional (3D) data structures. A natural concern is how much memory they need and how efficient the rendering algorithm is. I answer those questions in this chapter.

To put the above questions in perspective, keep in mind that the strength of the hierarchical representations is to model a large environment where a large number of reference images are required. To keep the information that is represented in all those images does require large amounts of storage space. However, we need only a subset of the available information during the rendering. In other words, the memory or storage requirement is driven by the input data, while the data accessed during the rendering are driven by the requirement of the output image. These relationships are depicted conceptually in Figure 4-1.

4.1 Memory Usage

There are two approaches to analyze the memory usage of an LDI Tree. The first approach is to define the memory usage as a function of the number of reference images and the number of pixels in each reference image. In Section 4.1.1, I use the first approach to show a loose upper bound of memory usage that grows linearly with the total number of pixels in reference images. The second approach is to define the memory

usage as a function of the sum of all surface areas in the scene. In Section 4.1.2, I use the second approach to show that there exists a theoretical upper bound that is independent of the number of reference images.

The first approach shows that although the LDI Tree is a 3D data structure, its memory usage is limited by the size of the two-dimensional (2D) reference images. It reflects a scenario where the sample merging never occurs and thus the memory grows linearly with the number of reference images used. In practice, such a scenario rarely happens and the memory usage does level off when more reference images are added. The existence of such an upper bound is supported by the second approach of analyzing the memory usage. However, computing the upper bound requires knowledge of the surface areas, which are difficult to obtain from reference images.

I validate my analyses by experiments, which are shown in Section 4.1.4.

4.1.1 Loose Upper Bound

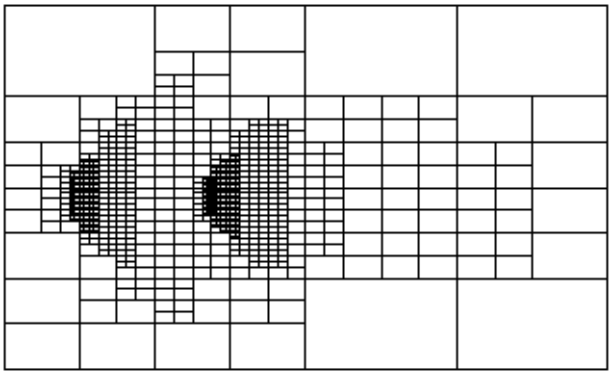


Figure 4-2: Top view of the octree cells after two reference images are added to an LDI Tree.

The concern that an LDI Tree consumes too much memory might come from perceiving it as a nearly complete octree, i.e., as if the octree is almost fully expanded at each non-leaf cell. However that is rarely the case because an octree cell is expanded (i.e. subdivided) only if it receives pixels that exceed its sampling rate from reference images. Figure 4-2 shows an example where the octree cells of an LDI Tree are viewed from the top when two reference images have been added to the LDI Tree. Those finely

subdivided cells are those that are near the camera positions of the reference images because they receive samples at greater level of detail.

One way to estimate the memory usage of an LDI Tree is to look at how the memory is allocated during the construction process. When we construct the LDI Tree from reference images, we add a constant number of unfiltered LDI pixels to the octree cell chosen for each pixel of the reference images. We also add a constant number of filtered LDI pixels to each of the h ancestor cells. That means the amount of memory used by the LDI Tree grows in the same order as the amount taken by the original reference images, only if h is bounded.

We can further assume that h is bounded because the maximal height of the LDI Tree exists. It is because we are not interested in infinite details of surfaces. This limit may be characterized by defining a minimal viewing distance.

Let L be the longest side of the bounding box of the scene, N be the resolution of an LDI, d be the smallest feature in the scene the human eyes can discern at a minimum distance, and H be the maximal height of the LDI Tree. Then we have:

$$H = \left\lceil \log_2 \frac{L}{N \times d} \right\rceil$$

Therefore h is bounded and we have obtained an upper bound of the memory size. This upper bound grows linearly with the total number of pixels in the reference images. It is a loose upper bound because it assumes that no sample merging occurs, which should not happen for applications that require the use of LDI Trees. Although this loose upper bound reveals little about the real memory size, it does show that the memory usage of an LDI Tree is akin to that of 2D images.

4.1.2 More Practical Upper Bound

An LDI Tree is an intermediate data structure for merging the reference images. Ideally, its memory usage grows only if new information is added. In other words, we may consider an LDI Tree as a database for storing the information that is available in the reference images, and maintaining the indexes of 3D locations. The new information that a reference image adds to the database may be separated into two categories:

- It provides information on surfaces that were not seen by previous reference images,
- Or it provides more detailed (i.e. higher sampling rate) information of surfaces that were already seen.

So what we may expect is that the memory size will grow rapidly when the first few reference images are added to the LDI Tree because most of the information in those images is new. But the growth should slow down as more images are added, containing mostly redundant information. However, will the growth stop at all?

It is likely that the growth from the new information in the first category, i.e. from previously occluded surfaces, will decelerate quickly. In fact, it will be a good thing even if it keeps growing because the reason why we took so many reference images was to see those surfaces. However, there is no guarantee that the growth from the second category will slow down because a reference image may add more detailed views of certain surfaces. To take it to extreme, we may take microscopic views of the surfaces in the scene! But in practice those kinds of details are rarely desirable. Usually we are only interested in details down to a certain level. By limiting the level of details, we can stop the growth from the new information in the second category when all the desired details have been recorded.

There are many different ways to specify the level of details we are interested in. For example, we may use any one of the following:

- Specify the minimal distance between two sampling points on a surface.
- Specify the minimal viewing distance between the viewer and a surface, when the field of view and the image size (or resolution) are given.
- Specify the maximal level of the hierarchy in the octree.

When the level of detail is limited, no more new information can be added when every surface has been sampled to that level of detail. Therefore the maximal amount of information will be proportional to the sum of all surface areas. This means that we have an upper bound of memory usage. Note that unlike the loose upper bound we derived in Section 4.1.1, this upper bound does not depend on the number of reference images.

To verify that the memory usage of an LDI Tree observes this upper bound, we need knowledge of the surface areas, which is difficult to obtain from the input reference images alone. However we do have the prior knowledge of surface areas for reference images that are generated synthetically from polygonal models. One example using a simple model of an empty room is shown in Section 4.1.4.

4.1.3 Octree Overhead

To simply the above discussion, so far I have ignored the amount of memory that is required for maintaining the octree structure. This overhead is now addressed as follows.

Let us look at how much memory is required for a single octree cell. Typically, an octree cell has the following items:

1. A pointer to its parent cell.
2. Eight pointers to its children cells.
3. Six numbers to represent the spatial boundaries of the cell. Six additional numbers may also be used to represent a tighter bounding box of actually occupied space.

In the above example, an octree cell consumes about 84 bytes of memory in the C programming language.

In contrast, a sample that is stored in an LDI contains at least:

1. The color, which is usually represented as the red, green, blue, and alpha components.
2. The depth.
3. The image coordinates.

This consumes about 7 to 12 bytes per sample, depending on whether byte, short integer, integer, or floating-point numbers are used for the depth and the image coordinates. Let us assume that it is 8 bytes per sample for the following discussion.

Now we want to estimate how much memory is used for storing the samples in the LDIs of an octree cell and compare that to the 84 bytes that we estimated for the octree cell itself. Remember that all LDIs in the LDI tree have the same image resolution (Section 3.1.1). However, the number of samples in each LDI is not fixed because some pixels may not have any samples, yet some other pixels may have multiple samples. Therefore we cannot find the exact amount of memory taken by the samples. But we can still see that overhead is relatively small if at least some pixels are occupied, because merely 11 samples consume more memory than the overhead of the octree cell.

In my experiments that are shown in Section 4.1.4, the overhead of the octree comprises less than 1% of the memory usage.

Note that although the image resolution for the LDIs in octree cells may be set arbitrarily, the above suggests that a very low resolution should be avoided since it will increase the overhead of the octree.

4.1.4 Experimental Results

To characterize the memory usage of LDI Trees, I use the experiments in this section to support the two upper bounds that I mentioned earlier in this chapter. The first experiment involves a simple scene of an empty room, which allows easy estimates of surface areas. The second experiment uses a Cathedral church model, which reveals the growth in memory usage when the number of reference images increases. The third experiment explores the relationship between the scene complexity and the memory usage, by replicating the objects in the scene.

Experiment of an Empty Room

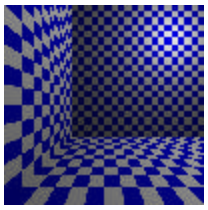


Figure 4-3: A reference image from the scene of an empty room.

The first scene is an empty room, which consists of six walls of checker-board pattern. I generate the reference images by ray tracing using the POVRay ray-tracer with my own extension to generate depth images [POVRay].

The width, height, and depth of the room are all 200 units. Each reference image has 100×100 pixels and a 90-degree field of view. They are taken from a minimal distance of 50 units to the closest wall. These parameters are chosen such that the maximal sampling rate of the surfaces is exactly one unit per pixel. Figure 4-3 shows one of the reference images.

I generated 80 reference images for this experiment. The first 48 images are taken from the eight vertices of the cube that represents the range of the allowable viewpoints within the room. Six images are taken at each vertices of the cube, each of which looks at one of the six walls. The other 32 images are taken at random locations.

The total area of surfaces is 200×200×6 = 240,000. As I discussed in Section 4.1.2, we can expect the maximal number of samples in its LDI Tree to be around 240,000×4/3 = 320,000. The extra one third is for the filtered samples of the LDI Tree.

The following chart shows the result of my experiment:

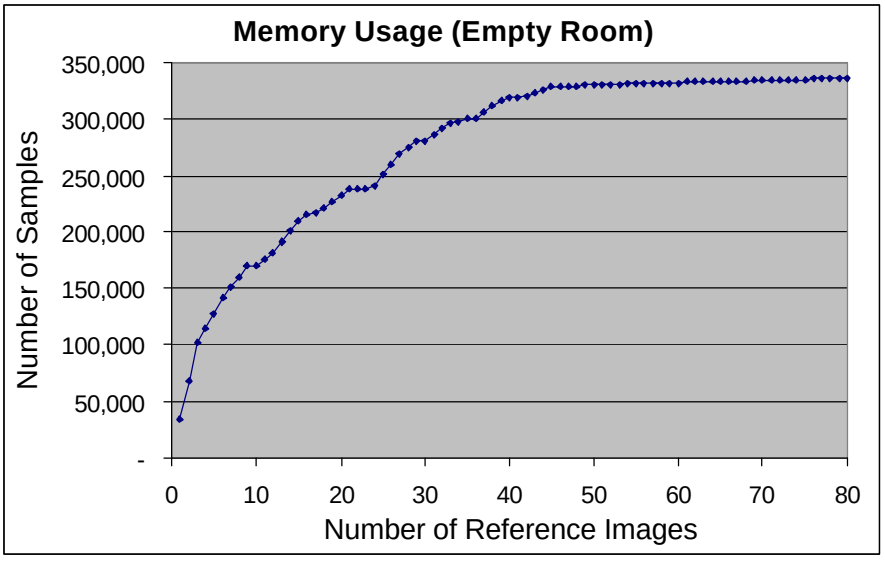


Figure 4-4: The memory usage for the scene of an empty room.

The actual numbers of samples when 80 reference images are used is 335,403. It is slightly larger than the 320,000 samples that we expected. The extra samples comes from the following:

- An extra LDI pixel may be needed for samples from the edges of walls.
- I assign each sample according to its normal vector to one of the 6 LDIs in the selected cell. The normal vectors are derived from the depth images according to the gradients between adjacent pixels. But this method cannot correctly handle special cases such as those samples from where two walls meet. Therefore some samples may be assigned to incorrect LDIs and cost some extra LDI pixels.

In my implementation, each sample uses 16 bytes. So the memory usage is about 5.12MB. Note that the LDI Tree has 73 octree cells and the overhead of maintaining the octree is merely 0.01MB, which is about 0.2% of the total memory usage.

Size of Root Cell

The LDI Tree built from those 80 reference images has LDIs of 64×64 image resolution. The size of the root cell is 256 units in each dimension. As discussed in Section 3.2.1, we may need to subdivide the root cell recursively, so that the LDIs in the octree cell have high enough sampling rates to store the samples from reference images. In this experiment, the height of the LDI Tree is 3 (i.e., two levels of subdivision). Note that if we use a different size for the root cell, the number of samples in the LDI Tree will increase because the sampling rates in the LDIs might not match exactly the sampling rates of the reference images. For example, if we use a slightly smaller root cell of 200 units in each dimension, the height of the LDI Tree is still 3 but the LDI in a leaf cell will have only 200/256 unit of distance between pixels. This means that more pixels will be used for resampling. If we use a slightly larger root cell instead, then the original leaf cell will have more than one unit of distance between pixels. Therefore, one more level of subdivision will occur, and the distance between pixels in new leaf cell will be only slightly larger than 0.5. This will result in even more LDI pixels being used for resampling.

In short, unless in a highly simplified scene such as the one in this experiment and with carefully chosen parameters, a reference image pixel usually requires more than only one LDI pixel to be stored in the LDI Tree.

Experiment on A Cathedral Model

The scene in the second experiment is a model of the interior of a cathedral, Palladio's Il Redentore in Venice [O'Brien93]. It better represents the scenes that we are more likely to encounter in practice. The reference images are generated by ray-tracing using the Rayshade program [Kolb94]. Each reference image has 512×512 pixels and a 90-degree field of view. The reference images are taken from up to 25 different positions, with four images taken at each position to complete a 360-degree view. Figure 4-5 shows one of the reference images.

Figure 4-5: A reference image from the scene of a cathedral model.

The following chart shows the growth of the memory size. Again the chart shows that the memory size is approaching an upper bound as the number of reference images increases, even though we cannot calculate the upper bound analytically. Note that the growth of memory size does not level off as dramatically as in the previous experiment of an empty room. That is expected because new details near each new viewpoint are still being added to the LDI Tree.

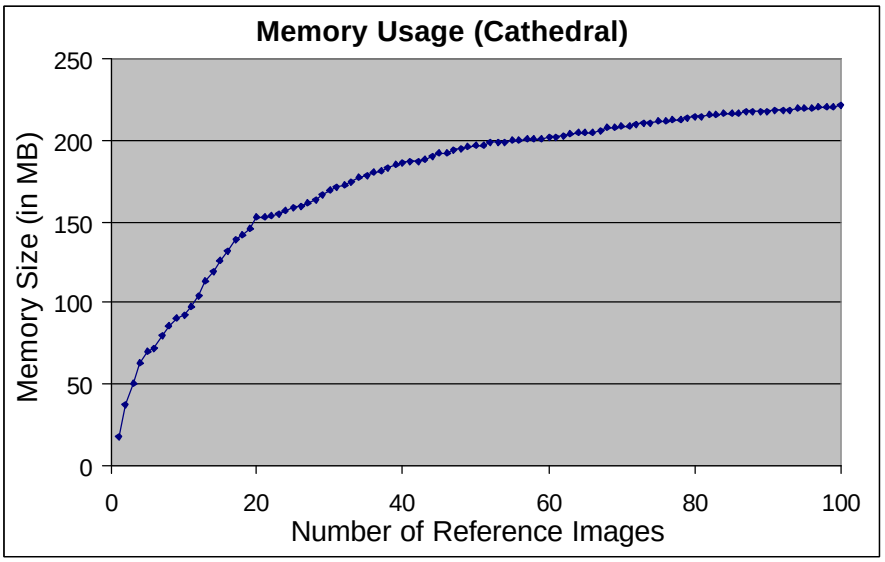


Figure 4-6: The memory usage for the scene of a cathedral model.

It is worth noting that the first reference image adds 1.2 million samples, or about 18 MB[1] of memory to the LDI Tree. If each of the 512×512 pixels in the first reference image resulted in exactly one unfiltered sample of the LDI Tree, we would have 512×512 unfiltered samples and about 512×512×0.33 filtered samples, i.e., about 0.35 million samples or 5.3 MB. Although the number in my experiment is about 3.4 times as large, it is indeed a reasonable number. The reason is that, in practice it is highly unlikely for a reference-image pixel to add only one unfiltered sample to the LDI Tree. I was able to achieve that in the experiment of an empty room because the parameters (especially the bounding box for the root cell) were carefully chosen such that the LDIs at some level had the same sampling rate as the reference-image pixels. Otherwise, a reference-image pixel is more likely to cover more than one pixel in the LDIs of octree cell.

Experiment with Tree Models

So far, I have focused on the growth of memory usage when the number of reference images increases. In the third experiment, I investigate how the scene

[1] Each sample uses 16 bytes.

complexity affects the memory usage. This is done by replicating the objects in the scene.

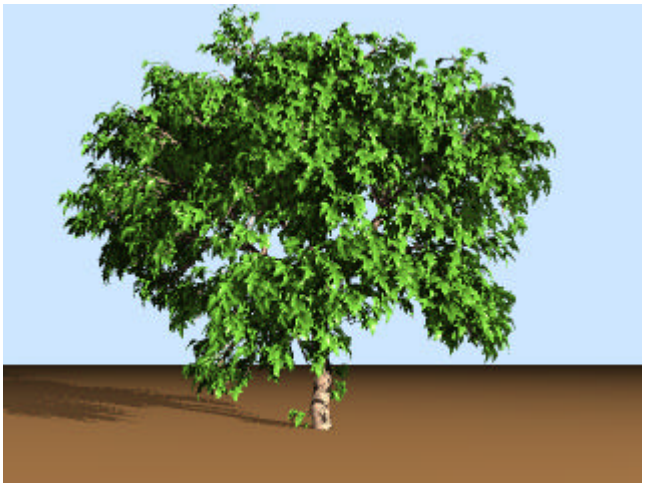


Figure 4-7: A reference image from the scene containing one tree.

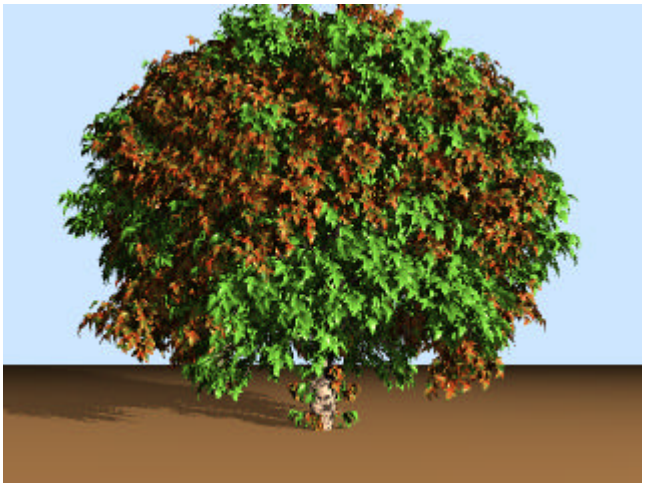


Figure 4-8: A reference image from the scene containing four trees that are replicated from the tree in Figure 4-7.

The scenes in this experiment contain tree models which are generated procedurally. The reference images are generated by ray tracing using the POVRay ray-tracer with my own extension to generate depth images [Povray00]. Each reference image has 640×480 pixels.

In the first part of this experiment, there is one tree in the scene. The tree contains 37,175 primitives[2], which include 24,948 spheres and cones for the branches and 12,227 triangle meshes for the leaves. I generate 32 reference images from positions that circle the tree. Figure 4-7 shows one of the reference images. In the second part of the experiment, the scene contains four trees that are replicated from the first tree. Two of the replicated trees also have different colors of leaves, but the geometry of the leaves remains the same. The three replicated trees are rotated 90, 180, and 270 degrees respectively so they look like a single tree but with four times as many branches and leaves. There are also 32 reference images taken from positions identical to those in the first part. Figure 4-8 shows a reference image of the replicated trees.

The LDI Tree built for the scene of one tree consumes 146.46 MB of memory. The memory usage increases to 197.17 MB for the scene of four trees. Although the scene of four trees contains almost four times as many objects as the scene of one tree, the memory usage only increases by 34.6%. Later in this chapter, we will also see how the scene complexity affects the rendering time.

It is not surprising that a more complex scene consumes more memory in the LDI Tree, because each new reference image is more likely to see previously occluded areas. A scene with more objects may also contain larger total surface areas, which means a higher upper bound for the memory usage as discussed in Section 4.1.2. This also implies that we may need to use more reference images to approach that upper bound. However if the number of reference images is fixed, this experiment shows that the scene complexity has only a limited impact on the memory usage.

4.1.5 Disk Storage and Prefetching

In my current implementation, the whole LDI Tree resides in the main memory. However the space partitioning of the LDI Tree makes it easy to store the whole or parts of LDI Tree on disks instead, if a larger dataset demands such a solution. Furthermore, prefetching is easy to implement for the rendering algorithm, because only a subset of the

[2] I do not know the exact number of triangles in the tree because the objects in POVRay may not be represented as triangles or polygons.

LDI Tree is needed at each frame and the subsequent frames are likely to access only the immediate children cells of that subset.

4.2 Rendering Time

To characterize the rendering time for generating an output image from the LDI Tree, I revisit two of the experiments that are described in Section 4.1.4. Those experiments show how the rendering time changes with the number of reference images and the scene complexity.

In image-based rendering, we would like the rendering time to be dependent on only the image size. 3D image warping achieves that goal when only one reference image is used, but leaves the disocclusion problem unsolved. In this work, I address the disocclusion problem by showing the following in the experiments:

- While multiple reference images are used to reduce the disocclusion artifacts, the rendering time approaches a limit as the number of reference images increases.
- The limit of the rendering time for a given scene is comparable to the size of output image. In fact, the number of splatting operations used to generate an output image is a small constant times the number of pixels in the output image.
- Even in the most difficult cases such as the one demonstrated in the experiment on tree models, the scene complexity only increases the rendering time by a relatively small fraction.

Currently, rendering output images from an LDI Tree is still not at interactive rates. The main reason is that the rendering algorithms are implemented in software only because the following are not well supported in currently available graphics hardware:

- The basic operation of the rendering is splatting.
- As discussed in Section 3.3, the splatting generates partially transparent pixels in unsorted order.

Therefore in addition to the actual wall-clock time, I also measure the cost of rendering in terms of number of basic operations, i.e., the number of splattings. Those numbers reflect the potential performance that may be achieved by new hardware designs. In Section 5.3, I will further discuss the hardware issues.

The rendering time for the experiments in this chapter is measured on a Silicon Graphics Onyx2 workstation with 300 MHz MIPS R12000 processors and 16 gigabytes of main memory.

4.2.1 Experimental Results

Experiment on A Cathedral Model

The first scene is the cathedral model, which is also used in Section 4.1.4. I rendered 40 output images along a typical walkthrough path. Each frame has 480×360 pixels. To investigate how the rendering time changes when the number of reference images increases, I repeat the rendering of the same walkthrough path when the number of reference images is 4, 12, 20, 36, and 100 respectively. The results are in the following chart.

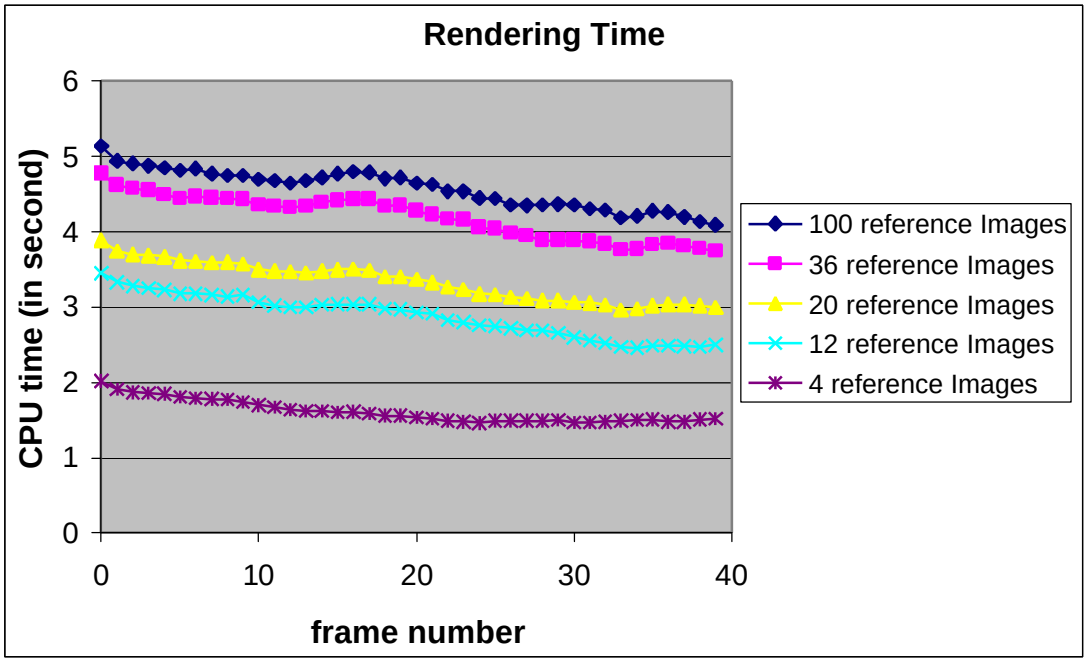


Figure 4-9: The rendering time for a walkthrough of the cathedral model.

To see the trend more clearly, the following chart shows the average rendering time versus the number of reference images.

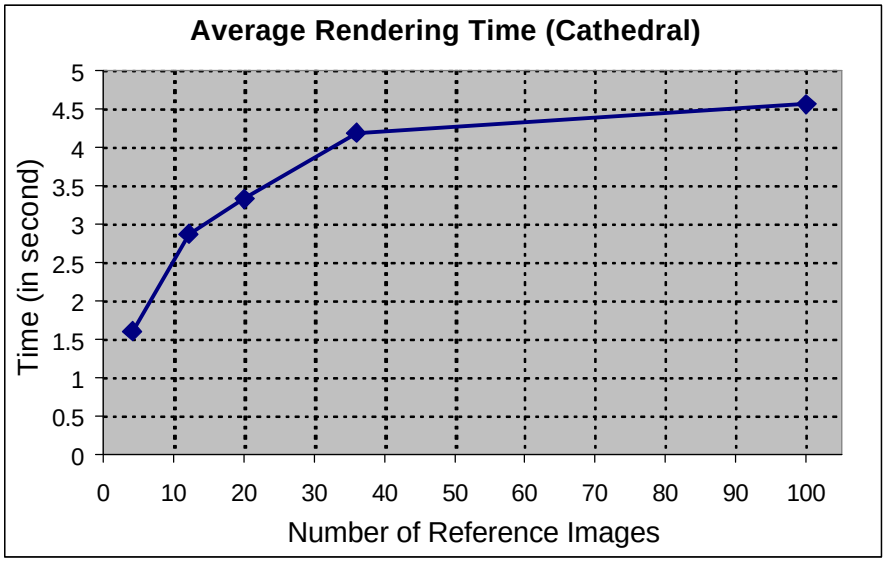


Figure 4-10: The average rendering time per output image versus the number of reference images for the scene of cathedral model.

The above charts show that the rendering time approaches a limit when the number of reference images increases. This is actually no surprise since we know from Section 4.1.4 that the memory usage and the number of samples approach an upper bound as well. So the rendering time could not be worse than the time required to process all samples.

However, we are still interested in the relationship between the rendering time and the number of pixels in an output image. The following chart shows the average number of samples that are accessed from the LDI Tree and drawn to the output image by splatting at each frame. The numbers are already divided by the number of pixels in an output image.

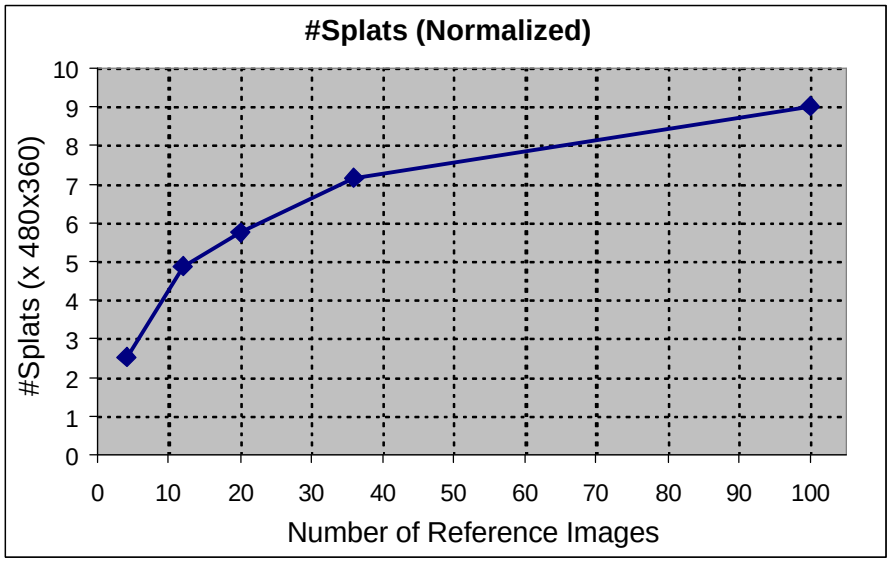


Figure 4-11: The average number of splatting operations per output image for the cathedral model versus the number of reference images. The numbers shown are already divided by the number of pixels (480×360) in an output image.

The above shows that the number of splats is within a small constant time of the number of pixels in an output image. Note that a fraction of that constant is actually caused by resampling the reference images in the LDIs of higher than necessary sampling rates. For example, when only four reference images taken from the same position are used, the relative number of splats is already 2.5. When 100 reference images are used, the relative number of splats increases to 9.2, which represents a ratio of 3.68.

Experiment with Tree Models

In the second experiment, I use the scenes of tree models (which are also used in Section 4.1.4) to investigate the impact of the scene complexity.

There are 36 viewpoints, which are placed evenly on a circle surrounding the trees for generating the output images. Each output image has 640×480 pixels. Figure 4-12 and Figure 4-13 show the average rendering times and average number of splats under difference numbers of reference images. The lower line in each chart is the result from the scene with only one tree. The upper line is from the scene with four replicated trees.

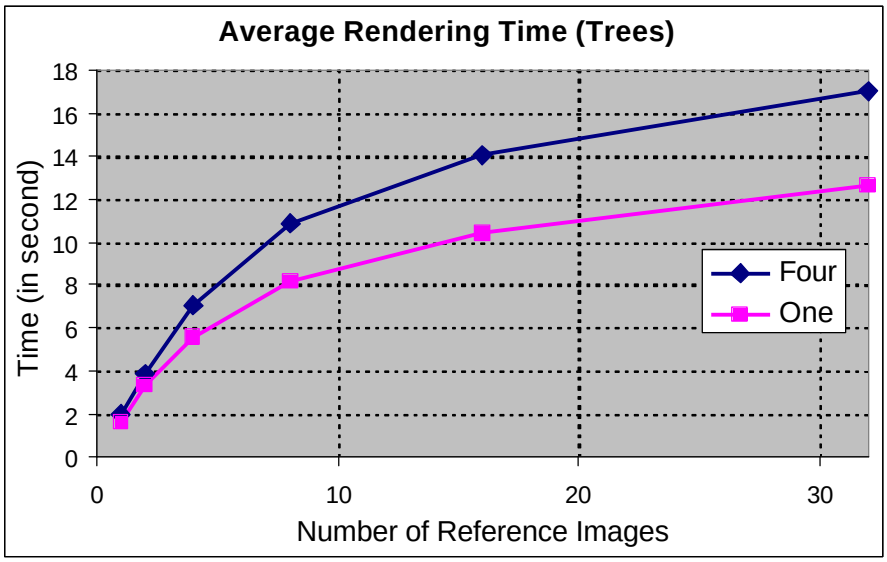


Figure 4-12: The average rendering time per output image for the tree models versus the number of reference images. The upper line is for the scene with four tress. The lower line is for the scene with one tree.

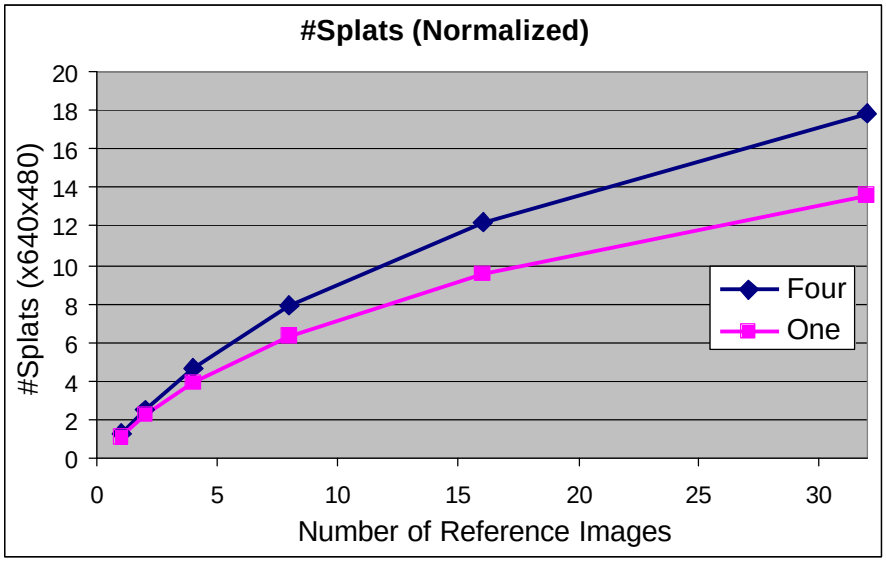


Figure 4-13: The average number of splatting operations per output image for the tree models versus the number of reference images. The upper line is for the scene with four tress. The lower line is for the scene with one tree. The numbers shown are already divided by the number of pixels (640×480) in an output image.

The charts show continuing growth in rendering time and the number of splats when more reference images are added. Because there are many occlusions between leaves, each reference image is more likely to contribute new information to the LDI Tree. However, our interest in this experiment is to see how the rendering time and the number of splats change with scene complexity.

For example, let us look at the rendering times and the numbers of splats when 32 reference images are used. In the scene with one tree, the rendering time is 12.65 seconds and the number of splats relative to the output image is 13.59. In the scene with four replicated trees, the rendering time increases to 17.03 seconds and the relative number of splats increases to 17.82 when the scene contains four replicated trees. The increases are 34.6% and 31.1% in rendering time and the number of splats respectively. Those increases are considered small since the number of objects in the scene has increased by four times.

4.3 Conclusions

Using the experiments in this chapter, I have demonstrated the following:

1. No matter how many reference images are added to the LDI Tree, its memory usage does not exceed an upper bound if we do not need infinite level of detail. Also, this upper bound is proportional to the total areas of surfaces.
2. The rendering time is linear to the number of pixels in the output images, instead of the number of reference images. Moreover, when measured in numbers of splats, the rendering cost is a small constant times the number of pixels in an output image.

The reason why the rendering time is proportional to the output image is that we are able to use the hierarchical representations to select the samples that cover about one pixel in the output image. However, each pixel in the output images may be covered by more than one sample because of the depth complexity. Another way to look at the depth complexity is by considering which octree cells are intersected by a ray connecting the viewer to a pixel in the output image. The surfaces in those octree cells may be warped to that specific pixel in the output image. When the number of reference images or the

scene complexity increases, the depth complexity also increases and results in slightly higher rendering cost. But keep in mind that the octree cells that are selected for rendering become larger when they are farther away from the viewer. Therefore the number of layers of octree cells along each viewing direction is still limited.

Another approach to address the issue of depth complexity is by using the techniques of occlusion culling, which I will discuss later in the next chapter.

CHAPTER 5 – CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

In this dissertation, I have demonstrated the following:

1. A novel data representation for merging reference images and removing their redundancy.
2. The algorithms to identify the sampling rates and levels of detail of reference images and to preserve them using hierarchical representations.
3. No matter how many reference images are added to a hierarchical representation, its memory usage does not exceed an upper bound when the level of detail is finite. This allows us to use any number of reference images to remove the disocclusion artifacts.
4. A rendering algorithm that selects from the hierarchical representations the samples that have the matching sampling rates with the output images.
5. A hole filling method that is based on the available three-dimensional information, instead of the two-dimensional information on the output images.
6. A compatible back-face culling algorithm.
7. The rendering cost is asymptotically of the same order as the size of the output images.

In the rest of this chapter, I discuss some other issues that are related but not essential to my work, and possible future work.

5.2 Occlusion Culling

Occlusion culling is a technique to reduce the rendering time by detecting the objects that will not appear on an output image before spending major efforts in rendering

those objects. It is sometimes referred as visibility culling. Unlike the Z-Buffer which determines whether a pixel is occluded, occlusion culling determines whether a surface or a set of surfaces within a bounding volume are occluded. Two of the representative works in occlusion culling for rendering polygon-based objects are the Hierarchical Z-Buffer [Greene93] and the Hierarchical Occlusion Maps [Zhang97].

I implement the Hierarchical Z-Buffer in the LDI Tree. It is impractical to perform occlusion culling for each sample because we need its screen coordinates for occlusion culling. But there is little saving in rendering time even if the sample is occluded because most of the effort in rendering has already been done in finding its screen coordinates. Therefore the occlusion culling is performed for each octree cell of the LDI Tree by the following steps:

1. Before an octree cell is warped to the output image, the 8 corners of the cell are warped to the output image to produce their positions and the depths in the coordinates of the output image.
2. A minimal bounding rectangle and a minimal depth for the above 8 corners are then defined in the coordinates of the output image.
3. The minimal bounding rectangle and the minimal depth for the octree cell is then compared to the hierarchical Z-Buffer to determine whether the cell is occluded.

However, we still have the following choices of how often the Hierarchical Z-Buffer is updated:

1. It may be updated whenever a sample is splatted to the output image.
2. It may be updated whenever an octree cell is rendered.
3. It may be updated only once per frame, either after a set of potential occluders are already rendered or by warping the depth buffer of the previous frame.

Obviously, the occlusion culling is more accurate if the Hierarchical Z-Buffer is updated more frequently. However the cost of updating the Hierarchical Z-Buffer is nontrivial, especially without hardware support. Since the occlusion culling is performed at the octree cell level, the first two choices should produce the same results.

In order to see how effective the occlusion culling is, I implement the second choice, which refreshes the Hierarchical Z-Buffer from the output buffer (described in Section 3.3.1) after each octree cell is rendered. Since most of the rendering time is now spent in refreshing the Hierarchical Z-Buffer, I will show only the numbers of splats in the following results. The numbers of splats tell us how effective the occlusion culling could be, if I implemented a better occlusion culling technique which requires fewer updates to the Hierarchical Z-Buffer.

5.2.1 A Scene Where Occlusion Culling Works Well

To create a scene that is ideal for occlusion culling, I take the model of an empty room that is used in Section 4.1.4 and replicate the room to create a scene of 4 rooms. The rooms are placed side by side along the Z-axis. There are 36 output images rendered in this experiment. The viewpoint is always at the center of the first room for all 36 output images but the viewing direction changes. The viewing direction is along the Z-axis at the beginning and is rotated by 10 degrees around the Y-axis in each subsequent frame.

Figure 5-1 shows the numbers of splats at each frame. The top line is when occlusion culling is disabled. The middle line shows that the results when the occlusion culling is performed. The bottom line serves as the absolute lower bound because it is obtained when there is only one room in the scene.

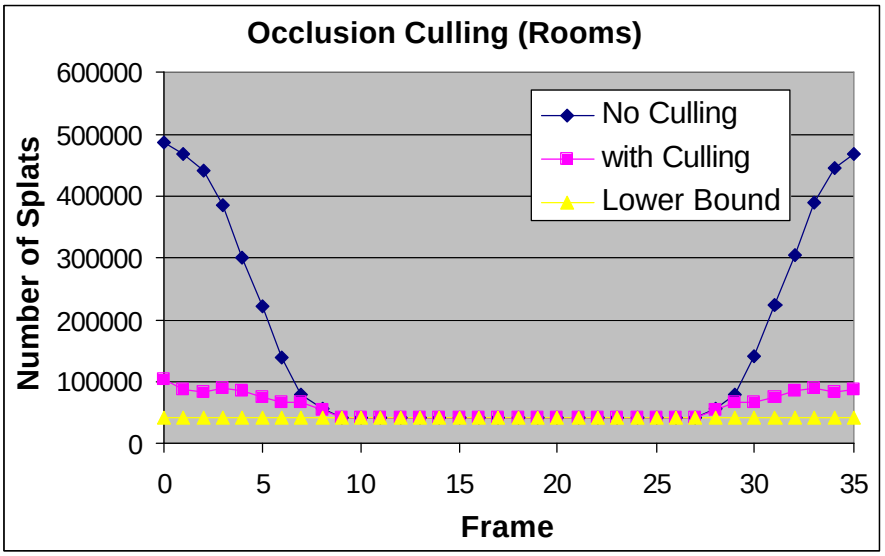


Figure 5-1: The performance of occlusion culling, measured in the number of splats at each frame, for the scene of empty rooms.

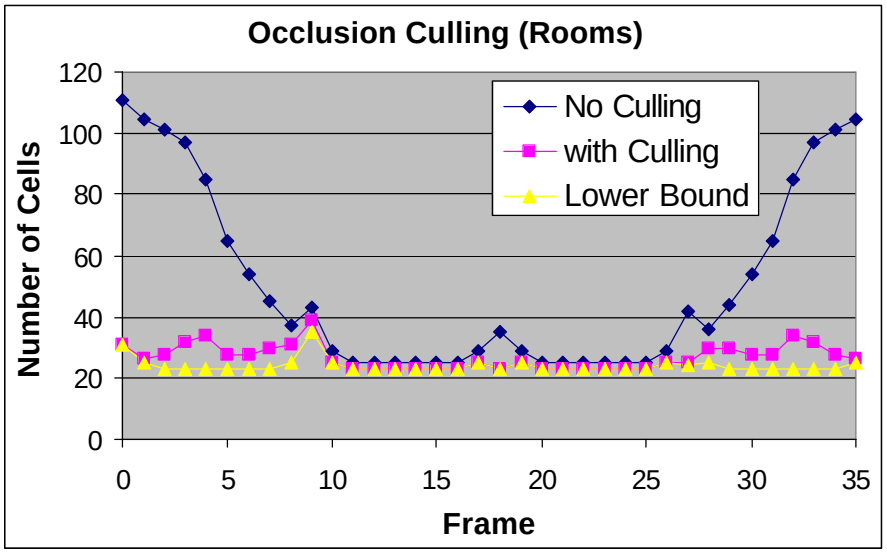


Figure 5-2: The performance of occlusion culling, measured in the number of rendered cells at each frame, for the scene of empty rooms.

The chart in Figure 5-1 shows that occlusion culling reduces the numbers of splats from about 10 times of the lower bound to about only twice of the lower bound. A reason why the numbers with culling do not reach the lower bound is because of those

rendered octree cells that contain samples from both the viewer's room and the next room. Another reason is that the bounding rectangle of the actual projected area of an octree cell on the screen is larger than the projected area of the cell. This sometimes causes an occluded cell to not be culled. Figure 5-2 shows the numbers of octree cells that are rendered instead of the numbers of splats. It shows that the occlusion culling is very effective in this scene.

5.2.2 A Scene Where Occlusion Culling Works Poorly

To see how the occlusion culling actually performs in scenes that cannot easily be divided into isolated spaces or rooms, I repeat the experiment of the cathedral model which is described in section 4.2.1, but with occlusion culling. The following chart shows the results in average number of splats with and without occlusion culling.

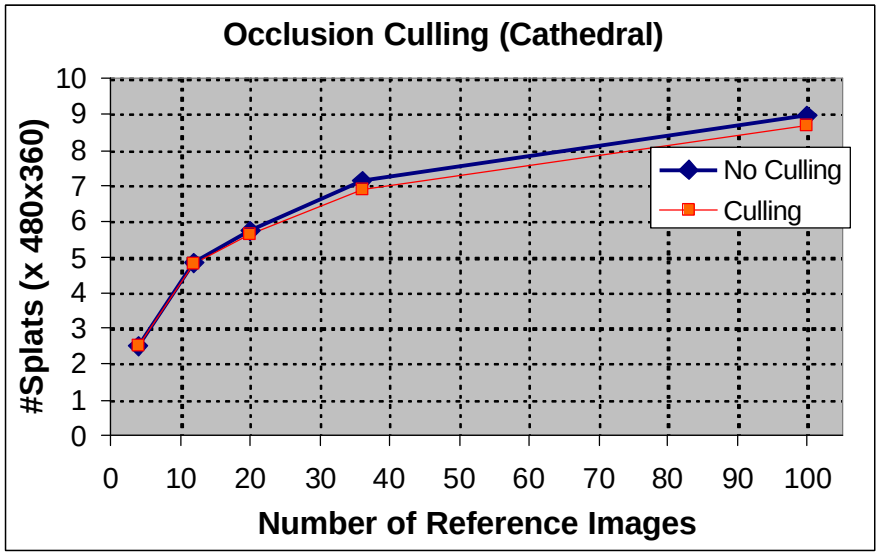


Figure 5-3: The performance of occlusion culling, measured in the number of splats at each frame, for the scene of cathedral model.

Using occlusion culling to reduce the depth complexity to one is actually a lofty goal in general. Although occlusion culling is not effective in scenes with large open spaces like the cathedral model, it does expand the possible applications of the LDI Tree to scenes such as the building models, where each room is individually modeled.

5.3 Hardware Issues

I chose the splatting method to reconstruct the surfaces from the samples that are warped to the output images because the splatting works well with the multi-resolution nature of the hierarchical representations. However, it does slow down the rendering because splatting is not well supported by the architecture of traditional graphics hardware. Also, it requires an A-Buffer-like output frame buffer to properly store, blend, and antialias the pixel fragments generated by the splatting.

Therefore the most desirable new features of graphics hardware to support the LDI Trees are the support of a faster splatting operation and the support of order-independent transparency. The support of faster splatting operations may also find other applications in volume rendering. Although the order-independent transparency is infeasible in theory, an approximation algorithm for hardware support of order-independent transparency has been proposed in [Jouppi99].

There are also variations of my rendering algorithms that could approximate the original algorithms on currently available graphics hardware, if slightly incorrect results are acceptable. The following are two of such variations:

1. Since the splat size is comparable to the pixel size of the output image, we may simply draw a point during each splatting operation. This could produce adequate output images for previewing.

2. Although the occlusion compatible order that is described in McMillan's 3D image warping does not exist between a parent and a child cell, we may simply ignore their order and process the samples in the children cells first. The reason is that the samples in the parent cell are at lower sampling rates and thus have less importance. If they are incorrectly occluded by the samples in the children cells, the errors are usually tolerable.

5.4 Placement of Camera Positions for Acquiring Reference Images

During the acquisition of reference images, we face the question of where the camera should be positioned in order to see any many surfaces as possible while keeping the number of reference images small. However, finding the optimal placement of

camera positions is a difficult problem, which is also discussed in [Stuerzlinger99] and [O'Rourke87]. Therefore, the alternative of placing the cameras in a regular pattern is often used instead.

The hierarchical representations I propose in this work remove the adverse impact on the memory usage and the rendering time when more reference images result from using a more regular or mechanical placement of camera positions. This makes the planning of the acquisition process simpler. However my work is compatible with any placement of camera positions, if the acquisition process still favors an approach of using fewer camera positions that are more strategically placed. One reason to favor such an approach may be that moving the camera to a different position requires significant re-calibration of equipment or other manual work.

5.5 LDI Tree as Image Cache

As mentioned earlier in Section 2.9, Image Caching is a technique to use image-based representations to speed up the rendering of polygonal objects. However, the technique is limited by the fact that the imposters that are used to represent a set of already rendered objects become invalid when the viewpoint is moved beyond a certain range.

If the LDI Tree is used as an image cache, then the imposters in each space partition are replaced by the LDIs. By using 3D Image Warping, the same LDIs may be used for any viewpoint. Therefore the need to regenerate the imposters is eliminated. To generate the initial image cache, we may use any previously rendered output images as the reference images and add them to the LDI Tree. Those output images may be recorded from a previous walkthrough of the scene.

Furthermore, we may divide the rendering task into the rendering of near objects and the rendering of far objects, which is similar to the concept in [Aliaga97]. This means that we may increase the minimal viewing distance of the surfaces that are stored in the LDI Tree. As discussed in Section 4.1.2, this will greatly reduce the memory usage of the LDI Tree.

However, the biggest issue is the slow rendering time of the LDI Tree because of the lack of hardware support. In the imposter-based image cache, the imposters may be generated quickly by the graphics hardware. It may be faster than using 3D image warping even though the latter is a more elegant solution.

5.6 Future Work

So far I have used only circular or square[1] footprints for splatting, regardless of surface normal or orientation information. However the footprints for splatting could be stretched in a certain direction that is determined by the surface orientation. This is in a way similar to the anisotropic filtering in texture mapping. Therefore a direction for future work is to incorporate the surface orientation into my framework for use in the splatting and the calculation of stamp size.

When a surface is sampled in multiple reference images, we should be able to get better sampling of the surface than what we can get from any single image. How to explore this type of cross-image supersampling is another direction of future work.

Like the original LDI, pixels that fall into the same pixel location and have similar depth values are merged together in an LDI Tree. That is based on the assumption that the surface is diffuse and little view-dependent variance can occur. However we may extend the cell representation (see Section 3.1) of an LDI Tree to store additional information about those merged pixels and to produce view-dependent shading effects during rendering. How to extract view-dependent properties of the surface from those merged pixels (or even from other nearby pixels) is yet another direction for future work.

[1] Depending on the distance metric. The footprint is circular if Euclidian distances are used, or square if Manhattan distances are used.

APPENDIX A

DERIVATION OF 3D IMAGE WARPING EQUATION

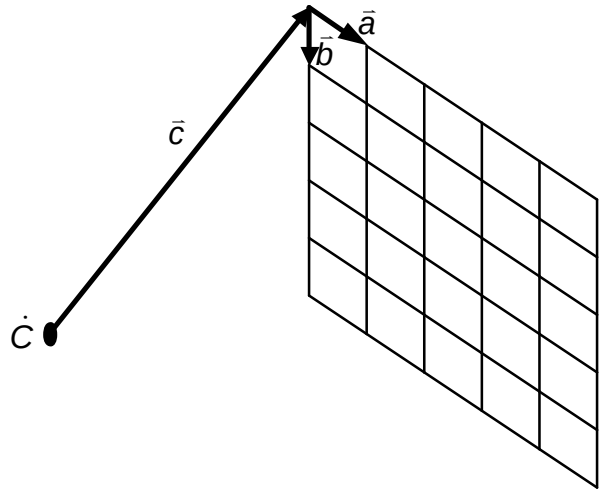


Figure A-1: The input image and the camera model.

Given an image and the camera parameters of that image, 3D image warping can generate images for the new viewpoints by moving the pixels of the image to new pixel locations on the output images.

Figure A-1 describes a planar input image and the associated camera parameters using the pinhole camera model.

I borrow the notation used in [McMillan97] and [Mark99]. Each pixel at the image coordinates (x, y) contains a color values $color(x, y)$ and a disparity value $d(x, y)$. The disparity value is closely related to the depth information. In fact we may obtain the depth d by:

$$d = f / d$$

$$f = \vec{c} \cdot \frac{(\vec{a} \times \vec{b})}{\left\| (\vec{a} \times \vec{b}) \right\|} \qquad \text{Eq.A-1}$$

where f represents the distance from the center of projection to the projection plane.

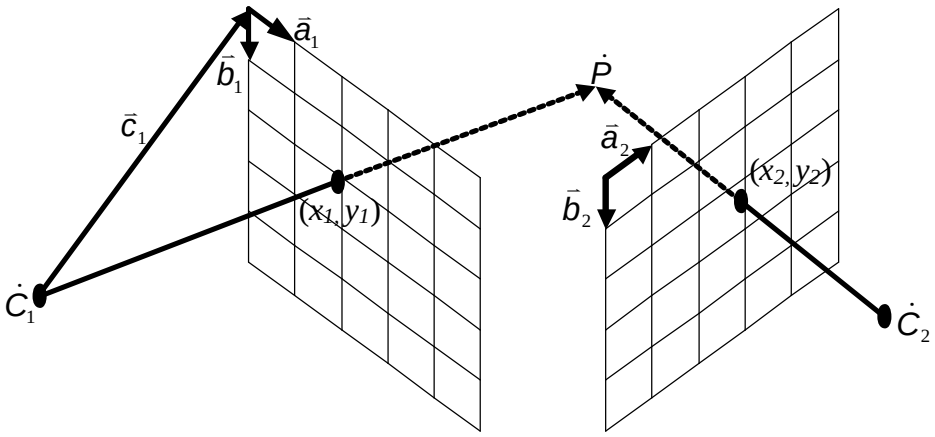


Figure A-2: A point that is represented by a pixel of the input image is re-projected to a new view.

The camera parameters and the disparity values allow us to project a pixel to a 3D location in the scene, i.e., the sample point from the surface or object that the pixel is representing. This may be expressed as the following equation:

$$\dot{P} = \dot{C} + (\vec{a}x + \vec{b}y + \vec{c})/d \qquad \text{Eq.A-2}$$

Or the following equation if expressed in the matrix form:

$$\dot{P} = \dot{C} + \begin{bmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ a_z & b_z & c_z \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \frac{1}{d} \qquad \text{Eq.A-3}$$

Figure A-2 shows that point and the second camera for the new viewpoint. To find out the image coordinates on the second image for that pixel, we can simply inversely project the pixel to the image plane of the second camera. That leads to the following equation:

$$\dot{P} = \dot{C}_1 + \begin{bmatrix} a_{1x} & b_{1x} & c_{1x} \\ a_{1y} & b_{1y} & c_{1y} \\ a_{1z} & b_{1z} & c_{1z} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \frac{1}{d_1} = \dot{C}_2 + \begin{bmatrix} a_{2x} & b_{2x} & c_{2x} \\ a_{2y} & b_{2y} & c_{2y} \\ a_{2z} & b_{2z} & c_{2z} \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} \frac{1}{d_2} \qquad \text{Eq.A-4}$$

Therefore,

$$\begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} \frac{1}{d_2} = \begin{bmatrix} a_{2x} & b_{2x} & c_{2x} \\ a_{2y} & b_{2y} & c_{2y} \\ a_{2z} & b_{2z} & c_{2z} \end{bmatrix}^{-1} (\dot{C}_1 - \dot{C}_2) + \begin{bmatrix} a_{2x} & b_{2x} & c_{2x} \\ a_{2y} & b_{2y} & c_{2y} \\ a_{2z} & b_{2z} & c_{2z} \end{bmatrix}^{-1} \begin{bmatrix} a_{1x} & b_{1x} & c_{1x} \\ a_{1y} & b_{1y} & c_{1y} \\ a_{1z} & b_{1z} & c_{1z} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \frac{1}{d_1}$$

Eq.A-5

Because $\dot{C}_1$, $\dot{C}_2$ and the camera parameters are fixed, the above may be rewritten as the following form:

$$\begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} \frac{d_1}{d_2} = \begin{bmatrix} k_1 x_1 + k_2 y_1 + k_3 + k_4 d_1 \\ k_5 x_1 + k_6 y_1 + k_7 + k_8 d_1 \\ k_9 x_1 + k_{10} y_1 + k_{11} + k_{12} d_1 \end{bmatrix} \quad \text{Eq.A-6}$$

where k_1 through k_{12} are defined as:

$$\begin{bmatrix} k_4 \\ k_8 \\ k_{12} \end{bmatrix} = \begin{bmatrix} a_{2x} & b_{2x} & c_{2x} \\ a_{2y} & b_{2y} & c_{2y} \\ a_{2z} & b_{2z} & c_{2z} \end{bmatrix}^{-1} (\dot{C}_1 - \dot{C}_2) \quad \text{Eq.A-7}$$

$$\begin{bmatrix} k_1 & k_2 & k_2 \\ k_5 & k_6 & k_7 \\ k_9 & k_{10} & k_{11} \end{bmatrix} = \begin{bmatrix} a_{2x} & b_{2x} & c_{2x} \\ a_{2y} & b_{2y} & c_{2y} \\ a_{2z} & b_{2z} & c_{2z} \end{bmatrix}^{-1} \begin{bmatrix} a_{1x} & b_{1x} & c_{1x} \\ a_{1y} & b_{1y} & c_{1y} \\ a_{1z} & b_{1z} & c_{1z} \end{bmatrix} \quad \text{Eq.A-8}$$

That leads to the familiar 3D image warping equation:

$$(x_2, y_2) = \left(\frac{k_1 x_1 + k_2 y_1 + k_3 + k_4 d_1}{k_9 x_1 + k_{10} y_1 + k_{11} + k_{12} d_1}, \frac{k_5 x_1 + k_6 y_1 + k_7 + k_8 d_1}{k_9 x_1 + k_{10} y_1 + k_{11} + k_{12} d_1}\right) \quad \text{Eq.A-9}$$

The above equation is evaluated once for each pixel of the input images. Therefore the cost of rendering depends only on the size or the total number of pixels of the input images. This distinguishes the image-based rendering from the traditional polygon-based rendering where the cost of rendering depends on the complexity of the scenes or the number of polygons.

APPENDIX B

EPIPOLAR GEOMETRY

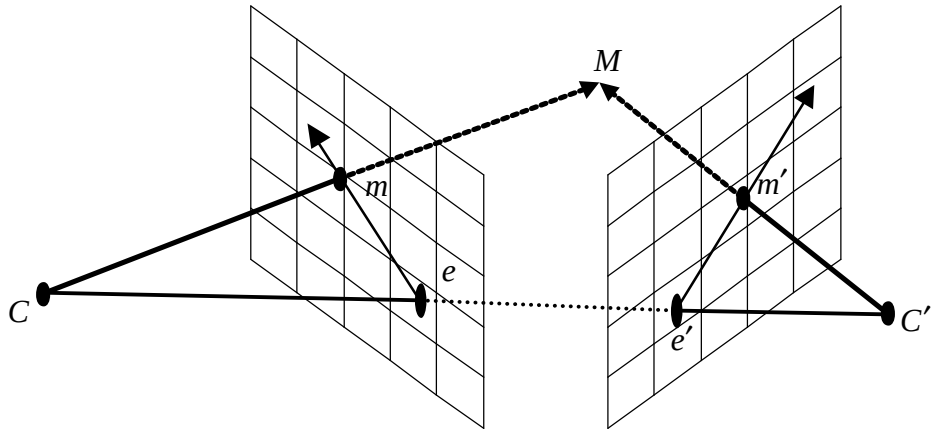


Figure B-1: The epipolar geometry between two images.

Many properties of 3D Image Warping are derived from the epipolar geometry between two images. Here I briefly discuss two of the important properties that are closely related to this dissertation. They are:

1. A pixel on one of the images is corresponding to a line on the other image.
2. The existence of the occlusion compatible order, i.e., if the pixels from one image is warped to the other image in a certain order, then the pixels arrive in back-to-front order if they are warped to the same pixels on the second image.

Figure B-1 shows a pair of images whose camera positions are at C and C'. M is a point in object space which projects to m and m' on the two images. e and e' mark the points where the camera position from one image is projected to a point on the other image.

Let us look at the pixel m on the left image. Assume that the pixel coordinates is fixed but we do not know the disparity of the pixel. This means that M could be anywhere on the line passing through C and m. Although we cannot compute the pixel

coordinates of m' without knowing the disparity, we still know that m' falls on the line where the plane formed by m, C and C' intersects with the right image. This results in the first property listed above.

Conversely, if we want to search for the pixels on the right image that could fall on the pixel coordinates of m on the left image, we need not look further beyond a line on the right image. This property is used in the inverse warping which is described in Section 2.5.

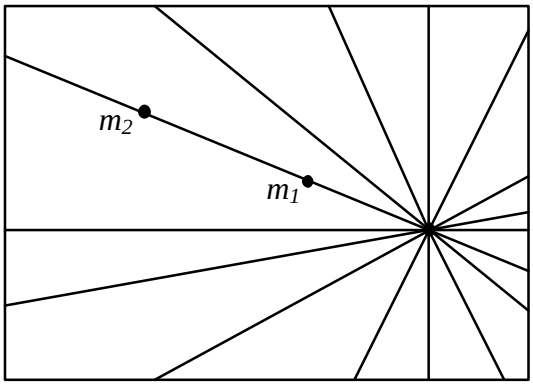


Figure B-2: The pixels from one image are mapped to lines passing through the epipole on the other image.

When we consider many different pixels on the right image, they correspond to different lines on the left image. What is interesting is that all those lines pass through the pixel e, which is also called epipole. It is where the line connecting two camera positions intersects the image. Figure B-2 depicts those lines and the epipole.

The epipole is important in defining the occlusion compatible order, which is the second property listed above. In fact we can derive the occlusion compatible order from the first property. For example, if Figure B-2 depicts the left image and the line passing through m_1 and m_2 corresponds to the pixel m' on the right image, then any pixel on that line could potentially be warped to m'. Assume that m_1 and m_2 happen to be two of those pixels. From Figure B-1, we can see that m_1 represents a point in object space that is closer to C', the camera position of right image, than m_2. Therefore if we can

somehow guarantee that m_1 is warped to the right image after m_2, then they are warped in the occlusion compatible order. This occlusion compatible order is clearly observed if we traverse the pixels along the lines toward the epipole. If we need to traverse the pixels in scanline (or horizontal) directions, then we can divide the image into four rectangles based on the location of the epipole and traverse the pixels in the directions that are approaching the epipole.

Note that the above is not meant to formal proofs of the mentioned properties. I only discussed the simple cases where the epipoles are visible on the images. For more complete discussion of the epipolar geometry, please refer to [Faugeras93].

BIBLIOGRAPHY

[Adelson91] Edward H. Adelson and James R. Bergen. "The Plenoptic Function and the Elements of Early Vision". *Computational Models of Visual Processing*, Chapter 1, Edited by Michael Landy and J. Anthony Movshon. MIT Press, Cambridge, Mass. 1991.

[Aliaga97] Daniel G. Aliaga and Anselmo A. Lastra. "Architectural Walkthroughs Using Portal Textures". In *Proceedings of IEEE Visualization 97*, pages 355–362, October 1997.

[Carpenter84] Loren Carpenter. "The A-buffer, an Antialiased Hidden Surface Method". In *Computer Graphics (SIGGRAPH 84 Conference Proceedings)*, volume 18, pages 103–108, July 1984.

[Chang99] Chun-Fa Chang, Gary Bishop and Anselmo Lastra. "LDI Tree: A Hierarchical Representation for Image-Based Rendering". In *SIGGRAPH 1999 Conference Proceedings*, pages 291–298, August 1999.

[Chien88] C. H. Chien, Y. B. Sim and J. K. Aggarwal. "Generation of Volume/Surface Octree from Range Data". *The Computer Society Conference on Computer Vision and Pattern Recognition*, pages 254–60, June 1988.

[Cohen96] Jonathon Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Agarwal, Frederick Brooks and William Wright. "Simplification Envelopes". In *SIGGRAPH 1996 Conference Proceedings*, pages 119–128, August 1996.

[Connolly84] C. I. Connolly. "Cumulative Generation of Octree Models from Range Data". In *Proceedings, Intl' Conf. Robotics*, pages 25–32, March 1984.

[Curless96] Brian Curless and Marc Levoy. "A Volumetric Method for Building Complex Models from Range Images". In *SIGGRAPH 96 Conference Proceedings*, pages 303–312, August 1996.

[Erikson00] Carl Erikson. *Hierarchical Level of Detail to Accelerate the Rendering of Large Static and Dynamic Polygonal Environments.* Ph.D. Dissertation. University of North Carolina at Chapel Hill, Department of Computer Science, 2000.

[Faugeras93] Olivier Faugeras. *Three-dimensional computer vision: a geometric viewpoint*. MIT Press, 1993.

[Fuchs80] Henry Fuchs, Zvi M. Kedem, and Bruce Naylor. "On Visible Surface Generation by A Priori Tree Structures". In *Computer Graphics (SIGGRAPH 80 Conference Proceedings)*, volume 14, pages 124–133, June 1980.

[Gortler96] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski and Michael F. Cohen. "The Lumigraph". In *SIGGRAPH 96 Conference Proceedings*, pages 43–54, August 1996.

[Greene93] Ned Greene, Michael Kass and Gavin Miller. "Hierarchical Z-Buffer Visibility". In *SIGGRAPH 93 Conference Proceedings*, pages 231–238, August 1993.

[Jouppi99] Norman P. Jouppi and Chun-Fa Chang. "Z^3: An Economical Hardware Technique for High-Quality Antialiasing and Order-Independent Transparency". In *Proceedings of 1999 Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 85–93.

[Kolb94] Craig Kolb. *Rayshade.* http://www-graphics.stanford.edu/~cek/rayshade/.

[Laur91] David Laur and Pat Hanrahan. "Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering". *Computer Graphics (SIGGRAPH 91 Conference Proceedings)*, volume 25, pages 285–288, July 1991.

[Lee98] Aaron W. F. Lee, Wim Sweldens, Peter Schröder, Lawrence Cowsar, and David Dobkin. "MAPS: Multiresolution Adaptive Parameterization of Surfaces". In *SIGGRAPH 98 Conference Proceedings*, pages 95–104, July 1998.

[Levoy85] Marc Levoy and Turner Whitted. "The Use of points as a Display Primitive". Technical Report 85-022, University of North Carolina at Chapel Hill, Department of Computer Science, 1995.

[Levoy96] Marc Levoy and Pat Hanrahan. "Light Field Rendering". In *SIGGRAPH 96 Conference Proceedings*, pages 31–42, August 1996.

[Li94] A. Li and G. Crebbin. "Octree Encoding of Objects from Range Images". *Pattern Recognition*, 27(5):727–739, May 1994.

[Lischinski98] Dani Lischinski and Ari Rappoport. "Image-Based Rendering for Non-Diffuse Synthetic Scenes". *Rendering Techniques '98 (Proc. 9th Eurographics Workshop on Rendering)*, June 29–July 1, 1998.

[Luebke97] David Luebke and Carl Erikson. "View-Dependent Simplification of Arbitrary Polygonal Environments". In *SIGGRAPH 97 Conference Proceedings*, pages 199–208, August 1997.

[Marcato98] Robert W. Marcato Jr. *Optimizing an Inverse Warper*. Master's of Engineering Thesis, Massachusetts Institute of Technology, 1998.

[Mark97] William R. Mark, Leonard McMillan and Gary Bishop. "Post-Rendering 3D Warping". In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, pages 7–16.

[Mark99] William R. Mark. *Post-Rendering 3D Image Warping: Visibility, Reconstruction, and Performance for Depth-Image Warping*. Ph.D. Dissertation. Technical Report 99-022, University of North Carolina at Chapel Hill, Department of Computer Science, 1999.

[Max96] Nelson Max. "Hierarchical Rendering of Trees from Precomputed Multi-Layer Z-Buffers". *Rendering Techniques '96 (Proc. 7th Eurographics Workshop on Rendering)*, pages 165–174, June 1996.

[McCormack99] Joel McCormack, Ronald Perry, Keith I. Farkas and Norman P. Jouppi. "Feline: Fast Elliptical Lines for Anisotropic Texture Mapping". In *SIGGRAPH 99 Conference Proceedings*, pages 243–250, August 1999.

[McMillan95a] Leonard McMillan. "A List-Priority Rendering Algorithm for Redisplaying Projected Surfaces". Technical Report 95-005, University of North Carolina at Chapel Hill, Department of Computer Science, 1995.

[McMillan95b] Leonard McMillan and Gary Bishop. "Plenoptic Modeling: An image-based rendering system". In *SIGGRAPH 95 Conference Proceedings*, pages 39–46, August 1995.

[McMillan97] Leonard McMillan. *An Image-Based Approach to Three-Dimensional Computer Graphics*. Ph.D. Dissertation. Technical Report 97-013, University of North Carolina at Chapel Hill, Department of Computer Science, 1997.

[Nyland99] Larss Nyland, David McAllister, Voicu Popescu, Chris McCue, and Anselmo Lastra. "Interactive exploration of acquired 3D data". In *Proceedings of the SPIE Applied Image and Pattern Recognition Conference (AIPR99)*, Washington, DC, October 13–15, 1999.

[O'Brien93] Nathan O'Brien. Rayshade - Il Redentore. http://www.fbe.unsw.edu.-au/exhibits/rayshade/church/

[O'Rourke87] J. O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, New York, 1987.

[POVRay] The Persistence of Vision Raytracer. http://www.povray.org/

[Rademacher98] Paul Rademacher and Gary Bishop. "Multiple-Center-of-Projection Images". In *SIGGRAPH 98 Conference Proceedings*, pages 199–206, July 1998.

[Schaufler96] Gernot Schaufler and Wolfgang Stürzlinger. "A Three-Dimensional Image Cache for Virtual Reality". In *Computer Graphics Forum*, 15(3), pages 227–236, Blackwell Publishers, August 1996.

[Schaufler98] Gernot Schaufler. "Per-Object Image Warping with Layered Impostors". In *Rendering Techniques '98 (Proc. 9th Eurographics Workshop on Rendering)*, pages 145–156, June 29–July 1, 1998.

[Schaufler99] Gernot Schaufler and Markus Priglinger. "Efficient Displacement Mapping by Image Warping". In *Rendering Techniques '99 (Proc. 10th Eurographics Workshop on Rendering)*, pages 175–186, June 1999.

[Shade96] Jonathan Shade, Dani Lischinski, David H. Salesin, Tony DeRose and John Snyder. "Hierarchical Image Caching for Accelerated Walkthrough of Complex Environments". In *SIGGRAPH 96 Conference Proceedings*, pages 75–82, August 1996.

[Shade98] Jonathan Shade, Steven Gortler, Li-wei He and Richard Szeliski. "Layered Depth Images". In *SIGGRAPH 98 Conference Proceedings*, pages 231–242, July 1998.

[Stuerzlinger99] W. Stuerzlinger. "Imaging all Visible Surfaces". In *Graphics Interface Proceedings 1999*, pages 115–122, June 1999.

[Westover91] Lee Westover. *SPLATTING: A Parallel, Feed-Forward Volume Rendering Algorithm.* Ph.D. Dissertation. Technical Report 91-029, University of North Carolina at Chapel Hill. 1991.

[Williams83] Lance Williams. "Pyramidal Parametrics". In *Computer Graphics (SIGGRAPH 83 Conference Proceedings)*, volume 17, pages 1–11, Detroit, MI, July 25–29, 1983.

[Zhang97] Hansong Zhang, Dinesh Manocha, Tom Hudson and Kenneth E. Hoff III. "Visibility Culling using Hierarchical Occlusion Maps". In *SIGGRAPH 1997 Conference Proceedings*, pages 77–88, August 1997.