Image Based Rendering With Depth Information

Dan Hathaway

December 12, 2009

1 Problem

The purpose of this project is to perform image based rendering on synthetic images with depth information in real time to produce animation. The approach is to store points in 3d space so that they can be combined when there is too much data. The intuitive algorithm on which I am basing my algorithm can be found in [2]. In that paper, points with depth information are transformed from one camera space to another and a final image is constructed by blending nearby points. In my application, the user is able to control a camera to move through an image base rendered scene where in every frame a small number of pixels are added to the data set. This project can be seen as the application of image based rendering to allow real time rendering processes that can selectively draw part of a frame, such as ray-tracing, to recycle old image data. Note that this algorithm requires synthetic images where every pixel has a depth value making it an explicit geometry IBR technique [4]. This is the crucial information not available in real world images. However, the algorithm can still be incorporated as a final processing step into those IBR techniques that either compute depth information or reconstruct the scene as a set of points in 3d space.

Here is a formal description of the problem: Let the bar notation $\bar{v}$ denote a 3x1 vector. Let $C_t = (\bar{c}_{cen}, \bar{c}_1, \bar{c}_2, \bar{c}_3)$ represent the camera center, left axis, up axis, and front axis respectively at time t. Let $A_t = \{(x_i, y_i, d_i, \bar{c}_i)\}$ be a set of updated pixels on the screen with depth d_i and color $\bar{c}_i$ information at time t. Note: increasing x, y, and d values correspond to moving negatively, positively, and positively on the $\bar{c}_1$, $\bar{c}_2$, and $\bar{c}_3$ axes respectively. Let $D_t = \{(\bar{p}_i, \bar{c}_i)\}$ represent the (world, or some other) coordinates and colors of previously rendered points in 3d space that are available at time t. That is, if $(\bar{p}, \bar{c}) \in D_t$, then this means, allegedly, that there is a point with coordinates p and color c that can be used for rendering. Let $I_t = \{\bar{c}_{n,m} : n \in [0, N_n), m \in [0, N_m)\}$ represent the image that is rendered at the end of time t. The problem of my project is as follows:

At time step t' that immediately follows t, given $(C_t, D_t, C_{t'}, A_{t'})$ compute $(D_{t'}, I_{t'})$.

The key to this problem is that $D_{t'}$ can contain as much or as little information as is desired. It is *not* even required that $D_t \subseteq D_{t'}$. The two fundamental issues that need to be solved here are:

(i) How can the size of D_t be kept from growing too large over time which would slow the algorithm? The process of keeping D_t from growing should correspond to merging redundant data points.

(ii) When moving the camera forward towards a surface, nearest neighboring points in D_t may correspond to pixels in $I_{t'}$ that are more than a few pixels away. Hence, how should $I_{t'}$ be computed so that surface does not have holes (see Figure 1)?

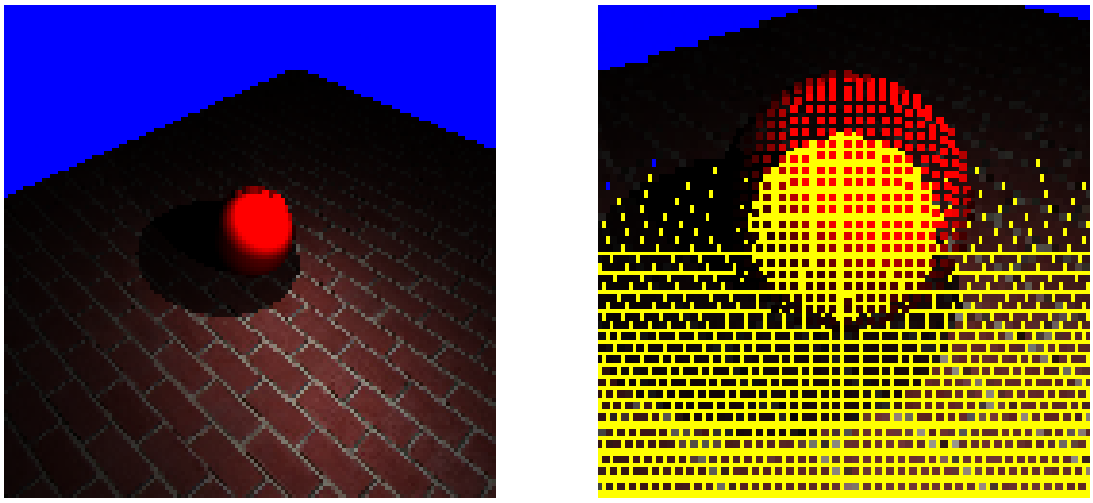


Figure 1: Moving the camera towards a surface makes points in the world more than several pixels apart on the screen. The backbuffer is initially yellow.

My solution to both of these problems will be given in the algorithm section below.

2 Algorithm

2.1 Transforming the Points

First, the equation for shifting a pixel from one camera space to another needs to be described. That is, let $C = (\bar{c}_{cen}, \bar{c}_1, \bar{c}_2, \bar{c}_3)$ and $C' = (\bar{c}'_{cen}, \bar{c}'_1, \bar{c}'_2, \bar{c}'_3)$ be two camera positions. A point (x, y, z) in camera space of C means that the point has world coordinates given by the 3x1 vector:

$$\bar{c}_{cen} + \bar{c}_1 * x + \bar{c}_2 * y + \bar{c}_3 * z$$

Thus, a point (x, y, z) in camera space C corresponds to the following point in camera space C':

$$A * (B * (x, y, z)^T - (\bar{c}'_{cen} - \bar{c}_{cen}))$$

where B is a 3x3 matrix whose columns are left to right $\{\bar{c}_1, \bar{c}_2, \bar{c}_3\}$ and A is a 3x3 matrix whose rows are top to bottom $\{\bar{c}'_1, \bar{c}'_2, \bar{c}'_3\}$.

2.2 First Tries

In trying to solve fundamental issue (i) described in the problem section, first consider a simple algorithm. Specifically, suppose that the points in D_t coincide with those in I_t except with more information (depth). $D_{t'}$ is obtained by 1) moving the points D_t that are in the camera space C_t into the camera space $C_{t'}$ 2) adding the points in $A_{t'}$ and 3) snapping the resulting points' (x, y) coordinates to lattice points so that they can be added into the image $I_{t'}$. The problem with this algorithm is the snapping phase. The positions of the points in the set D_t will migrate over time due to this snapping. The same problem occurs if we replace the snapping with bilinearly blending a point in D_t into 4 points in $D_{t'}$. Thus, we must retain the exact position of the points in D_t to prevent drifting.

For an initial version of my algorithm, I solved fundamental issue (i) by having the points in D_t represent the last $1.5 * N_n * N_m$ points from the $\{A_{t_1}, A_{t_2}, ..., A_t\}$ sets. That is, updated pixels are stored as points in the world in a circular buffer so that old points are overwritten. This solves the growth problem of D_t but in an inadequate way. We would prefer to merge spatially nearby points in D_t together. Although this could be done in some kind of 3d-BSP tree, I felt this would be too time consuming and isolated points not appearing on the screen could accumulate and thereby bloat the algorithm.

Because of this, I decided to store points in D_t in buckets where there is one bucket for each lattice (x, y) point in C_t. While similar to the algorithm described in the previous paragraph, points do not drift because each point records its offset from the lattice point that stores it.

2.3 Solving Fundamental Issue (i)

Elaborating on this process: D_t is represented in the form of a two dimensional array of *buckets*, one bucket for every pixel on the screen. Each bucket contains the same number of *tracks*, and each track contains the same number of *slots*. The purpose of a slot is to contain a single point. A slot can either be empty or filled. The specific values held by a filled slot are *depth*, d_1, d_2, and *color*. The values d_1 and d_2 represent the sub-pixel distance to the discrete x and y pixel coordinates for the bucket respectfully. The purpose of a track is to group together points with similar depth values so their attributes can be safely arithmetically averaged when a track is overfilled. The depth of a non-empty track is defined to be depth d of the first (filled) slot in the track and a point with depth d' can be added to the track iff $0.95 * d < d' < 1.05 * d$. The idea behind defining depth similarity in terms of a ratio instead of a difference is so that if two points are merged together that actually belong to different surfaces in the world, then the distance that the camera needs to be moved parallel to the image plane in order to notice the mistake is a fixed multiple of the distance from the camera to the points.

Once a track is filled and a new point needs to be added to it, the points in the track together with the new point are merged together, by arithmetically averaging their attributes, and the contents of the track are replaced with this single point. Note that while merging points together does free up space for new points, it could introduce serious artifacts due to merging points from different nearby surfaces. An alternate approach would be to kick the oldest or a random point from a track once it is overfilled. There is also a delicate question as to how many tracks per bucket and slots per track are needed. The more tracks per bucket, the more layered surfaces with different depths can be stored. The more slots per track, the more accurately the surface will be rendered. It may seem like a good idea to have enough slots per track so that almost all buckets will contain at least one point, but as we will see in the *Hole Phenomenon* section, this is not practical.

In practice, there are two different two-dimensional arrays of buckets analogous to a front and back screen buffer. The algorithm loops through the buckets in one of the two arrays putting them into the other array, clearing the old buckets along the way. Thus, it is a single pass algorithm. Implementation details from this section can be found in the file *Shifter1.cpp*.

2.4 Solving Fundamental Issue (ii)

Ignoring fundamental issue (ii), the process of creating $I_{t'}$ is not particularly difficult. A scratch buffer is used to store depth and other information. Specifically, for each pixel on the screen the values d_s, $\bar{c}_s$, and w_s are stored. The value d_s represents depth and the value $\bar{c}_s/w_s$ represents color.

Initially, every d value is set to infinity. The algorithm loops through each point in $D_{t'}$. For each of these points $(x, y, d, \bar{c})$ with floating point x, y coordinates, the algorithm loops through the 4 adjacent points $(\tilde{x}, \tilde{y})$ with integer coordinates. Given $(x, y, d, \bar{c})$ and $(\tilde{x}, \tilde{y})$ let f_1 be the distance from x to $\tilde{x}$ and f_2 be the distance from y to $\tilde{y}$ $(0 \leq f_1 \leq 1, 0 \leq f_2 \leq 1)$. At the location $(\tilde{x}, \tilde{y})$ of the scratch buffer, if $d > 1.05 * d_s$, then the point in $D_{t'}$ must be occluded by another point already written to that location of the scratch buffer. If $d < 0.95 * d_s$, then the point in $D_{t'}$ occludes everything that was previously written to the location $(\tilde{x}, \tilde{y})$ in the scratch buffer so the location is cleared: $d_s = d$, $\bar{c}_s = (1 - f_1)(1 - f_2)\bar{c}$, $w_s = (1 - f_1)(1 - f_2)$. If neither of those cases hold, then $0.95 * d_s \leq d \leq 1.05 * d_s$ and in this case we set $\bar{c}_s = \bar{c}_s + (1 - f_1)(1 - f_2)\bar{c}$ and $w_s = w_s + (1 - f_1)(1 - f_2)$. After the algorithm loops through each point in $D_{t'}$, the algorithm loops through the scratch buffer to write to the screen buffer using the formula: color $= c_s / w_s$.

In order to solve fundamental issue (ii) we need a way for pixels to affect a region that is more than 1 or 2 pixels wide. One way to do this would be to make all pixels *larger*, but this could cause surfaces to bleed over past their boundary and for blending to take place unnecessarily. The approach used here is to effectively have a high and low resolution version of the image and to use the low resolution image at a pixel location in which there is a discrepancy in depth with the high resolution image. To prevent bleeding of surfaces, the usage of the low resolution image is only used when there are high resolution pixels on either side of the low resolution pixel with the appropriate depths. This is explained below.

At this stage in the algorithm, the information in D_t is ready to create the image. The algorithm loops through every point in D_t. Let the red dot in Figure 2 be an arbitrary point in D_t (projected onto the screen). The intersection of the grey lines in the figure represent *high resolution lattice points* and the think black lines represent *low resolution lattice points*. The high resolution lattice points store the values d_s, $\bar{c}_s$, and w_s and are dealt with exactly as is described in the first paragraph of this section. The low resolution lattice points, on the other hand, store more information. That is, while high resolution lattice points are used to blend together the points in D_t with the smallest depth value, the low resolution lattice points blend together points in D_t with various depths. The point in D_t represented by the red dot in Figure 2 is recorded into the four surrounding low resolution data points. The information that is recorded into a low resolution lattice point is color, depth, and which quadrant (with respect to the lattice point) it is located.

After all points in D_t have been recorded into the high and low resolution lattice points, a post-processing step must be applied to each square region surrounded by four low resolution lattice points. A depth d is said to be *acceptable* if there are points in the regions 1, 2, 3, and 4 with depths similar to d. The post processing step consists of determining the smallest acceptable depth.

After this processing step, the final color of a pixel (at a high resolution lattice point) is computed by the following:

If the depth of the high resolution lattice point is smaller than the regions smallest acceptable depth a, then use color of the high resolution lattice point. Otherwise, bilinearly interpolate the color information from those low resolution lattice points that was recorded from points of the depth similar to a.

3 Data

The data used for this project was synthetic images generated by ray tracing. That is, following the notation of the problem formulation in Section 1, each point in the set A_t is generated by casting

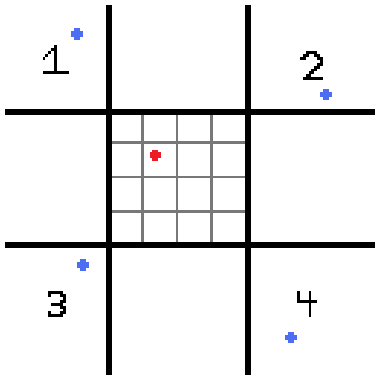


Figure 2: High resolution lattice points are at intersections of grey lines. Low resolution lattice points are at intersections of thick black lines. When there is a (blue) point in each region 1, 2, 3, and 4, then the region around the red point is contained within their convex hull.

a ray into the scene. Surfaces in the scene are Lambertian and so they appear the same when viewed from different angles. This fundamentally simplifies the problem and it allows the results to focus on the geometric aspects of rending a novel image instead of the lighting issues. For textured surfaces, the ray-tracing process determines the color of the intersection of the ray with the surface by performing tri-linear filtering. Tests of the algorithm have the following basic form:

1. Create a 3D scene that will bring out an aspect of the algorithm.
2. Render an image (completely) from one camera point by ray-tracing.
3. Move the camera through the world (yielding an animation) of image based rendered frames.
4. Compare the final (or any intermediate) frame with a completely ray-traced image from the same camera position.

Comparison is done qualitatively by taking the difference of images and converting to grayscale as in [1]. A variant of the testing method is to modify step (3) to move the camera while continuing to raytrace a sparse collection of pixels every cycle to yield the final image. This is the context in which the algorithm would be used.

4 Implementation

4.1 How to use Program

Since the purpose of this project was to perform image based rendering in real time, the program is interactive (using the SDL library). The program I wrote allows a user to move a camera through a simple scene, specified by a text file scene.txt, and to control the pixel updating process. The program is called PixShifter.exe and it is located in the Release subdirectory of the root directory of the project. Here is a description of the controls:

Mouse: changes the direction the camera is facing

ESC: exits the program

W, A, S, D, Space, C: Spatially translates the camera

P: Takes a screenshot of the pixel buffer

O: Toggles the clearing of the pixel buffer after every frame

Q: Clears the pixel buffer

E: Toggles feeding new pixel information to the algorithm

2: Disables algorithm, writing new pixels directly to the pixel buffer

3: Uses the algorithm described in this paper

4: Uses the simpler algorithm where points D_t are stored as a circular buffer

4.2 How to Build Program

On my system, I build the program by running the command "make". When I add a source file to the project, I edit the file "source_file_names.txt" and run the command "python gen_make.py" which will create a new makefile. Running the command "make" will make the final project: PixShifter.exe.

Compilation: The source code assumes that OpenGL header files can be included with the directive #include<GL/X.h> where X is gl, glu, etc. Similarly, SDL header files should be included with the directive #include<SDL/X.h>. If this is a problem, my source code can be easily changed since there are only a trivial number of such include statements. Since I use the MinGW compiler on Windows, I placed the OpenGL header files in C:/MinGw/include/GL (creating the GL directory in the process) and similarly with the SDL header files.

Linking: I placed the required OpenGL libraries in C:/MinGw/lib/GL and similarly with the SDL libraries. On a different system, the libraries need to be placed in the appropriate location, and the link target in the makefile needs to be changed to reflect the change from c:/MinGw/lib/GL and c:/MinGw/lib/SDL. The specific libraries that were used were: opengl32, glu32, SDL_image, SDLmain, and SDL. Note: the linking directive in the makefile also has the string -mwindows being passed to g++. This is needed so that SDL knows to generate a dummy WinMain function (only needed on Windows).

4.3 How to Read Source Code

This project was implemented entirely in C++ with all code written by the author. There are many source code files in this project. Many of these I have written in the past for other projects but required tuning and various other files had to be created from scratch. For instance: this is the first project where I have used doubles instead of floats for the core math functions. I also had to create classes for matrices, coordinate frames, etc. I also pulled in some ray tracing code from previous projects but had to update the engine to return the proper information needed for image based rendering. The most important source code files are the following:

MainLoop.cpp: The basic execution flow for the program

ShiftingEngine.cpp: Contains the important math functions needed by the algorithm

Shifter1.cpp: The algorithm described in this paper

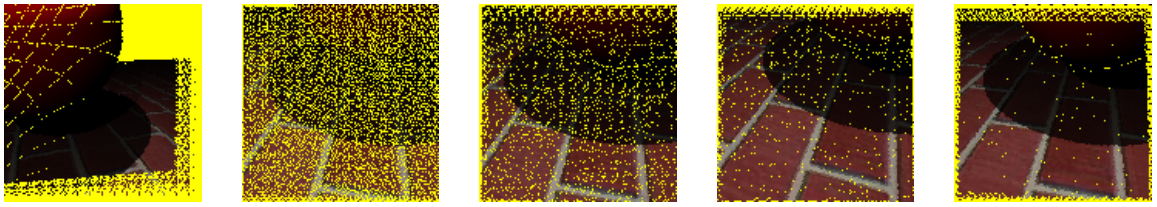


Figure 3: Left Picture: Hole Phenomenon when points are initially on lattice points. Right Pictures: phenomenon when n = 2,4,6,8.

Shifter2.cpp: Simpler algorithm that circularly discards old points

BlendProcessor.cpp: The part of the algorithm that fills in holes in surfaces by blending

5 Hole Phenomenon

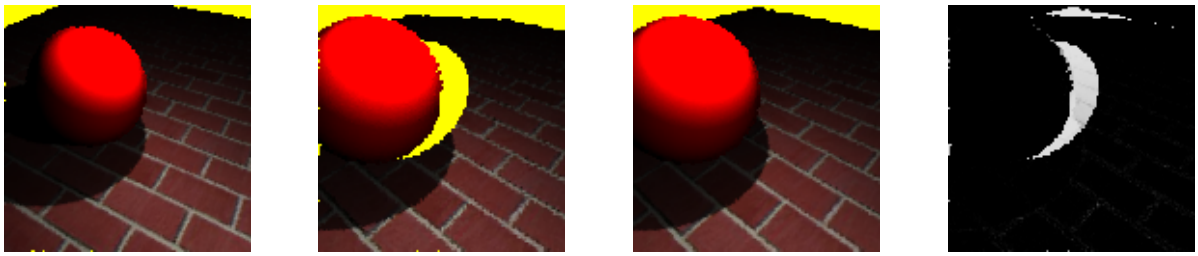
While investigating something that I thought was a bug in my implementation, I discovered what I will call the *Hole Phenomenon*. That is, when the camera turns slightly, there will be buckets that do not have any pixels in them (see bottom of section 2.2). When all points are initially on lattice points on the camera plane ($d_1 = d_2 = 0$), then turning the camera slightly causes holes to form in curvelike patterns. I believed that this problem could be fixed by initially placing points to have random d_1 and d_2 values and by increasing the number n of allowable points per bucket (one track per bucket for simplicity), but it turned out that even with large n (5 or 6), there were still many empty buckets (see Figure 3). This prevalence of empty buckets is what I call the Hole Phenomenon.

I took some time to model the probability that a given bucket would be empty. That is, suppose that the plane is filled with points such that for any integers a, b the square (a,a+1)x(b,b+1) has n uniformly randomly distributed points. Suppose that the x and y coordinates of all points are translated by λ and σ respectively. What is the probability $P(n)$ that the square (0,1)x(0,1) will contain no points given that λ and σ are both uniformly randomly chosen in the interval (0,1)? I calculated this to be:

$$P(n) = \int_0^1 [1-\lambda\sigma]^n[1-(1-\lambda)\sigma]^n[1-\lambda(1-\sigma)]^n[1-(1-\lambda)(1-\sigma)]^n$$

Using Mathematica, I found that $P(1) = 0.25$, $P(2) = 0.0662$, $P(3) = 0.0180$, $P(4) = 0.0050$, $P(5) = 0.00139$, $P(6) = 0.00040$. Even though these probabilities become small, if there are many thousands of pixels on a screen, then holes become prevalent. It should be noted that the probabilities of adjacent squares being empty are not independent. However, we could consider one quarter of the squares as having independent probabilities to obtain a lower bound. There is another force leading to holes, which is that when squares are filled with too many points, the extra points are lost / merged. This decreases the average number of points per buckets significantly after several iterations and hence makes it easier for buckets to be empty.

If at least one pixel per bucket is desired, then the algorithm could be modified to deal with overflowing buckets by placing points in adjacent buckets and modifying d_1 and d_2 values. The message to take away from this is that the process of creating an image from the buckets point data should work even with (a sparse number of) empty buckets.



(a) Original (b) Generated (c) Final (d) Difference

Figure 4: The algorithm applied to a simple scene.

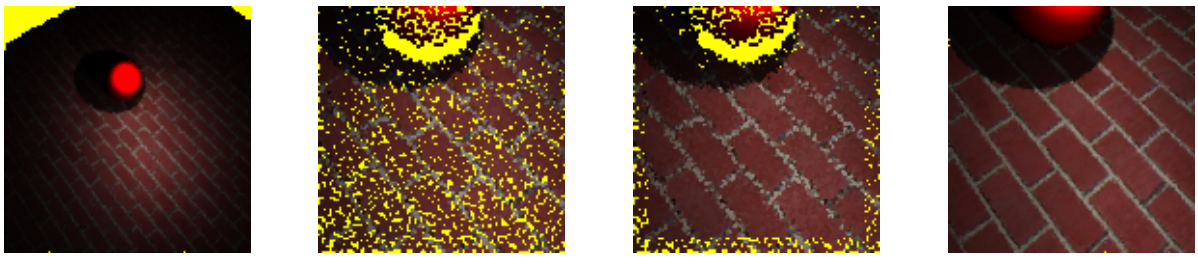
6 Results

6.1 Basic Test

For the first test, consider the 3D scene that consists of a sphere above a texture plane (see Figure 4). After creating an initial synthetic image of the scene (a), the camera is both translated and rotated to render a novel image (b). This is compared to the actual image yielded (c) at the final camera position. The grayscale difference (d) of the images is shown. As can be seen, the algorithm works well in this simple example.

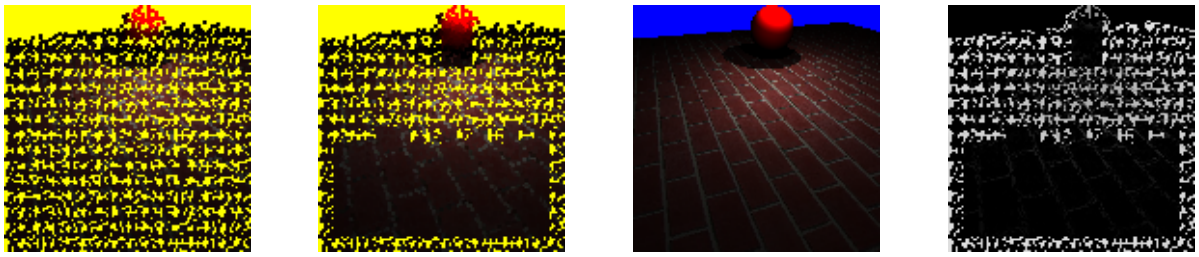
6.2 Fundamental Issue (ii) Positive Results

The next test shows the usefulness of the part of the algorithm described at the bottom of Section 2.4. Let us call that process *blending*. In Figure 5 (a) we see the original synthetic image and in (b) we see the resulting image when only high resolution lattice points are used to generate the image (there are many holes). Using the large lattice points to fill in these holes we have (c). As can be seen, this yields a closer approximation to the desired image (d).



(a) Original (b) Without Blending (c) With Blending (d) Final

Figure 5: The importance of incorporating large lattice point blending.



(a) Without Blending (b) With Blending (c) Final (d) Difference

Figure 6: Failure of blending to be applied to top of plane.

6.3 Fundamental Issue (ii) Negative Results

Alghough the blending procedure does an adequate job at filling in the gaps, the required conditions for it to be applied are difficult to satisfy. Even in the case of viewing a plane at an angle, there may not be points in regions 1, 2, 3 and 4 (Figure 2) with *similar* depths. This can be seen in Figure 6 (b) because the gaps in the upper part of the plane are not filled in.

While on one hand the requirements needed to apply the blending are difficult to satisfy, in other situations where blending should not occur, they are satisfied. For example, the surface that consists of removing a small region from the plane will be blended incorrectly. Thus, the method for determining which points are on the same surface is inadequate and should be based on more than just depth.

6.4 Fundamental Issue (i)

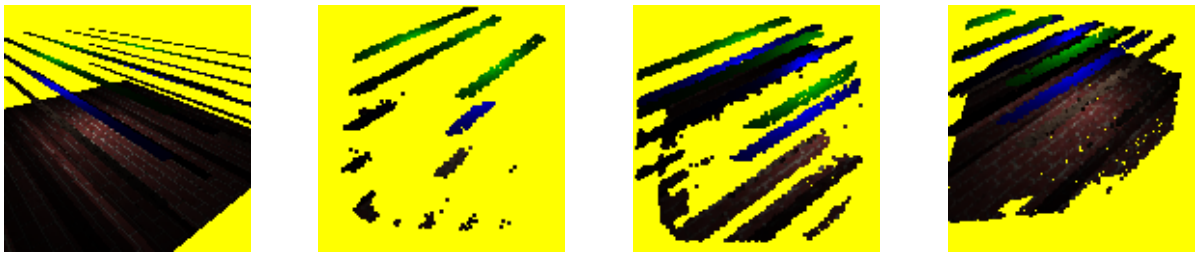
This part of the problem was solved much more effectively than the other half. That is, the method of storing points in buckets, one bucket per pixel, yielded good results while at the same time perventing the growth of data over time. Using the terminology of Section 2.3, it was found that having one slot per track caused an unacceptable artifact but as little as two slots per track gave good results. To make the algorithm fast and still have good results, I decided to use 3 slots per track.

The number of tracks per bucket needed varies per scene. For an example, consider the 3D scene, Figure 7 (a), that consists of a plane with three layers of horizontal strips above the plane. Suppose the camera is placed initially to view all of the lowest plane below the strips. Let the camera be moved over the top of the strips to the other side (performing the algorithm). Figure 7 (b), (c), and (d) show the result of this when different numbers of tracks per bucket are used.

7 Scope, Limitations, Extensions

7.1 Scope

If this technique was perfected, it could provide an alternate rendering technique in which only a fraction of the screen needs to be updated every frame. Also, this algorithm can be trivially changed



(a) Entire Scene (b) 1 Track / Bucket (c) 2 Tracks / Bucket (d) 4 Tracks / Bucket

Figure 7: The algorithm using 3 slots per track and various numbers of tracks per bucket.

so set A_t, defined in Section 1, represents updated pixels that have been sent over the network. The algorithm can allow an arbitrary amount of latency because this new data need only be transformed to the user's current camera frame once it arrives.

While having many benefits, this algorithm is restricted to the rendering of static scenes with Lambertian surfaces. The Lambertian surface limit can be loosened if the algorithm was to store lighting information for every point. While moving objects could be rendered as a post processing step, the shadows cast by these objects would cause significant changes to the image. Ignoring shadows, as well as the lighting of moving objects, the algorithm could handle moving objects by transforming the points on their surfaces differently to incorporate their timestep motion transformation in world space. However, this is straining the scope of the algorithm.

7.2 Extensions

There are many ways to modify the algorithm described. First, the number of points that can be stored in a track need not be the same for all tracks in a bucket. Having many tracks that can only hold a single point would allow low resolution detail for regions of the background occluded by multiple objects. Also, instead of using just high and low resolution lattice points for blending, multiple levels of detail could have been used.

Another way to modify the algorithm is to allow the algorithm to have more geometric information about the scene. That is, in addition to every pixel in the initial image having a depth value, every pixel could also have a normal vector. That normal vector would allow a much more accurate test for gaps in the blending process. It would also be possible to record multiple depth layers per pixel via the method of Layered Depth Images [3].

Another change to the algorithm would be to store the points D_t in buckets on the surface of a sphere instead of in buckets on the surface of a plane. Storing the points in buckets on a plane has the somewhat artificial restriction that information is lost as soon as the user rotates the camera. A caveat of storing points in buckets on the surface of a sphere is that the density of points per bucket should be uniform so when using spherical coordinates extra care must be given to the angle to the x-y plane.

The more significant change to be made to this algorithm is to redo the way in which gaps are filled (Fundamental Issue (ii)). The difficulty is being able to accurately determine what constitutes a gap in the image. The present technique yields both false negatives and false positives in terms

of detecting legitimate gaps. The technique of *splatting* a pixel onto multiple pixels on the output image [4] circumvents the gap detection issue by making the pixels large enough to prevent gaps. Clearly, the size of each pixel to splat depends on the depth of the point in both the initial and final image. I chose not to incorporate this in the algorithm because I felt that these large pixels would muddle information from future A_t sets. Also, even with splatting pixels in this manner there will still be holes given an initially sparse D_t set. This is a problem that needs attention and splatting pixels is only a band-aid.

7.3 Comparison With Other IBR Techniques

As mentioned in the beginning, the algorithm described here fits into the category of an image based rendering technique with *explicit geometry* information [4]. That is, the depth value per pixel as input constitutes explicit geometric information about the scene. Indeed, this is the crucial information lacking from real world images. Other explicit geometry IBR techniques work differently than in this paper. Specifically, instead of representing the scene in terms of a set of points in space, the scene could be represented as a textured mesh (Relief Texture technique [4]). I chose not to do this so that later information could easily be incorporated into the procedure. That is, when information from A_t entered that algorithm, this would have required the mesh and its texture to be recomputed.

Apart from having explicit geometry, there are the cases of having both no geometry and implicit geometry. For the techniques that do not use explicit geometry, such as Light-Field, Lumigraph, and Mosaicing [4], an image is rendered without having any depth information or calculating such information intermediately. These algorithms are less applicable for the problem in this paper because of the limitations they pose on the position of the new camera. In the simplest case, mosaicing, the position of the novel camera is the same as the position of the original cameras, differing only in orientation. For Light-Field and Lumigraph, the novel camera must be outside the convex hull of the objects in the scene, which is too restrictive for the desired applications.

The implicit geometry IBR techniques, such as View Interpolation, and View Morphing and Transfer Methods [4] work by computing geometric information about the world even though geometric information is not explicitly provided in the input images. For example, view morphing works by computing point correspondences in the input images from cameras that are parallel and this can be used to determine disparity and depth as was done in class. With this depth, the algorithm described in this paper could be applied to generate a novel image. For instance, a vehicle with two parallel mounted cameras could use the first few steps in the view morphing process to obtain depth information, and this could be fed into the algorithm of this paper as the A_t set.

7.4 Conclusion

Overall, fundamental issue (i) was solved very effectively but the solution for issue (ii) has much room for improvement. Given that the images are synthetic, a good improvement would be to store the normal vector to every point and use this to determine which points are on the same surface. Even better, if the objects in the scene have been decomposed into convex sets and given identifiers, then these could be used to trivialize the surface gap ideintification and filling problem. The algorithm needs work, but if perfected it could yield an attractive way to render complicated static scenes in real time.

References

[1] Evers-Senne, J.-F. and R. Koch. "Image-based Rendering of Complex Scenes from a Multi-Camera Rig." *Vision, Image, and Signal Processing, IEE Proceedings*, 152.4 (2005): 470-480.

[2] Nguyen, Ha T. and Minh N. Do. "Image-based Rendering with Depth Information Using the Propogation Algorithm." *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2005. Philadelphia, PA.

[3] Shade, Gortler, et al. "Layered Depth Images." *Proceeding of SIGGRAPH98*, 1998, 231- 242.

[4] Shum, Heung-Yeung and Sing Bing Kang. "A Review of Image-based Rendering Techniques." *IEEE/SPIE Visual Communications and Image Processing*, 2000, 2-13.