INVESTIGATION OF LAYERED DEPTH IMAGES

By

CHARLES RANDALL BERRY

Bachelor of Science
University of Southern Colorado
Pueblo, Colorado
1983

Bachelor of Science
University of Central Oklahoma
Edmond, Oklahoma
1990

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTERS OF SCIENCE
August, 2001

INVESTIGATION OF LAYERED

DEPTH IMAGES

Thesis Approved:

[illegible]

Thesis Advisor

Blayne E. Mayfield

[illegible]

[illegible]

Dean of the Graduate College

PREFACE

This study was conducted to further the work being done in image based rendering (IBR) and in particular layered depth images (LDI). Through my research I have discovered some of the features of LDIs. LDIs are capable of generating synthetic images at several frames per second. It is also possible to produce images that are a close approximation to an actual image through the use of LDIs. There are some limitations to the use of LDIs. For instance, if the desired viewing angle is far from the initial camera there is a larger error. There is also a small problem with the Gaussian kernel. It allows some of the items in the back to bleed through to the front.

I would like to thank my masters committee–Drs. Douglas Heisterkamp, Nophill Park, George Hedrick, and Blayne Mayfield.– for their guidance and support in the completion of this research. I would like to extend a sincere thanks to Dr. Heisterkamp. Without his support I would have not been able to complete this research. I would also like to thank my wife and sons for their support and understanding during these past years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

A current concern in the field of computer graphics is the generation of synthetic images within a short period of time. Ideally synthetic images would be generated fast enough to be able to create a smooth continuous image stream. There are a number of ways to generate synthetic images. Synthetic images can be generated by changing the viewing position, light intensity, or light location. Synthetic images can also be generated by a combination of any of the three view position, light intensity, or light position. There are numerous applications for a method that can generate novel scenes at several frames per second. Some examples are interactive walk through of a scene, virtual reality, and computer games.

1.1 Three Dimensional Modeling

There have been several methods presented that generate synthetic images by varying the view positions [2, 20, 10, 14, 5, 22, 6, 16, 17, 19, 17, 8, 9]. One method that is common is three-dimensional modeling. This method of generation includes ray tracing. This method requires the longest time to render a scene. All of the elements of the scene are stored as drawing or rendering instructions in the computer. When a new view is desired the entire scene has to be regenerated. While this does not present much of a problem for simple scenes, it does pose a real problem for complex scenes. A scene that is composed of a few hundred polygons can be rendered in a short time. A scene that has several thousand or even millions of polygons can take a considerable length of time to render.

1.2 Image Based Rendering

On the other end of the spectrum is a method called *Image Based Rendering* (IBR). As the name suggests, this method uses images as the drawing primitives instead of geometric objects. One IBR technique is view morphing [18]. This paper describes a method of generating a synthetic image given two images. It is interesting to note that the two source images do not have to be of the same object. In one of the example image sequences the view morphing is done between two different people. A degenerate case of IBR is a texture map. This is probably the simplest case of IBR. An image is warped directly onto an object. For example, if you wanted to display a wooden crate, you could render a simple cube and then warp an image of a crate side onto the cube side. Another example would be displaying a building. Most buildings can be represented with a few simple cubes and then have an image warped onto the cubes. This reduces the computational complexity considerably and saves considerable computation time.

1.3 Layered Depth Images

One image based rendering method that has been recently introduced is the layered depth image [19, 6]. A layered depth image is composed of pixels with depth. This is to say that the pixel contains the usual color and *alpha* values as well as depth information. An alpha value can be thought of as a transparency value. A degenerate case of layered depth image is the sprite with depth. In this case there is a limit of one pixel per location. In an LDI

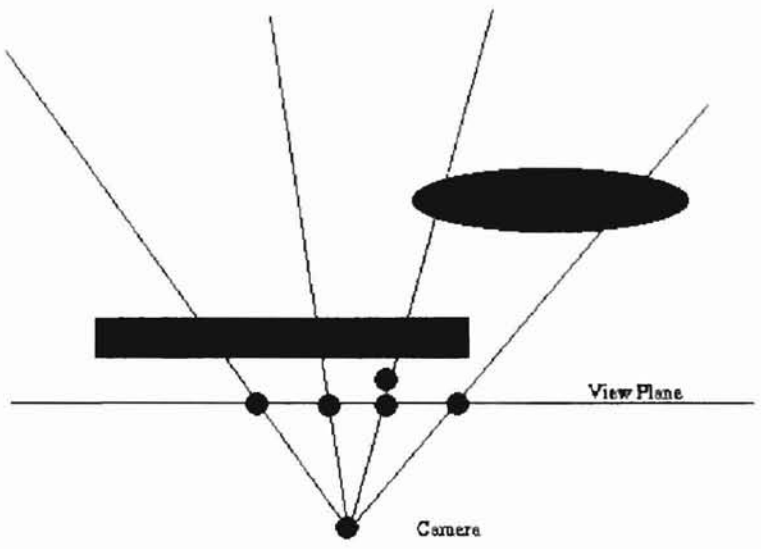

Figure 1.1: Layered Depth Image

there can be multiple pixels per pixel location. Figure 1.1 shows the details of an LDI. To construct an LDI you can think of rays emanating from the camera. As each ray strikes a surface, another pixel is added to the list for that image location. Layered depth images can be generated from rendered images or real images. To redisplay a layered depth image, McMillan's warp ordering algorithm [11] is used.

1.4 Combining Three Dimensional Modeling, Image Based Rendering, and Layered Depth Images

A complete image may contain the elements of several of the methods. There may be an environment map that shows the elements of the scene that are not apt to change. The environemt map is also the furthest from the camera. The next element of the scene is the sprite. After this comes the sprite with depth. The layered depth image comes next. Finally there are the geometrically rendered objects that are closest to the camera.

1.5 Varying Lighting Conditions

There have been several methods introduced to generate synthetic views by varying the lighting conditions [3, 4, 7, 13, 23, 24]. Again we go back to three-dimensional modeling. A change in lighting requires a complete regeneration of the scene. For simple scenes this does not present much of a problem. For complex scenes a considerable length of time can be required to generate a new image. The other end of the spectrum is the use of image based rendering. One method of modifying the lighting conditions is through the use of radiance map. A new image can be generated in a short period of time with the use of a radiance map. Images can be generated at several frames per second using this method, even for complex scenes. A radiance map contains lighting modification information for each element in the scene or pixel.

There are a few different ways to generate radiance maps [4, 23, 24]. Radiance maps are generated from real images or previously generated images. The essence of a radiance map is the capturing of the reflective properties of the items in the scene. With the reflectance properties known, a change in the light conditions can be accurately represented in a synthetic image. Most algorithms concentrate on the Lambertian model which accounts for the diffuse reflection and ignores the specular reflection. It is much easier to assume that the light energy reflected back is uniform. If you try to account for differences in reflection the rendering becomes much more time consuming. In Figure 1.2 the two types of reflection are shown. The diffuse part is represented by an arc at the point where the light ray strikes the surface. The specular reflection is represented by the ray leaving the surface. It is much easier to ignore the specular reflection. That way you do not have to calculate the angles

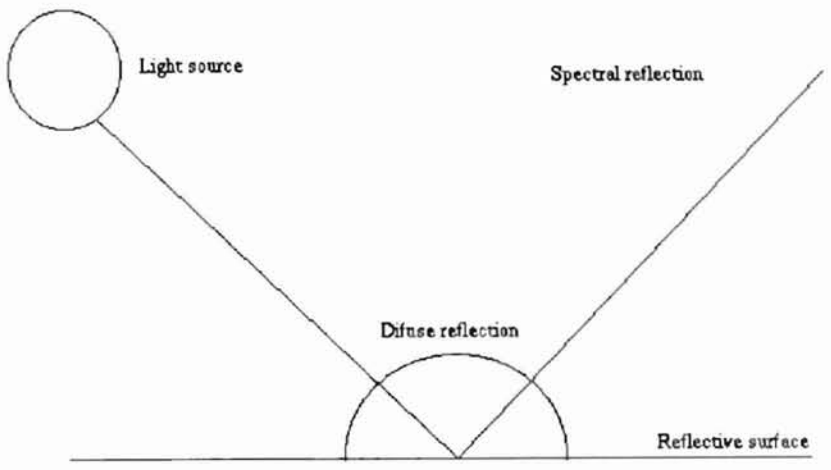

Figure 1.2: Reflection model

associated with the reflection.

Chapter 2

THESIS STATEMENT

Layered depth images are an effective way of generating synthetic images. Images are generated that are visually close to the actual image and are generated at several frames per second.

Hypothesis 0 — Layered depth images can be used to efficiently generate a synthetic image for a change in view that is within the prescribed camera motion range. This synthetic image will be within an error, ε, of the actual image.

Hypothesis 1 — Layered depth images can be used to generate images at a rate greater than, α, frames per second on a desktop PC.

Chapter 3

BACKGROUND / LITERATURE REVIEW

I will be implementing layered depth images. To do this I will create a set of programs that will generate a set of depth images, create the LDI from the depth images, render the LDI, and compute the error for a given set of images.

3.1 Layered depth images

As shown in Figure 1.1 an LDI is a stack of pixels for each image location. The actual data structure for a layered depth image is shown in Figure 3.1. You can see that along with the color and alpha information, there is also depth information for each pixel. Each pixel in the image is actually an array of ***depth pixels***. A depth pixel is a pixel with associated depth information. The list of depth pixels is built up in front to back order. The furthest pixel is the last in the list and the closest pixel is the first in the list. When rendering a pixel on the screen the list is processed in back to front order. This allows for easy alpha manipulation when rendering the unique view.

```
DepthPixel =
   ColorRGBA: 32 bit integer
   Z: 20 bit integer
   SplatIndex: 11 bit integer

LayeredDepthPixel =
   NumLayers: integer
   Layers[0..NumLayers-1]: array of DepthPixel

LayeredDepthImage =
   Camera: camera
   Pixels[0..Xres-1.0, 0..Yres-1]: array of LayeredDepthPixels
```

Figure 3.1: Layered Depth Image Structure[19]

3.2 High dynamic range radiance maps

A radiance map is a two dimensional array of radiance information. The radiance information is used to vary the pixel intensity in a more realistic manner than is accomplished by just increasing the RGB value of the pixel. Some materials reflect more light than others. The high dynamic range radiance map (HDRRM) accounts for these differences. When adjusting the light level, materials that reflect more light are increased more than the materials that do not reflect as much light. To generate an HDRRM a series of images with varying light intensities are obtained. The series of images is run through an algorithm described in P. Debevec's paper [4].

There are a couple of ways to vary the image intensity for real images. One way is to actually vary the light level. Another way is to adjust the exposure time on the camera. To change the light intensity in a computer generated or rendered image is accomplished by changing the intensity of the lights in the scene.

Chapter 4

EXPERIMENTAL FRAMEWORK AND RESULTS

The following sections outline the steps that I took to verify the operation of LDI. I will cover the depth image generation, LDI creation, LDI rendering, measuring the frame rate, and measuring the image accuracy.

All of the image generation was accomplished with the OpenGL graphics library [21]. Using OpenGL eased the task of generating synthetic images, letting me concentrate more on the details of the LDI.

4.1 Generate Depth Images

Before anything else can be done a set of depth images had to be created. I wrote a program that loosely followed Popescu's [15] method for gathering the images that are used to create the LDI. This method takes a series of images that form a semi circle around the object of interest. Figure 4.1 illustrates the method of gathering images used in Popescu's [15] paper. I vary slightly from this method. Instead of alternating sides for each image I process one side and then the other side. The first image, vector 0 in figure 4.1 is used as the LDI

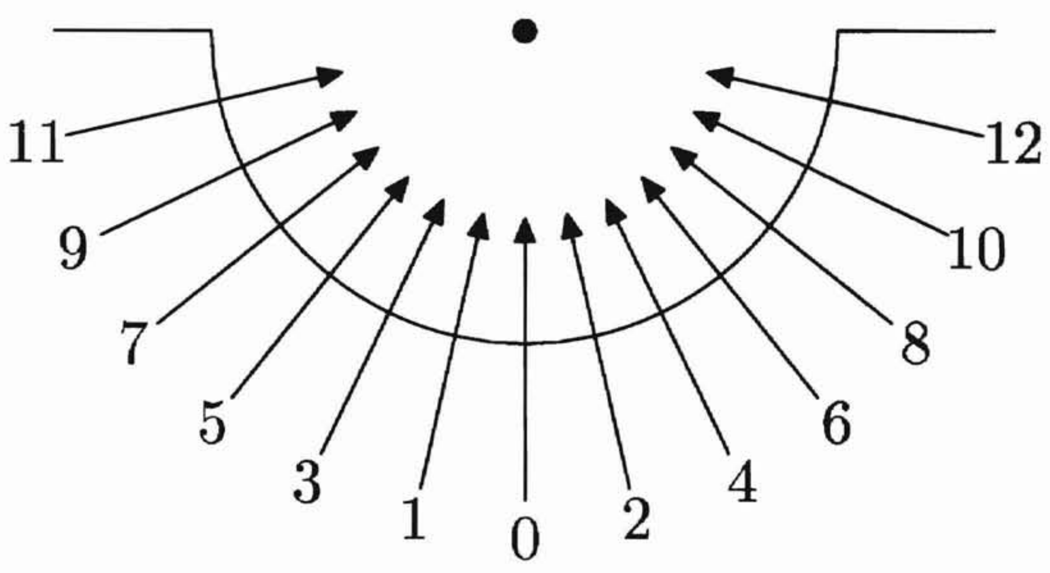

Figure 4.1: View vectors of images used to construct a LDI.

image, or LDI camera. After the first image is generated and stored, the rest of the files are generated and stored. Each image has five associated files. These files are bmp, camera, depth, mask, and normal.

4.2 File type bmp

The bmp [12] files store the color information for the image. At each pixel location, (x, y), there are twenty four bits of information stored. These bits are divided evenly into three eight bit values. Each eight bits stores a particular color. There are eight bits for each of the red, green, and blue colors. These files can be viewed using an image viewing program such as xv [1]. You can see a sample of the image in figure 4.2. The file structure for this type of file is as follows: identifier - 2 bytes, file size - 4 bytes, Reserved - 4 bytes, Bitmap data offset - 4 bytes, Bitmap header size - 4 bytes, width - 4 bytes, height - 4 bytes, planes - 2 bytes, Bits per pixel - 2 bytes, Compression - 4 bytes, Bitmap data size - 4 bytes, H resolution - 4 bytes, V resolution - 4 bytes, Colors - 4 bytes, Important colors - 4 bytes, rgb color - bitmap data size bytes.

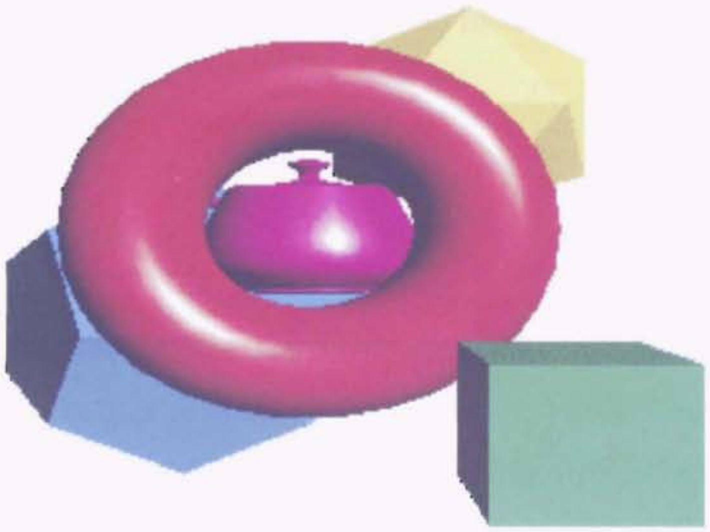

Figure 4.2: A sample of a bmp file viewed using xv.

4.3 File type camera

The camera file contains a 4x4 matrix that is the camera for the image. The camera matrix is used to project the image from the global world coordinate system to the camera's projected image coordinate system. This matrix is made up of a viewport matrix, perspective matrix, and an affine transformation matrix. If $\mathbf{C}_1$ is a camera for a given image then the camera can be expressed as $\mathbf{C}_1 = \mathbf{V}_1 \cdot \mathbf{P}_1 \cdot \mathbf{A}_1$. The image coordinates $(x,y,z,w)^T$ are obtained after multiplying the global world coordinate point $(X,Y,Z,1)^T$ by the camera $\mathbf{C}_1$ and dividing out w. Mathematically this shown as $(x \cdot w, y \cdot w, z \cdot w, w)^T = \mathbf{C}_1 \cdot (X,Y,Z,1)^T$.

The file format for this type of file is theta - 4 bytes (float data), 4 x 4 matrix - 16 * 4 bytes (float data).

4.4 File type depth

The depth file contains the depth information for the image. The depth information in these files is the depth in the camera's projected image coordinate system. Doing this lets me combine these projected depths with the projected information from the other files. I have created a program that reads this file and displays the information in a gray scale. This program is called DisplayGray. A sample image is shown in figure 4.3. From figure 4.3 you can clearly tell that the teapot, dodecahedron, and icosahedron are the furthest objects. The torus is in the center and the cube is the closest. The file format for this type of file is number of rows - 4 bytes, number of columns - 4 bytes, depth - rows * columns * 4 bytes (float data).

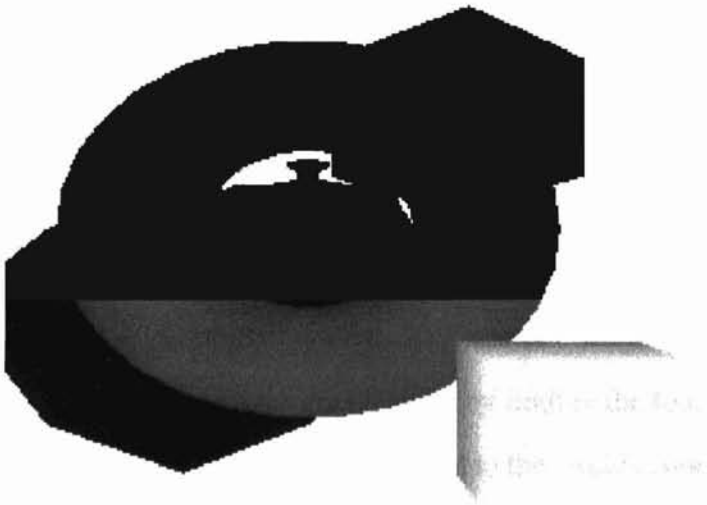

Figure 4.3: Gray scale image showing depth.

4.5 File type mask

The mask file contains a mask of the image. Each byte in the file is either a 0 or a 1. If the byte is a 0, there is no image information (i.e. background). If the byte is a 1, there is image information at that byte. Once again this is derived from the projected image. The contents of this file are used to speed up the creation of the LDI. The LDI creation program does not process any of the information from the file if the mask bit is 0. The file format for this type of file is number of rows - 4 bytes, number of columns - 4 bytes, mask - rows * columns * 1 bytes (char data).

4.6 File type normal

The normal file contains the normal information for each pixel in the image. The normals for each pixel are computed and stored in this file. This file is the one exception to the data being in the camera's projected coordinate system. The normals are from the image as it sits in the worlds global coordinate system. This is done by finding the four points next to the pixel of interest. These points are then projected back to the worlds coordinate system. This is done by taking the inverse of the camera and multiplying the points by the inverse camera matrix. If $\mathbf{C}_1$ is the camera matrix, the inverse of the matrix is computed. This matrix is $\mathbf{C}_1^{-1}$. Now the points are multiplied by the inverted camera matrix to move them to the global coordinate system. From the four points, now in the global coordinate system, I create two vectors. This is done by finding the difference between two sets of points. Now I do a check to make sure that I am not trying to compute the normal between two different surfaces. This is done by comparing the depths of the two vectors. If the difference in the two depths exceeds a preset limit, the vectors are assumed to be on two different surfaces. The limit on the preset depth is critical. If the limit is set to small, normals are not computed that need to be computed. IF the limit is to large, different surfaces will be treated as the same surface and a normal will be computed for the space between the two surfaces. The next step is to find the cross product of the two vectors. The cross product of the two vectors is another vector that is perpendicular to the plane defined by the vectors. I now have a normal for the pixel. This normal is stored in a file. The file format for this type of file is number of rows - 4 bytes, number of columns - 4 bytes, normal - rows * columns * 12 bytes (structure data). The normal structure contains three values nx, ny, and nz. These values are the normal for each of the three coordinate axis.

4.7 Constructing The Layered Depth Image

Now that all of the necessary information is at hand I can get down to creating the LDI. This is done using yet another module. The LDI creation module reads in each of the files created by the image generation module. This module takes one command line parameter. The parameter is a file name. This file in turn contains a list of files. These files are read in order. The first file read in is treated as the LDI image. Each of the remaining files is read in and the information from them is added to the information already stored in the LDI.

The procedure for creating the LDI is as follows. The first thing to do is to read in the information that will be used as the LDI camera. This information is added to the LDI just as it is read in from the files. This gives us a starting point for the LDI. The next thing to do is process the rest of the files. The first part of this is to read in the files for the second image. Now compute the inverse of the new camera. This inverse is used to create a transformation matrix. The transformation matrix, $\mathbf{T}_{21}$, will project a pixel from the new image to the LDI image. The transformation matrix is computed by multiplying camera one, or the LDI camera, by the inverse of camera two. If $\mathbf{C}_2^{-1}$ be the inverse of camera 2, the equation is $\mathbf{T}_{21} = \mathbf{C}_1 \cdot \mathbf{C}_2^{-1}$. With the transformation matrix computed the second image is ready to be projected to the first image. When the pixel from the second image is projected to the LDI, the depths are compared. If the depths are within a preset distance,

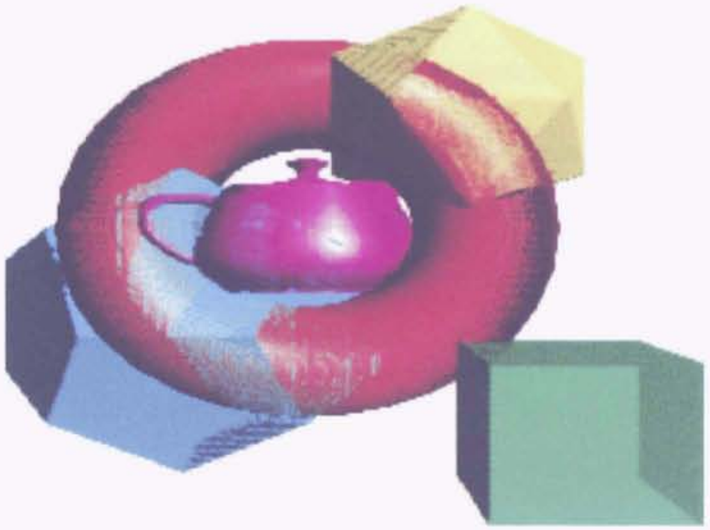

Figure 4.4: When the depth difference for creating the LDI is to large, view 1.

the pixels are assumed to be the same. The preset distance is a parameter of the algorithm and can be adjusted to optimize the generated LDI. If the pixels are at the same depth, the colors are merged. If the pixels are at different depths, the projected pixel is added to the LDI. Care has to be exercised when setting the distance that determines whether two pixels are actually the same pixel. If the distance is set to small, a lot of unnecessary pixels are added to the LDI. This probably does not harm the image appearance to much but it will reduce the rendering rate. The more pixels rendered the slower the rate. If the distance is to large, pixels that should be separate will be merged into the same pixel. In figure 4.4 the image appears to have a large number of pixels from the background bleeding through to the foreground objects. In figure 4.5 you can see that what has actually happened is that the pixels from the background objects have actually been mapped to the foreground objects.

When the process is complete the data is stored in the structure shown in figure 3.1.

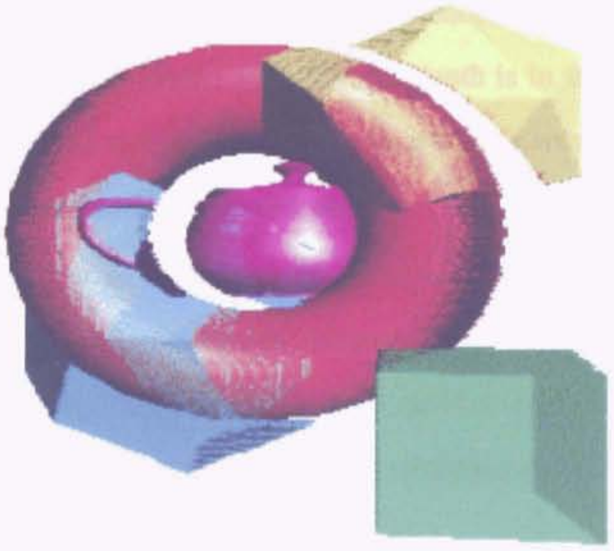

Figure 4.5: When the depth difference for creating the LDI is to large, view 2.

```
i = (normal.nx * 1.4361407) * 4.0 + 4.0;
j = (normal.ny * 1.4361407) * 4.0 + 4.0;
k = (depth*31.0)+0.5;
splatIndex = (((int)i)&0x07)<<8 | (((int)j)&0x07)<<5 | ((int)k)&0x1f;
```

Figure 4.6: Portion of encoding routine.

For the most part the population of the structure is straight forward. For example, the color entries are the colors from the source images, the depth is the depth from the source images. The splat index is slightly different. It is actually an encoding of the normals for the pixel. The equation for the encoding is shown in figure 4.6. While this is not a perfect solution for encoding it is sufficient to generate the three bits for encoding the normals for the x and y axis and the five bits for the depth; Note that the depth is in the images projected coordinate system and is linearly scaled so that it can be used as a component of the index into the array. This value is not used for the splat size.

This set of operations is one area where errors are injected into the system. The construction of the LDI is prone to all of the errors inherent in the warping operation. These construction errors are then amplified when the novel view is created during the rendering operation. One of the most obvious errors is forcing all of the pixels to integer boundaries.

Now that the information from all of the source files has been merged into one data structure, the LDI. The LDI is stored in a file. This new file is ready to be read in by a separate module that does the rendering.

4.8 Rendering the Layered Depth Image

Now that an LDI has been created we are ready to render our synthetic images. There are some initial house keeping items that need to be taken care of first. One of these house keeping chores is the generation of the Gaussian kernel. The next task is to read the LDI into memory from a file.

```
normal.x = ((i - 3.5)/4.0)/1.4361407;
normal.y = ((j - 3.5)/4.0)/1.4361407;
normal.z = sqrt(1.0-normal.x*normal.x-normal.y*normal.y);
normal.w = 1.0;

dotProduct = DotProduct(opticalAxis2, normal);

projectPoint.x = 0.0;
projectPoint.y = 0.0;
projectPoint.z = (float)k;
projectPoint.w = 1.0;
projectPoint = LDI.camera * projectPoint;
if(projectPoint.w == 0.0) {
d1 = 4.0;
}
else {
projectPoint /= projectPoint.w;
d1 = projectPoint.z;
}
```

Figure 4.7: Portion of decoding routine.

Now it is time to get down to rendering the images. A new viewing angle is entered by pressing an arrow key on the keyboard. This new angle is used to construct a rotation matrix. The first place the rotation matrix is used is to populate the splat table that is talked about in Shade's [19] paper on layered depth images. This table has to be generated once per new image. The code that performs the decoding is shown in figure 4.7. Note that the depth has been projected to the images projected coordinate system through the use of the camera matrix. This value is used to set the splat size. The decoding from i and j to the normals is also shown in the routine. This is the reverse of the encoding routine.

The next step is to find the epipole[1]. The epipole is used to determine the scan line order. A simple way to think of this is to always scan towards the epipole. McMillan has proved in his paper [11] that by following his list-priority rendering algorithm, you

[1]The epipole is the point of intersection of a line that passes through the camera centers and the image plane.

are guaranteed to generate a back-to-front occlusion-compatible ordering of the rendered image. This means that things are in back that are supposed to be in back and things in front are supposed to be in front. I followed McMillan's list-priority rendering algorithm to render my novel images. When finding the epipole, the epipole for the initial LDI image is used. This is done because the warping operation is a forward mapping operation and requires the epipole from the source image.

Just as in creating the LDI we need to create a transformation matrix. This time the transformation matrix will warp pixels from the initial LDI image to the new view location. The transformation matrix $\mathbf{T}_{12}$ is camera two times the inverse of camera one. Basically what this does is move the point from camera ones projected image plane to the global world coordinate system through the use of the inverse of camera one. Now the point is taken from the global world coordinate system to camera twos projected image plane. This can be shown mathematically as $\mathbf{T}_{12} = \mathbf{C}_2 \cdot \mathbf{C}_1^{-1}$. Now a pixel can be warped from image one to image two using the transformation matrix. $\mathbf{T}_{12} \cdot (x_1, y_1, z_1, 1)^T = (x_2 \cdot w_2, y_2 \cdot w_2, z_2 \cdot w_2, w_2)^T$ = result. Following the same procedure in Shade's paper [19] the equation is broken up into several components to speed the warping.

The next part is to actually warp the new image. This is done following the algorithm presented in Shade's paper [19]. The pseudo code from Shade's paper is shown in figure 4.8.

```
procedure Warp(ldpix, start, depth, xincr)
   for k=0 to ldpix.NumLayers-1
      z1 = ldipix[k].Z
      result = start + z1 * depth
      //cull if the depth pixel goes behind the output camera
      //or if the depth pixel goes out of the output camera's frustum
      if(result.w > 0 and IsInView(result) then
         result = result / result.w
         //see next section
         sqrtSize = z2 * lookupTable[ldpix.Layers[k].SplatIndex
         splat(ldpix.Layers[k].ColorRGBA, x2, y2, sqrtSize
      endif
   end for
   start = start + xincr
end procedure
```

Figure 4.8: Pseudo Code For Warping Algorithm[19]

Chapter 5

EVALUATION

In the next couple of sections I outline the procedure I followed to evaluate the LDI. All of the evaluations were done on a Toshiba laptop. The processor is an Intel Celeron running at 600 MHz. There is sixty four mega bytes of memory in the machine. The video chip is the ATI Mobility with four mega bytes of video memory.

5.1 Frame Rate

To compute the number of frames per second I created a function that:

1. read the system timer

2. generated a number of synthetic images

3. read the system timer

Now that I have the two times and the number of images generated I can compute the frame rate. I ran the timing routine on a number of images. I varied the size and complexity of the images. I chose the following four image sizes:

1. 200 x 100

2. 400 x 200

3. 500 x 300

4. 600 x 400

There is nothing of particular importance about these four sizes, any four sizes could have been selected. The images I chose for varying complexity are:

1. three points, one behind the other that are referred to as Three Points

2. a triangle that is referred to as Triangle

3. a complex image made of a teapot, torus, and cube that is referred to as Complex1

4. a little more complex image made of a teapot, torus, cube, dodecahedron, and a icosahedron that is referred to as Complex2

As an additional experiment I used two types of splat. The first was to use a fixed splat size of one. The second was to use a variable splat size that is selected by the normal for the pixel.

The routine that I followed to get the rendering time also includes a little overhead time for the looping operations. I tried to measure the overhead but it is so small that the effect can be considered negligible. To try to measure the overhead I set the loop counter to 36,500,000 and the time taken to run was less than a second. The timer I am using is a one second timer. For this reason I rendered a fairly large number of images so that I could calculate the average over the number of runs

As an additional experiment I also measured the time it takes OpenGL to render the same images. This is used as a comparison to the frame rate of the LDI.

5.2 Synthetic Image Accuracy

To test the image accuracy I chose a medium size image. The image size for this set of experiments was 400 x 300. Once again I used four different images. The images used here are the same four images used to measure the frame rate. I created a separate module that takes as input two images. These images are then compared pixel by pixel to determine the error. I am measuring the average color error for each pixel. I am using equation 5.1 to determine the average per color error for each pixel. The value returned from the equation is the percent error per pixel.

$$MSE = \frac{1}{9nm}\sum_{i=1}^{i=n}\sum_{j=1}^{j=m}\sum_{c=1}^{c=3}((r_{ijc}-r'_{ijc})^2+(g_{ijc}-g'_{ijc})^2+(b_{ijc}-b'_{ijc})^2) \qquad (5.1)$$

To more accurately reflect the true error, I have created a bounding box around the objects in the image. The error is the sum of the difference of all of the pixels in the bounding box squared divided by the width times the height of the bounding box. To identify the bounding box I look for the beginning and end of the images in both the row and column. This is done for both of the input images. The largest box is selected. Which may be a combination of the extents of the two images.

I performed three sets of experiments. The first experiment measured the image accuracy versus the splat size. I used all five splats 1x1, 3x3, 5x5, 7x7, and variable splat based on the normal. This experiment rotates the image eight degrees. The next experiment measures the image accuracy versus the viewing angle. I varied the viewing angle from 8 degrees to 80 degrees in 8 degree increments. This experiment uses the splat size based on the normal to the pixel. The last experiment measures the image accuracy versus the number of input images. I start with three images and go in increments of two until I have fifteen input images. This experiment rotates the image eight degrees and also uses the normals to determine the size of the pixel.

In the first two experiments I used the same set of images that I used in the timing tests. For the last experiment I used the image that had the teapot, torus, cube, dodecahedron, and a icosahedron. This image actually demonstrated some of the errors of under sampling where the other three images did not.

As a sanity check I compared an image against itself. The error returned was zero. This is what was expected. Had I received something other than zero there would have been something wrong with my error computation. Since I know the smallest error is zero I decided that it would be nice to know the maximum error. To do this I created a black image and a white image. The error returned for this experiment was one or one hundred percent. This is also what was expected. There are three color components for each pixel. Each color component has a minimum value of zero and a maximum value of one. Therefore the difference for each color would be one. The difference for each pixel would be three times the maximum difference for each color, or three. Three squared is nine which is why there is a nine term in the denominator of the equation. This equation returns the percent average error per pixel.

Chapter 6

RESULTS

In the next two sections I present the results of my experiments. I first present the data in tabular form, then in graphic form. I also interpret the data and explain the results.

6.1 Frame Rate

The first set of data are from the timing experiments varying the image size and image complexity. The data in table 6.1 shows the results of the timing test for a fixed splat size of one. The data does not present any real surprises. In general as the image size and complexity increase the frame rate drops and vice versa. There is a greater degree of change when the image size varies versus when the image complexity varies. This is as expected since we are working with IBR instead of geometric 3D rendering.

The data in table 6.2 shows the results of the timing test for a splat size based on the normal to the point. The data shows similar results to the test using a fixed splat size. The one thing to note is that the use of normals slows the rendering slightly. The images rendered with normals is generally rendered at a rate slower than the same image rendered using a fixed splat size. This would be expected since there are additional computations involved when using the normals.

Table 6.1: Rendering rate in frames per second using a fixed splat size of one. The rate is shown for several different image sizes in pixels versus several different image complexities.

Image Size	Three Points	Triangle	Complex1	Complex2
200x100	24.333334	22.812500	19.210526	18.250000
400x200	9.605263	8.902439	7.604167	6.886793
500x300	5.793651	5.214286	4.562500	4.147727
600x400	3.802083	3.411215	2.991803	2.723881

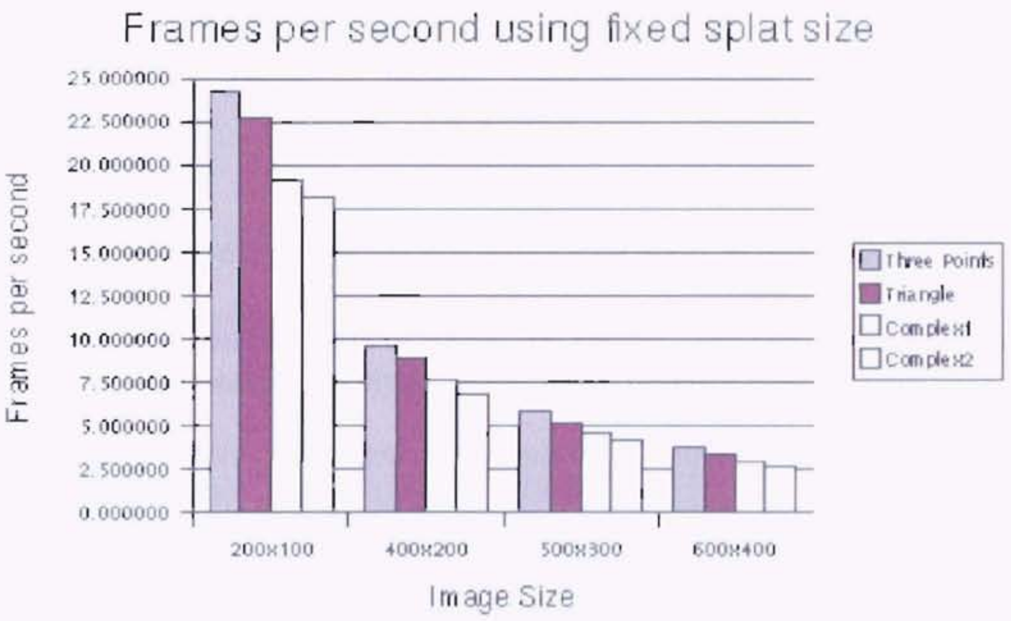

Figure 6.1: Rendering rate versus a fixed splat size of one.

Table 6.2: Rendering rate in frames per second using a variable splat size based on the normal. The rate is shown for several different image sizes in pixels versus several different image complexities.

Image Size	Three Points	Triangle	Complex1	Complex2
200x100	24.333334	21.470589	18.250000	16.590110
400x200	9.605263	8.488372	7.019231	6.293103
500x300	5.793651	5.000000	4.147727	3.686869
600x400	3.762887	3.288288	2.703704	2.433334

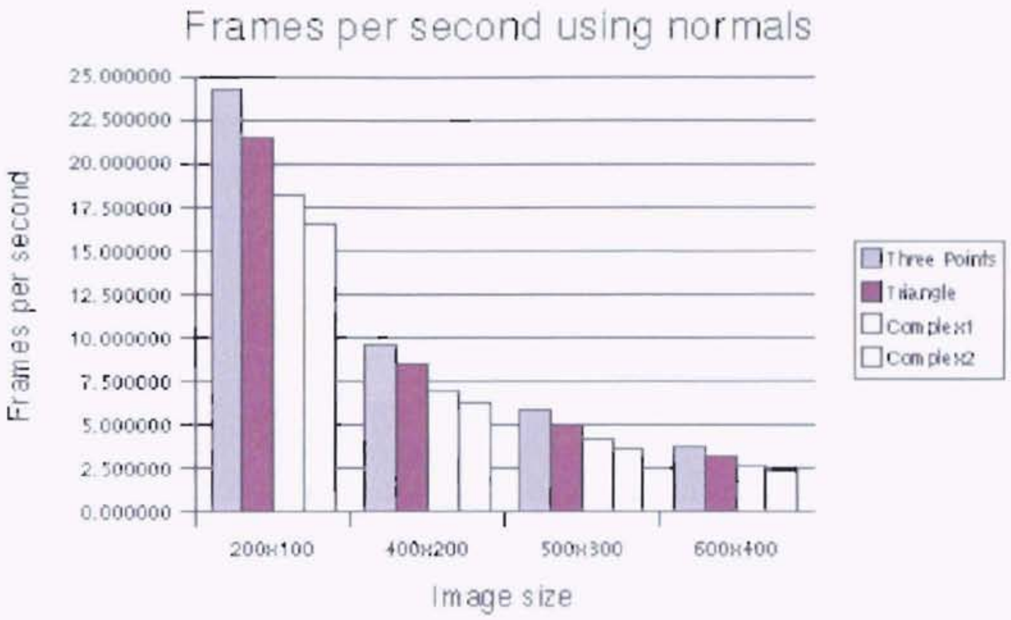

Figure 6.2: Rendering rate versus splat based on normals.

Table 6.3: OpenGL rendering rate in frames per second. The rate is shown for several different image sizes measured in pixels versus several different image complexities.

Image Size	Three Points	Triangle	Complex1	Complex2
200x100	803.000000	425.833334	21.470589	20.277779
400x200	200.750000	121.666664	12.732558	11.774194
500x300	182.500000	64.411766	9.358974	8.423077
600x400	109.500000	42.115383	7.348993	6.293103

The last experiment was to time the rendering rate for OpenGl. Table 6.3 and figure 6.3 shows the results of running the timing test on the rendering time for OpenGL.

The data does not present any real surprises. The frame rate was the fastest with the smallest and least complex of the images. The slowest frame rate was, as expected, when rendering the largest and most complex of the images. Another trend that can be seen is the frames per second varies more rapidly with image size than it does with image complexity. One point to address it the speed of the LDI versus the speed of the OpenGL rendered images. All of the images are relatively simple. Even the most complex image is composed

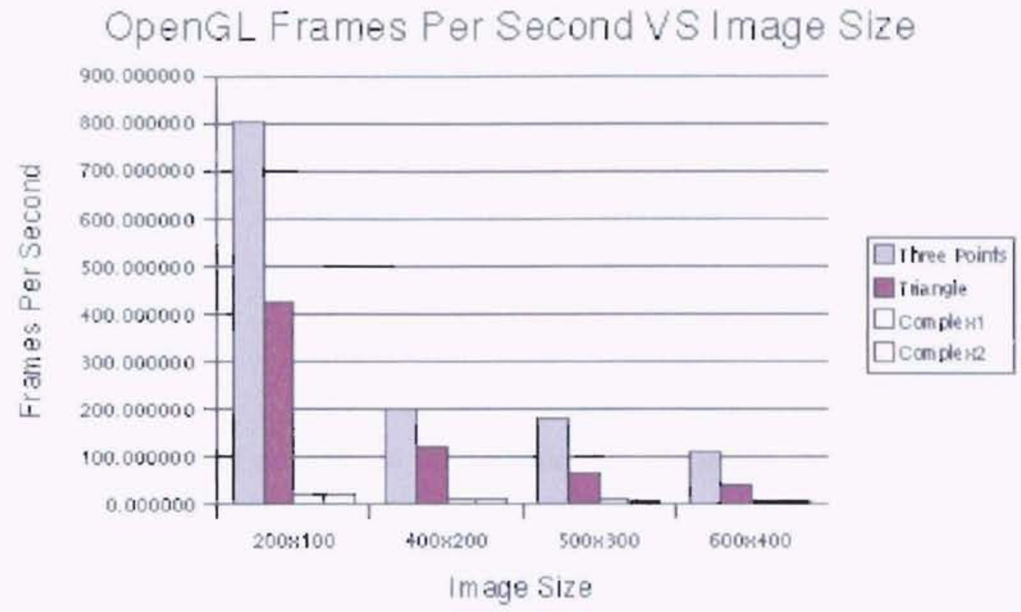

Figure 6.3: OpenGL rendering rate in frames per second.

of at most several hundred polygons. With this few polygons and hardware acceleration it is no surprise that the rendering speeds are so high. But why is the performance of the LDI so slow compared to that of OpenGL. There are a couple of reasons. The first is that I did not do anything to tailor the program to my machine. I use floating point numbers instead of integers. I also do a full screen erase between each of the images. It would be much faster to only erase what has been drawn. I have no doubts that if the LDI rendering program were tuned to particular system its performance would improve considerably.

6.2 Synthetic Image Accuracy

The results of the first experiment, image accuracy versus splat size are shown in table 6.4 and in figure 6.4. The data shows in general that the rendered images are a good approximation to the real image. One interesting thing to note from the data is that the error for a a splat based on the normal is the smallest. This is what is expected since the normals

Table 6.4: Average mean error for each color component of each pixel. The average mean error is shown for the different possible splat configurations versus several different image complexities. The angle of view for this measurement is at eight degrees from the LDI camera.

	Three Points	Triangle	Complex1	Complex2
1x1	0.021640	0.006948	0.006447	0.006814
3x3	0.033186	0.002661	0.008530	0.007654
5x5	0.035953	0.005094	0.013978	0.012824
7x7	0.054786	0.008549	0.019724	0.018522
Normal	0.021640	0.000742	0.006657	0.005764

would tailor the size of the splat to the conditions in the image. In all cases except the three points the error returned when using normals is the smallest. This would be expected since the normals were designed to aide in the creation of images that are closer to the original. In the case of the three points having the smallest error when the splat is fixed at 1x1 and based on the normal makes sense. The synthetic image is trying to represent three single points and a splat that is a single point would most closely approximate the image. Therefore a fixed splat of one would have the same error as the splat for the normals.

Tables 6.5 and 6.6 both show a series of images. The first image in the series is the result of the rendering operation. The second image in the tables is the reference image or the image that is being compared against. The third image in the tables is the actual error image. The areas that are white are the areas without error. In table 6.5 the errors are caused because we are forcing the output to integer boundaries. The holes on the interior of the image are what the splat was designed to eliminate. The splat does succeed in eliminating the errors due to forcing pixels to integer boundaries. But it introduces another type of error. This error creates a halo around each of the objects in the scene. You can see this type of error in table 6.6. If you look close at the error image for table 6.6 you will see what looks like smudges inside the objects. This type of error is caused by the merge operation when two pixels are warped to the same location during the LDI generation.

The second experiment I conducted was to measure the error as the angle of view in-

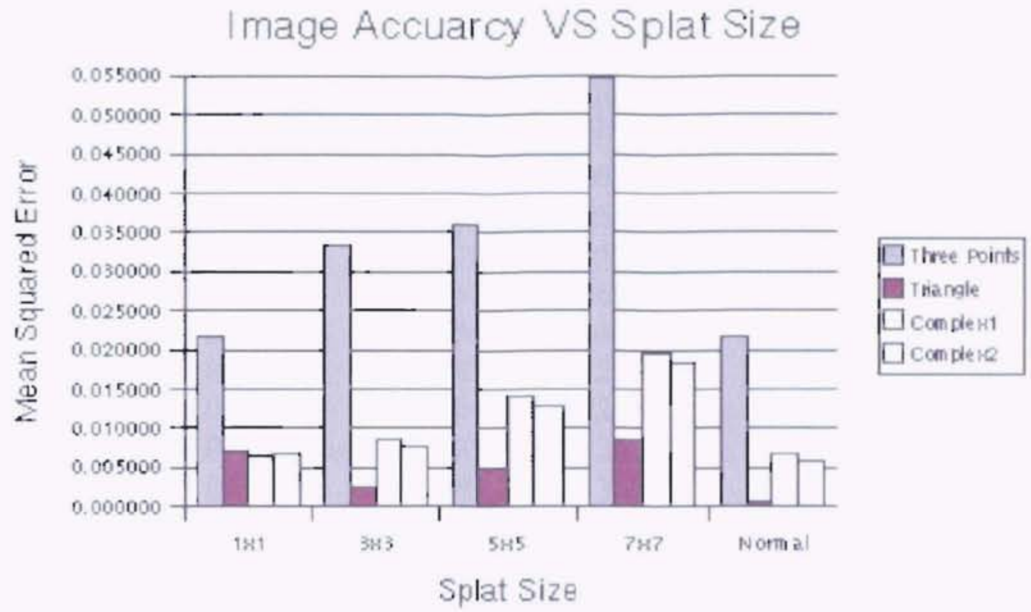

Figure 6.4: Image accuracy VS splat size.

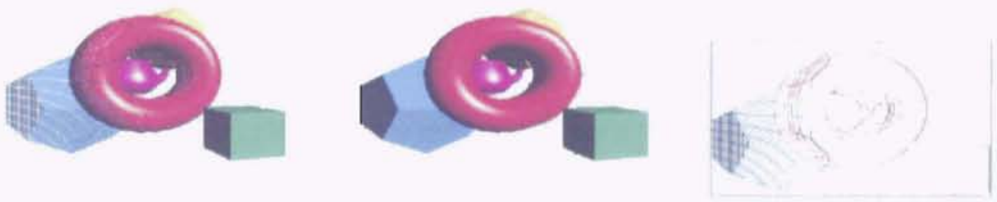

Table 6.5: Images used to calculate the error for a splat size of one.

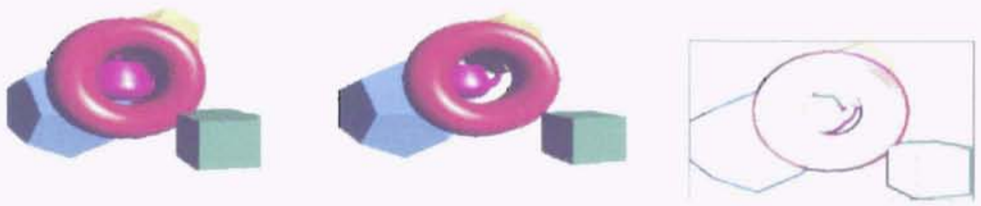

Table 6.6: Images used to calculate the error for a splat size of seven.

Table 6.7: Average mean error for each color component of (
error is shown for the different viewing angles. The images
mals to dictate the splat size.

	Three Points	Triangle	Complex1	Complex2
8	0.021640	0.000742	0.006657	0.005764
16	0.011148	0.000632	0.007980	0.006349
24	0.007508	0.000651	0.007281	0.005437
32	0.000000	0.000665	0.006570	0.005559
40	0.009197	0.000583	0.006162	0.006327
48	0.003872	0.000694	0.006794	0.007010
56	0.003375	0.000815	0.006247	0.012094
64	0.000000	0.001125	0.007404	0.012644
72	0.002705	0.001374	0.009937	0.013245
80	0.000000	0.002586	0.012649	0.013259

creases. The results from this experiment are shown in table
this data I did the rendering using the normals to set the spl
would be expected. The error increases as the angle of view i
the three points. The error for the three points continues to d
This is easily explained. As the angle increases the distance b
means that more of the background is being used as part of
sured. You will notice a few of interesting data points in the
two, sixty four, and eighty degrees the error for the three poin
angles the three points are mapped to exactly the right positi
that needs comment. When going from seventy two to eigh
from forty eight to fifty six degrees in Complex2 there is a
This is caused by the border that I force around the image in
At the points mentioned, a portion of the image is in this bor

The last experiment is to measure the image accuracy v
ages. For this experiment I chose the image that I have bee
this image because there are more occlusions than in any o
generated. The results of this experiment are shown in table

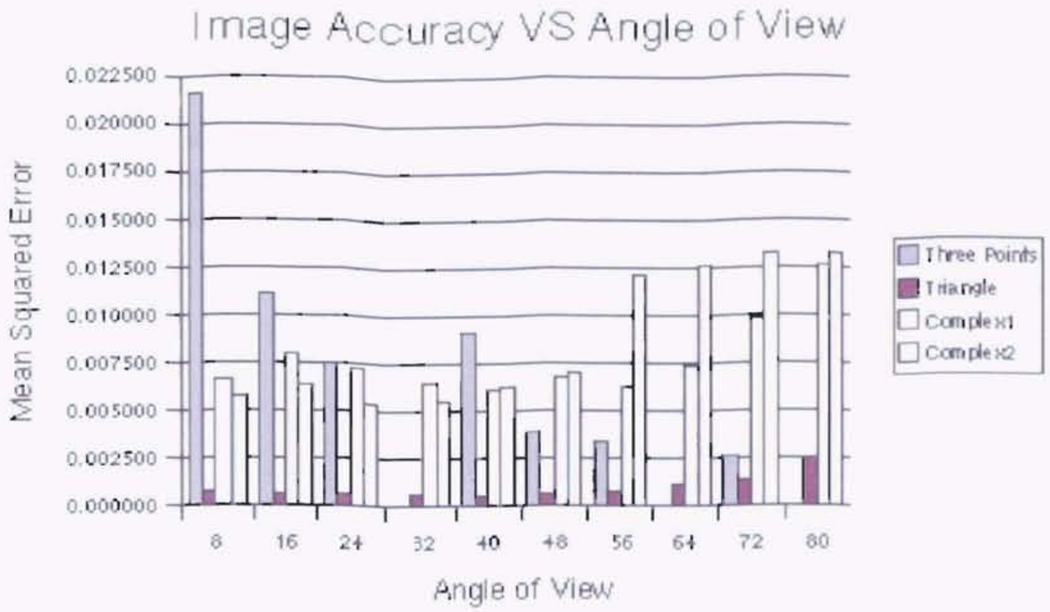

Figure 6.5: Image accuracy VS angle of view.

of this experiment was to determine if there were a minimum number of images that could be used and still guarantee an accurately generated synthetic image. The answer to this question is no. There is no way of knowing the number of images required to fully expose the objects. There have to be enough images to completely cover the scene. If there are any occluded areas that have not been taken into account, there will be holes in the rendered image. There is an example of this in figure 6.7. You can clearly see a hole in the dodecahedron in the lower left corner of the image.

There is one other type of error that I have encountered while rendering images. This error is caused by colors from background images bleeding through to foreground images. In figure 6.8 you can see the teapot through the torus. You can also see the back side of the cube through the front of the cube. This error is caused by the blending of colors when using a Gaussian kernel to splat the pixel. This type of error can be eliminated but there is a cost in terms of additional errors at the edges of objects. Instead of using a Gaussian kernel for the splat the entire kernel is at full strength. Meaning that instead of blending

Table 6.8: Average mean error for each color component of each pixel. The average mean error is shown for the different number of source images. The measurements were taken at a viewing angle of eight degrees and using the splat based on the normal.

Source Images	Image Accuracy
3	0.005848
5	0.005102
7	0.004649
9	0.004744
11	0.004800
13	0.004758
15	0.004815

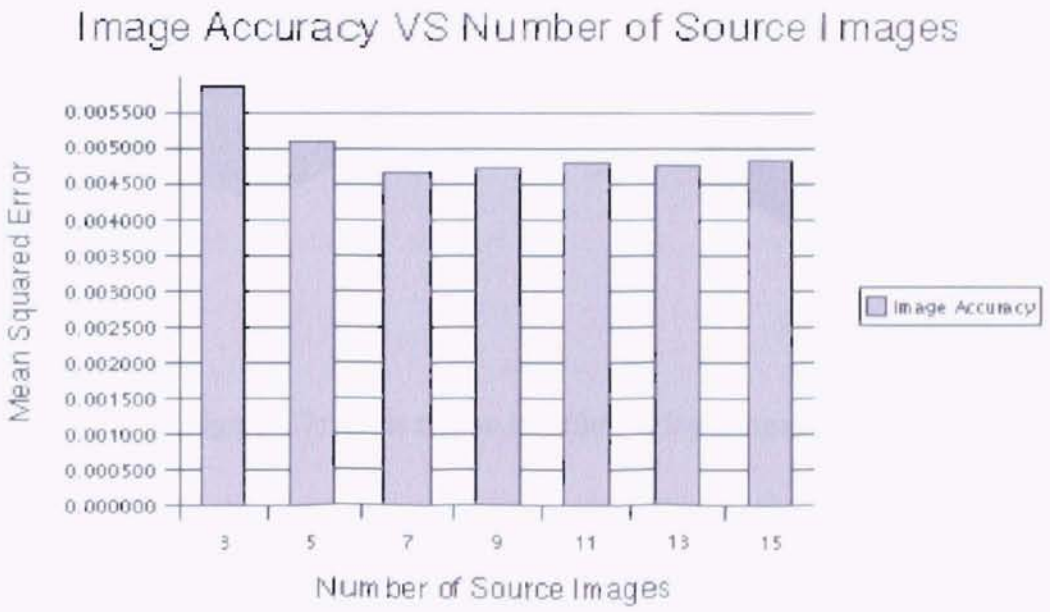

Figure 6.6: Accuracy versus number of input images.

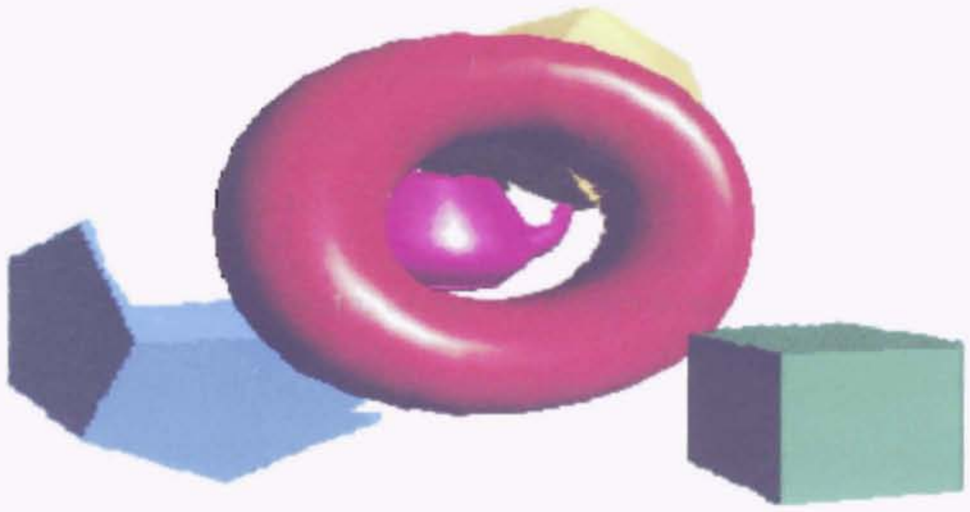

Figure 6.7: Hole due to insufficient input images.

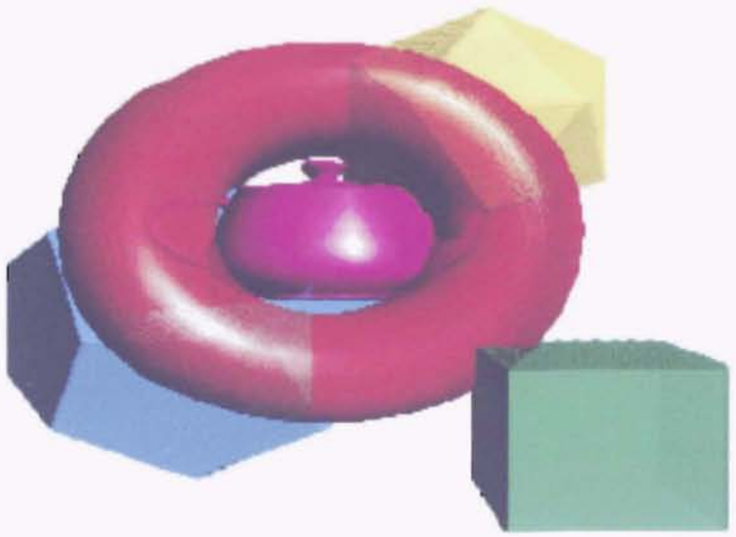

Figure 6.8: Bleed through of images in background.

the background colors with the current pixel the background is overwritten by the current pixel. This stops the bleeding through of background images but it expands the halo around the objects in the scene.

Chapter 7

FUTURE WORK

One approach that can be used to try to eliminate the errors caused by forcing pixels to integer boundaries is to do a reverse mapping. Instead of picking each pixel in the source image and mapping it to the output image the pixel in the output image is selected and the source image is searched to see which pixels would possibly be mapped to the selected pixel in the output image.

Another approach to reduce errors, particularly when the view position is fairly far from the LDI, is to dynamically create the LDI based on the angle of view [15]. This would require a little extra overhead in the form of additional disk storage space and some extra processing time. The rendering program would have to be set up to recognize when the angle of view had exceeded a certain limit. Once the limit was exceeded, the program would have to create a new LDI using the current viewing location as the LDI camera. If this location does not coincide exactly with a source image the program would have to select the closest image.

LDIs could be enhanced by adding the ability to change the lighting conditions as well as change the viewing location. This could be done using Debeveks work with high dynamic range radiance maps. A radiance map could be generated for each of the input images to the LDI creation process. Since a radiance map is just a two dimensional array of radiance information the radiance map could be warped in a similar fashion to the depth images that were used to create the LDI. The LDI structure could be extended to include a radiance value for each pixel. When a change in lighting is desired for a synthetic view, the radiance information could be used to render the image with the new lighting conditions.

Another area that could use some improvement is the selection of an encoding and decoding routine for the normals in the image. I used a rather simple means of encoding and decoding the normals. An approach that would possibly generate better results is to use the unit sphere as a model to work with the normals. This would give a full coverage of the entire area from zero to one. The method I used cuts off the top third of the normal range.

The edges also could use a little touch up. I used a method the set the splat size to one for all edge pixels. A method that could possibly produce better results is to create a second pass to the normal algorithm that would search for the edge pixels and use the normal for the adjacent pixel on the object.

Chapter 8

SUMMARY

In this study I have shown that a synthetic images can be generated using layered depth images and that these images are within a certain error, ε, of the true image. I have also shown that the images are rendered at a rate of α frames per second.

Using LDIs, synthetic images can be generated at multiple frames per second. There are several factors that can affect the actual rate of image generation. I have conducted several tests and have identified three factors that affect the rate. The factor that has the largest affect on image generation rate is the size of the image. The larger the image the slower the image was to generate. The factor that had the next to largest affect on image generation rate was the complexity of the image. The more complex an image the slower it was to generate. The factor with the smallest affect on image generation rate was whether a fixed splat size was used versus the use of normals to determine the splat size. Using a fixed splat size the images were generated at a slightly faster rate than it the normals were used. If you look back at figures 6.1 and 6.2 you will see the data from the experiments.

I have not done any system specific tuning to increase the rate that images are generated. There are a couple of things that could be done to increase the rendering speed. One thing would be to eliminate as much of the floating point math from the actual splatting of the pixels. Another thing that could be done is to erase only the area that has been drawn instead of the entire image. The program could also be enhanced by tailoring it to a particular hardware platform.

Images generated using LDIs are a visually close approximation to the original images. I ran a few tests to determine what factors played an important roll in the generated images error. The factors tested were the image complexity, splat size, angle of view, and the number of input images. All of the factors tested had an affect on the images error. The one factor that has the potential to cause the greatest error in the generated images is the number of input images. If the scene being modeled has a lot of occluded regions and there are only a few source images, the generated image will have holes in it everywhere there was an occlusion. It is very important to have enough source images to expose all of the occluded regions. The factor that would have the next largest affect was the splat size. If the splat sizes are wrong the image will have three types of visual defects. The most noticeable defect is the holes left in an image when a splat size is chosen that is to small. The next defect on the list is the growth of the object in the image. If the points at the edge of an object are generated with splats that are to large, there is a halo effect around the objects. The least detectable artifact is the slight change in color of objects because of the merging operation that is done during the LDI construction on points that are very close in distance and are deemed to be the same point.

There is one factor that has a negative impact on measuring the image error. The background is also used in the error calculation. Since the background color is the same for each of the images the error computed is reduced. The more background in the image the smaller the error. I have tried to reduce the error by creating a bounding box around the objects compared.

There is at least one defect that I found with my implementation of LDIs. Using a Gaussian kernel to do the splatting allows item in the background to bleed through to the foreground. This is most pronounced when the synthetic image is generated from the same location that the LDI camera occupies. In this case there really is not an epipole and I have fixed the epipole at the center of the image.

BIBLIOGRAPHY

[1] J. Bradley. "http://www.trilon.com/xv/", June 5, 2001. accessed on June 5 2001 at 13:00.

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

[3] P. Debevec. Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. *Proceedings of SIGGRAPH 98*, pages 189–198, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[4] P. E. Debevec and J. Malik. Recovering high dynamic range radiance maps from photographs. *Proceedings of SIGGRAPH 97*, pages 369–378, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[5] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen. The lumigraph. *Proceedings of SIGGRAPH 96*, pages 43–54, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[6] S. J. Gortler, L.-W. He, and M. F. Cohen. Rendering layered depth images. Technical Report MSTR-TR-97-09, Microsoft Research Advanced Technology Division, Redmond, WA, March 19 1997.

[7] B. Guo. Progressive radiance evaluation using directional coherence maps. *Proceedings of SIGGRAPH 98*, pages 255–266, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[8] M. Halle. Multiple viewpoint rendering. *Proceedings of SIGGRAPH 98*, pages 243–254, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[9] Y. Horry, K. ichi Anjyo, and K. Arai. Tour into the picture: Using a spidery mesh interface to make animation from a single image. *Proceedings of SIGGRAPH 97*, pages 225–232, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[10] J. Lengyel and J. Snyder. Rendering with coherent layers. *Proceedings of SIGGRAPH 97*, pages 233–242, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[11] L. McMillan. A list-priority rendering algorithm for redisplaying projected surfaces. Technical Report UNC Technical Report 95-005, University of North Carolina, North Caroline, 1995.

[12] Microsoft. http://msdn.microsoft.com/library/psdk/gdi/bitmaps4_vlh.htm, June 5, 2001. accessed on June 5 2001 at 13:00.

[13] Y. Mukaigawa, S. Mihashi, and T. Shakunaga. Photometric image-based rendering for virtual lighting image synthesis. *Proceedings of Workshop on Augmented Reality*, pages 115–124, October 1999.

[14] M. M. Oliveira and G. Bishop. Image-based objects. *1999 ACM Symposium on Interactive 3D Graphics*, pages 191–198, April 1999. ISBN 1-58113-082-1.

[15] V. S. Popescu, A. A. Lastra, D. G. Aliaga, and M. de Oliveira Neto. Efficient warping for architectural walkthroughs using layered depth images. *Proceedings of IEEE Visualization 1998*, pages 211–215, October 1998.

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

[17] S. M. Seitz and C. R. Dyer. Toward image-based scene representation using view morphing. In *Proceedings of the Thirteenth International Conference on Pattern Recognition Vol. I, Track A: Computer Vision*, pages 84–89, 1996. Available from ftp.cs.wisc.edu.

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

[19] J. W. Shade, S. J. Gortler, L. wei He, and R. Szeliski. Layered depth images. *Proceedings of SIGGRAPH 98*, pages 231–242, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[20] J. Snyder and J. Lengyel. Visibility sorting and compositing without splitting for image layer decomposition. *Proceedings of SIGGRAPH 98*, pages 219–230, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[21] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide*. Addison Wesley Longman, Inc., Reading, Massachusetts, third edition, 1999.

[22] D. N. Wood, A. Finkelstein, J. F. Hughes, C. E. Thayer, and D. H. Salesin. Multiperspective panoramas for cel animation. *Proceedings of SIGGRAPH 97*, pages 243–250, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[23] Y. Yu, P. Debevec, J. Malik, and T. Hawkins. Inverse global illumination: Recovering reflectance models of real scenes from photographs. *Proceedings of SIGGRAPH 99*, pages 215–224, August 1999. ISBN 0-20148-560-5. Held in Los Angeles, California.

[24] Y. Yu and J. Malik. Recovering photometric properties of architectural scenes from photographs. *Proceedings of SIGGRAPH 98*, pages 207–218, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

2

VITA

Charles Randall Berry

Candidate for the Degree of

Master of Science

Thesis: INVESTIGATION OF LAYERED DEPTH IMAGES

Major Field: Computer Science

Biographical:

Personal Data: Born in Flagler, Colorado, On March 9, 1961, the son of Leroy and Ruthann Berry.

Education: Graduated from East High School, Pueblo, Colorado in June 1979; received Bachelor of Science degree in Electronics Engineering Technology from the University of Southern Colorado, Pueblo, Colorado in June 1983; received Bachelor of Science degree in Computer Science from the University of Central Oklahoma, Edmond, Oklahoma in 1990. Completed the requirements for the Masters of Science degree with a major in Computer Science at Oklahoma State University in August, 2001.

Experience: Employed by Seagate Technology Inc. as a firmware engineer, 1983 to present.