

UC San Diego

UC San Diego Electronic Theses and Dissertations

Title

Interactive focus of light fields for 3D displays

Permalink

<https://escholarship.org/uc/item/4b17h33d>

Author

Lin, Karen A.

Publication Date

2007

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Interactive Focus of Light Fields for 3D Displays

A thesis submitted in partial satisfaction of the requirements for the degree
Master of Science

in

Computer Science

by

Karen A. Lin

Committee in charge:

Matthias Zwicker, Chair
Jurgen Schulze
David Kriegman
Falko Kuester

2007

Copyright
Karen A. Lin, 2007
All rights reserved.

The thesis of Karen A. Lin is approved:

Chair

University of California, San Diego

2007

DEDICATION

I dedicate this thesis to my mom and dad. Thanks for waiting.

EPIGRAPH

The real voyage of discovery consists not in seeking new landscapes but in having new eyes. —Marcel Proust

TABLE OF CONTENTS

Dedication	iv
Epigraph	v
Table of Contents	vi
List of Figures	viii
Acknowledgements	x
Abstract of the Thesis	xi
Chapter 1. Introduction	1
1.1 Motivation and Applications	1
1.2 Approach	2
Chapter 2. Related Work and Background	4
2.1 What is a Light Field?	4
2.1.1 The Plenoptic Function	4
2.1.2 The 4D Light Field	6
2.1.3 Epipoar Plane Image (EPI) Analysis	6
2.1.4 From Images to Sampled Light Fields	10
2.1.5 Interpolating New Views	11
2.2 3D Displays	13
2.2.1 Stereo Displays	14
2.2.2 Autostereoscopic Displays	17
2.2.3 Tracking Devices	17
Chapter 3. System Overview	19
3.1 Display	20
3.2 Hardware	20
3.3 Tracker	20
3.4 Software Platform	21
3.5 Plugin GUI Overview	21
3.6 Input Data Collection	22
Chapter 4. Rendering Algorithm	23
4.1 Elliptical Filters	23
4.2 Filter Bounding Box	26
4.3 Rendering Stereoscopic Images from Filtered EPIs	28
4.3.1 Convolution	28
4.3.2 Varying the Parallax Axis	29

4.3.3	Parallax versus Depth with Stereo	29
4.4	Interactive Focus	31
4.4.1	Determining Filter Orientation	31
4.4.2	Determining the Search Area	33
Chapter 5.	Implementation	35
5.1	Scene Graph	35
5.2	Initialization	36
5.3	Loading and Updating Filter Parameters	37
5.4	MouseClicked Event	38
5.5	GLSL Shaders	41
5.6	Workarounds	44
Chapter 6.	Results	47
6.1	Performance	47
6.2	Sampling	48
6.3	Filter Parameters	49
6.3.1	Edge Threshold Parameters	53
Chapter 7.	Future Work	56
Appendix A.	Implementation Details	59
A.1	Image Data Structures	59
A.1.1	Image	59
A.1.2	ImageSet	62
A.1.3	Filter	64
A.2	OpenSceneGraph and COVISE Object Extensions	66
A.2.1	DepthOfFocus	66
A.2.2	StereoGeometry	68

LIST OF FIGURES

Figure 2.1: This shows how the plenoptic function defines the direction of a light ray is specified by θ and ϕ at a point in 3 dimensional space (x, y, z) [12].	5
Figure 2.2: This shows the new parameterization of the unobstructed light field space in terms of two parallel planes, uv and st. This Light Slab formulation comes from Levoy and Hanrahan [12]	6
Figure 2.3: This shows the s-t-u space of an image set, created from stacking all the images in order or their camera positions, u.	7
Figure 2.4: This shows the relation between Light Fields and EPIs: (Left) A point is projected onto the st plane from projection centers u_0 and u_1 . The horizontal disparity of the projected point is defined relative to its projection center as $\Delta s = \ s_0' - s_1\ $. (Right) The lines in the EPI correlate closer objects with less slope of $\Delta u / \Delta s$	8
Figure 2.5: Scanline $t = 77$. Note the light pixels, corresponding with the six white columns	9
Figure 2.6: Scanline $t = 193$. Note the less sloped pixels corresponding with the water fountain in the foreground, compared to the pixels from the background columns	9
Figure 2.7: Scanline $t = 329$. Since all pixels along this scanline are at approximately the same depth, the slopes appear to be very similar.	9
Figure 2.8: Each highlighted scanline corresponds to each of the slice images. . .	9
Figure 2.9: This shows sheared perspective projection applied to the images captured along the camera plane in order to make the st samples line up at regular intervals[12].	10
Figure 2.10: We center our filter, outlined in blue, on the focal point. The red lines are the slope of the filter. The two horizontal black lines represent the slice that will be the 2D image plane for each eye. Note how part of these black lines fall outside the captured pixels.	12
Figure 2.11: Each filter outlined in blue lines up with the pixels that correspond to a point on the fountain. This keeps the fountain detail intact in the final image. The filter outlined in red is in an area with a different depth than the fountain, thus many different colors from different depths get combined in the filter, and is seen as blurring in the final image.	13
Figure 2.12: This shows multiple rays intersecting at the focal plane in a light field. The rays will be equivalent if the virtual object surface intersects with the focal plane, giving a sharp focused pixel color. The rays will have differing values if the object surface does not intersect with the focal plane, giving a blurred pixel color.	14
Figure 2.13: This shows how multi-view displays select out pixels to show for different view positions, and for autostereoscopic usage, for each eye[6].	16

Figure 3.1: The system consists of a screen, two projectors, a wand, and polarized glasses with an attached tracker for the user to wear.	19
Figure 4.1: The left shows ellipses, isocontours of the function, with the same transformation matrix but with varying r values. The largest ellipse in red corresponds to the largest r value. The right figure shows the ellipse corresponding to the r chosen to define a bounding box for the filter kernel.	24
Figure 4.2: All filter parameters are held constant except for shearing. From left to right, the shear values are $b = 0, 6$, and 27 . $\sigma_s = 0.8$, $\sigma_u = 10$	26
Figure 4.3: Red pixels outside the sheared bounding box are skipped over, while all pixels in left image are processed with a traditional rectangular bounding box.	27
Figure 4.4: This will show a diagram of the two head positions and the relating images seen in each eye.	30
Figure 4.5: This shows variance computed over an entire image for the original scene, top. The left variance image is calculated for a filter shear of 12 pixels. The right variance image is done with a shear of 22 pixels. Note the large number of black pixels which represent zero variance in areas without strong edge features.	32
Figure 4.6: In areas of the EPI with uniform color, multiple orientations can have a minimum variance of 0. By ignoring zeroed variance, the blue orientation will be found, but this is still the wrong orientation. To get a unique orientation that corresponds to the area, we check for variance near edge features, so that the values between filters are more dissimilar and the correct orientation (middle) can be found.	33
Figure 5.1: This is the main scene graph. The new nodes associated with this plugin are in bold.	36
Figure 6.1: This shows different filterHeight (σ_u) values in increasing order. A shorter filterHeight corresponds to a longer focal depth in the simulated camera. In the extreme case, of filterHeight = 1, everything in the image is in focus, as seen in the top image.	50
Figure 6.2: This shows different focal planes based on filter shear. The image on the left is focused on the fountain, so it has a larger shear than the image on the right, focused on the background tree.	51
Figure 6.3: This shows different focal planes on a different data set. Notice how the crispness of the text on the train is preserved in the lower image.	52
Figure 6.4: This shows two different view perspectives rendered with our algorithm. It shows the parallax axis as centered with the focal plane. The door right behind the center of the fountain moves from the left to the right of the spouting water. However, the bench on the left appears to move the opposite direction, from right to the left. The fountain lies in the plane of the parallax axis, therefore it does not appear to move.	55

ACKNOWLEDGEMENTS

I want to thank everyone who has contributed to the progress of this thesis. Thanks to Richard Liu for relating his knowledge of the thesis writing process and for jump starting my writing with his thesis template. Thanks to Tung-Ju Hsieh for helping me with OpenGL in OpenSceneGraph. Thanks to labmate Philip Weber who not only got me up to speed with all things COVISE and OpenSceneGraph, but also took screenshots for me when I was no longer in town, after driving me to the airport at 5am. Thanks to my busy sister, Teresa, for painstakingly editing various versions of this paper. Special thanks to my great advisors, Jurgen Schulze and Matthias Zwicker, for always being available to answer any of my small questions, for imparting their invaluable guidance, and for giving me the opportunity to do such interesting research while learning so much more than I expected. Thanks to my other committee members David Kriegman and Falko Kuester for their input. And, of course, I could not have accomplished this without the patience and support of Hubert Choi, Isabelle Ree, and my parents, through all the frustrations and unexpected. Thanks for being so understanding and for cheering me on throughout this process.

ABSTRACT OF THE THESIS

Interactive Focus of Light Fields for 3D Displays

by

Karen A. Lin

Master of Science in Computer Science

University of California, San Diego, 2007

Matthias Zwicker, Chair

We desire to render scenes onto 3D displays for realtime applications based on image data, targeting media such as streaming multi-view video. The conventional method for rendering would be to first perform 3D reconstruction of the image data, then render the 3D geometric and texture data. However, this approach would be too slow and would lack robustness for complex scenes. Instead, we take an image based approach by using appropriate filtering.

Our approach interpolates limited light field data of a 3D scene to simulate depth of field, so that we are able to render new stereo pair views from camera array data in realtime. This allows users to roam in front of a 3D display and still experience seamless natural viewing. We produce these images by convolving 2D image data with an elliptical Gaussian. By shearing the line-like Gaussian to minimize variance over an area-of-interest, the application determines the depth of a user designated pixel. This lets users change focus in realtime, so that their vision does not feel inhibited by the shallower focal depth. Instead, they are able to better focus on the details of their current

interest. Our method can be combined with eye trackers and video data for minimally invasive surgery, video conferencing, or any other remote controlled interactive visual applications for enhanced realistic viewing experience.

Chapter 1.

Introduction

We desire to render scenes onto 3D displays for realtime applications based on image data, targeting media such as streaming multi-view video. The conventional method for rendering would be to first perform 3D reconstruction of the image data, then render the 3D geometric and texture data. However, this approach would be too slow and would lack robustness for complex scenes. Instead, we take an image based approach by using appropriate filtering. By allowing users to also change focus in real-time, the user does not sense their vision being inhibited by the shallower focal depth. Instead, they are able to better focus on the details of the scene they are most interested in at that moment.

1.1 Motivation and Applications

The final goal of this application is to work with live video streams. As a proof of concept, we work with static images, while focusing on an implementation that would run in realtime. By interpolating images based on a single focal plane, we generate depth of field effects in the displayed image. Our choice of a single focal plane is because we want the camera video streams to process in realtime. This would lead to applications in minimally invasive surgery, telecollaboration, remote operation of

vehicles and robot arms. Alternatives such as full 3D reconstruction of the scene and rendering with conventional techniques would be hard to achieve in realtime. Thus, our technique leads to intuitive results by creating natural looking images from a minimal amount of required input.

Our enhancement to the 3D display becomes essential to the operator of remote controlled robotics and minimally invasive surgery. Depth of field has the characteristic of blurring out distracting peripheral details from a person’s vision, in order to emphasize the area of interest. This way, the viewer can concentrate on what is important. The freedom to move and see different viewpoints is also pertinent to such applications so that the user is unable to perceive the discretization of the display as much as possible. This allows the user to concentrate at the task at hand, instead of being distracted by unnatural artifacts and limitations in the simulated vision.

1.2 Approach

In general, 3D displays render scenes by interpreting data structures that represent the scene in terms of geometric primitives, textures, and lights. However, attaining good image quality from this method is too computationally expensive to be done in realtime with current hardware. We can take advantage of image based rendering on 4D light field samples to give the appearance of 3D rendering with lower computation costs and better image quality.

By using 4D light fields, we also gain the extra dimensionality of depth. We can thus enhance the 3D sensory capabilities of these displays. To do so, the light field is processed to create corresponding perspectives of 3D scenes with modifiable focal plane and focal length to render onto stereo and multi-view displays. This enhances the user experience through more realism, and decreases viewing discomfort [18] by blurring out depths where the eyes cannot properly fuse the stereo pair due to large disparities. This is all realized through a relatively low-cost solution of camera array data.

We will follow image based rendering ideas and use image interpolation tech-

niques based on filters to render new views. Depending on the filter design choices, the desired slice of the 4D light field can be extracted. Each of the input images represents a 2D slice of the light field. A straightforward method of quadrilinear interpolation can be done between two adjacent images [8, 16]. However, this leads to many disconcerting visual artifacts, especially when there is significant disparity between the images. Instead, we choose a specific family of Gaussian filters, characterized as long and thin, sheared to match the disparity of corresponding pixels in the input data set. This yields an image with a crisp image at a specific focal plane while gradually blurring out objects situated further from this plane.

The rest of this paper details earlier related research, our method, and its results. Chapter 2 discusses strategies of light field rendering and rendering methods of current 3D displays. Chapter 3 gives a description of the 3D display we work with. Chapter 4 defines our method. Chapter 5 describes how the realtime aspect of this application is achieved. Chapter 6 gives the results and effectiveness of our plugin. Finally, Chapter 7 gives the future direction for our work.

Chapter 2.

Related Work and Background

This chapter describes features and restrictions of light fields and 3D displays that previous work has established. Since our rendering algorithm makes extensive use of many characteristics of a specific subset of light fields, a general introduction to light fields will help in following assumptions made later. We then give background on existing 3D displays, to better understand how our application fits into such systems. By knowing what features our rendering algorithm retains and what limitations we improve upon from earlier methods, we can focus later discussions of Chapters 4 and 5 on our algorithm.

2.1 What is a Light Field?

2.1.1 The Plenoptic Function

To understand the definition of a light field requires understanding the general form it was derived from, the plenoptic function. Adelson and Bergen [1] introduce this function which attempts to capture all possible views of a scene over time through measurements of light intensity.

In order for the function to describe all possible images that can be taken in a space-time continuum of a given scene, light radiance needs to be specified for every

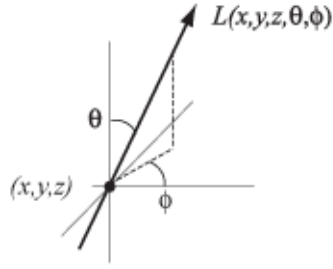


Figure 2.1: This shows how the plenoptic function defines the direction of a light ray is specified by θ and ϕ at a point in 3 dimensional space (x, y, z) [12].

point and time in space, in every direction. Seven parameters are necessary to do so, given in the form: $L(x, y, z, \theta, \phi, \lambda, p)$. First, each point V in 3D space needs to be represented uniquely. This introduces three dimensions to the function for describing the point as $V(x, y, z)$. Then two more parameters are necessary to specify the direction of each light ray passing through point V as in Figure 2.1. This is represented by θ and ϕ in spherical coordinates. Also, each light ray will have a wavelength, λ , to keep track of. Finally, the time parameter, p , is required to specify the specific moment of the radiance measurement. This gives a full description of the intensity of the light spectrum in every position, every direction, of a given time frame.

Adelson and Bergen suggest that depth information "near and far" [1] can be extracted from a subset of the plenoptic function. This way, no geometric structure understanding is needed with this light function. Instead, edges can be used as cues to interpret objects and movement in the original 3D space. This concept will help to quicken performance and increase robustness of our application.

But is this plenoptic function really more useful than following scene geometry for depth information? Considering the sheer amount of data necessary to describe the function, researchers are always looking for more ways to reduce its dimensionality.

2.1.2 The 4D Light Field

Since working with the full plenoptic function is impractical, Levoy and Hanrahan added constraints to reduce its dimensionality and yield a new definition, the 4D light field [12]. Instead of allowing any type of space-time continuum, light fields only deal with radiance in free space at a specific moment in time. Free space is defined as a region of space without any occluders. If there are no occluders, radiance does not change along a line. Therefore, this redundant data can be compressed. Instead of specifying the radiance per point per direction in space, these lines of constant radiance, analogous to light rays, now specify the light field.

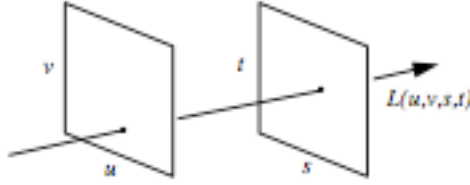


Figure 2.2: This shows the new parameterization of the unobstructed light field space in terms of two parallel planes, uv and st . This Light Slab formulation comes from Levoy and Hanrahan [12]

All rays going through a point in the free space are parameterized in terms of two arbitrary parallel planes, uv and st , Fig 2.2. The two intersections of a line with each of the planes can be used to define the radiance of each ray as $L(u, v, s, t)$. This representation is called a "light slab," since the line is analogous to a light beam entering one quadrilateral and exiting through another. Note that our convention is that the st coordinates of a ray are always measured relative to its intersection with the uv plane.

2.1.3 Epipoar Plane Image (EPI) Analysis

A 2D slice of a light slab in the st plane is analogous to an image from a specific center of projection on the uv plane. This is because the space between objects in the scene and the viewer is a free space. Each color value in the image corresponds to a

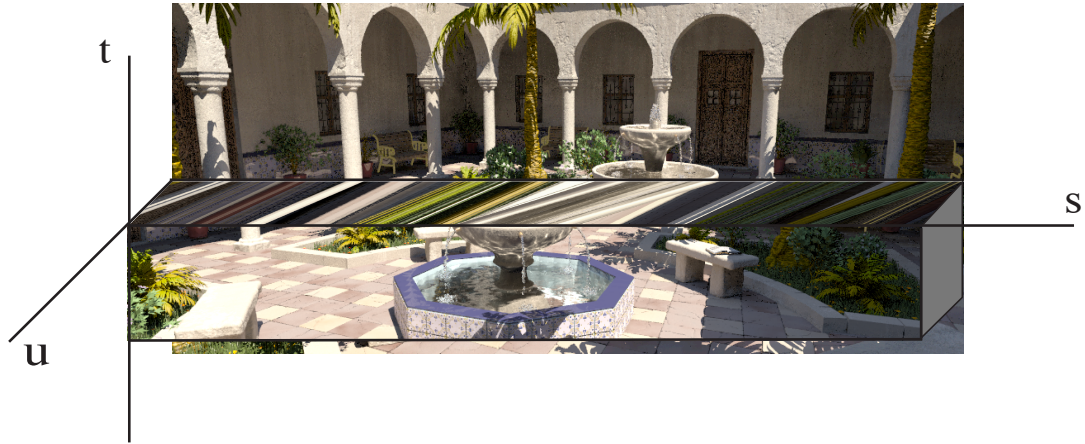


Figure 2.3: This shows the s-t-u space of an image set, created from stacking all the images in order of their camera positions, u .

ray radiance from that free space. By understanding this, we can think of images from the same scene as a subset of the rays of a light field. By analyzing the disparities across these images in the form of Epipolar Plane Images (EPIs), we can deduce depth information.

Before analyzing, we first define EPIs as used in Plenoptic Sampling [4]. An EPI is formed when the v and t values of a light field are fixed. A set of EPIs with the same v values can be visualized in the subspace of s, t, u . Each slice of the light slab's st plane is stacked in order of their center of projections, u , forming an image stack as shown in Figure 2.3. Since we are using relative s coordinates, the range of s values is the same for each image. The images are aligned along s in the EPI. This gives a better relation in horizontal pixel disparity between images. The $s-u$ slices are the EPIs, Figures 2.5, 2.6 and 2.7.

Returning to the original light slab representation will help understand how it relates to EPIs. Let us look at an example with only two center of projections lying on the same v . The distance between these two, Δu , is fixed. A point in the scene gets projected to two points on the st plane, s_0 and s_1 . The disparity in the relative s values of

these two projection points varies according to distance from the uv plane and st plane as shown in Figure 2.4. The further away a point is from the uv plane, the larger the $\Delta s = \|s_1' - s_0\|$. The closer a point is, the smaller this value gets. This means Δs approaches zero. Since the distance from the uv plane is the same concept as depth, the horizontal disparity, Δs directly correlates with depth.

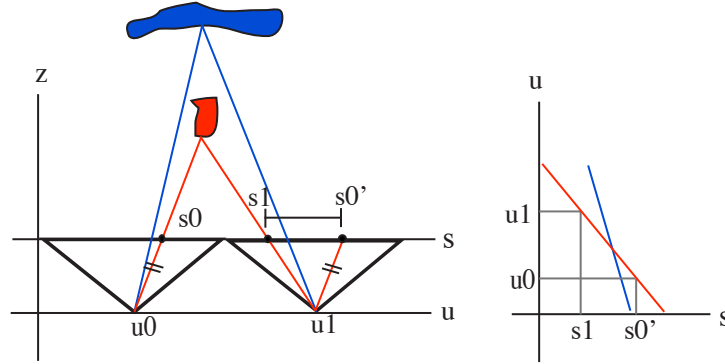


Figure 2.4: This shows the relation between Light Fields and EPIs: (Left) A point is projected onto the st plane from projection centers u_0 and u_1 . The horizontal disparity of the projected point is defined relative to its projection center as $\Delta s = \|s_0' - s_1\|$. (Right) The lines in the EPI correlate closer objects with less slope of $\Delta u / \Delta s$.

The slope of the lines in the EPIs helps illustrate and analyze this depth characteristic in a light field [7, 11]. In the EPI images, the s-u slices show lines sloped by the definition $\frac{\Delta u}{\Delta s}$. As Δs increases, the slope decreases. As Δs goes to zero, the slope goes to infinity, equal to a vertical line. Since the slope is equivalent to the optical flow from one viewpoint to the next, a vertical line signifies an unchanging pixel. Moreover, this is consistent with the characteristics of motion parallax. The further away a point is from the parallax axis, the more quickly it moves across the field of view. This follows the intuition that objects on the horizon seem to remain still while objects closer to the looker move more quickly out of the field of view. Thus, the slope of the s-u slices corresponds to the depth of a pixel in reference to a viewer in the uv plane looking toward the st plane.



Figure 2.5: Scanline $t = 77$. Note the light pixels, corresponding with the six white columns



Figure 2.6: Scanline $t = 193$. Note the less sloped pixels corresponding with the water fountain in the foreground, compared to the pixels from the background columns

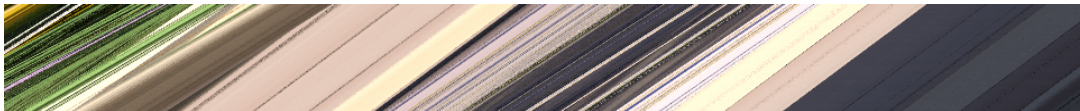


Figure 2.7: Scanline $t = 329$. Since all pixels along this scanline are at approximately the same depth, the slopes appear to be very similar.



Figure 2.8: Each highlighted scanline corresponds to each of the slice images.

2.1.4 From Images to Sampled Light Fields

A light field can be sampled by capturing images of a scene from well-chosen locations. This is based on the definition that a sampled light field consists of a regular grid of samples in the uv - st space. Each unique light ray in the free space corresponds to a pixel in an image. We can think of these images as a sample of the rays of a light field. Each image taken from a camera maps to a 2D slice of the light field in the st plane with a center of projection equivalent to the camera location, which is where the arbitrary uv plane gets placed.

This uv plane takes on a new name, the camera plane, because a regular grid of cameras lying in this plane is often used to capture more samples of the uv plane. These cameras usually have a view direction perpendicular to the camera plane, so the resulting pixels image do not necessarily correspond to the regularly spaced st samples. In order to fix this, a simple sheared perspective projection is applied to reproject each image pixel to the desired st values as shown in Figure 2.9. This is one reason that light fields and the light slab formulations are used extensively for image based rendering. Knowing the camera positions of each image makes constructing the sampled light fields even easier.

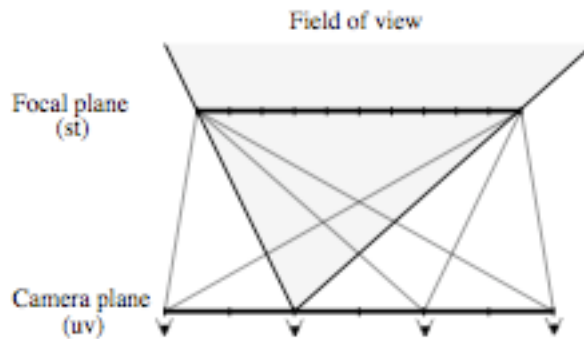


Figure 2.9: This shows sheared perspective projection applied to the images captured along the camera plane in order to make the st samples line up at regular intervals[12].

To further ease reconstructing and later analysis of the light field, our algorithm only considers light fields sampled from images captured from positions that all lie along the u axis in a camera plane. In addition to this, each camera is oriented in the same

direction, perpendicular to the camera plane. This specific sampling means the only disparity between images is in the horizontal s direction. This is defined as horizontal parallax-only.

2.1.5 Interpolating New Views

Now that we know how to construct a sampled light field from our input image sets, extracting new images from the sampling becomes our next goal. This problem is equivalent to extracting a 2D slice of the light rays passing through the desired center of projection and the range of desired st values. This st range is analogous to a projection plane with a desired field of view.

The sampled rays intersecting the st plane of the new image will not necessarily coincide with the position of the regularly spaced image pixels needed. A brute force solution to obtain these values would be to sample the light field at a higher rate. In the paper, Plenoptic Sampling [4], Chai determines the minimum sampling rate needed to generate new views free of artifacts, band-limited by the range of depth in the scene and texture complexity. He states that between two adjacent images in s - t - u space, the disparity must be "less than 1 pixel, (i.e., the camera resolution) or half cycle of the highest frequency presented in the EPI image" [4] in order to preserve textural complexity. Unfortunately, acquiring a dense enough sampling that captures all expected orientations of light rays in a free space is difficult [12]. Instead, interpolation of existing light field data extracts approximated new views without expanding the amount of acquired images.

Isaksen's paper [11] introduces an interpolation technique for undersampled light fields. He is able to emulate depth of field and variable focus with his method by combining rays from several cameras. This corresponds to filtering the EPI image, shown in Figure 2.10. Each desired pixel from a ray designated at (s, t, u) can be mapped to a position in the EPI s - u subspace. (When working with horizontal parallax-only data, the v value can be ignored, since it stays constant throughout the samples.) A filter kernel that approximates a line, outlined with a blue line, is centered at this position.

Since each u corresponds to a different camera, a filter that lies across all u values in a EPI would combine rays from all cameras.

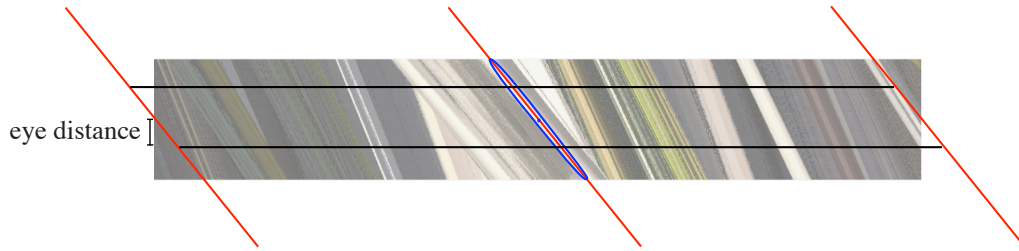


Figure 2.10: We center our filter, outlined in blue, on the focal point. The red lines are the slope of the filter. The two horizontal black lines represent the slice that will be the 2D image plane for each eye. Note how part of these black lines fall outside the captured pixels.

Orientation of the filter in the EPI determines the depth that will be in focus. Remember that in an EPI, pixels along a sloped line correspond to rays that intersect at the same point in space, shown in Figure 2.4. By assigning each output pixel to a slope orientation, which maps to an intersection point, each output pixel is given a depth value. Since each point represents the depth in focus, it is called the focal surface. If filter orientations are kept constant, then we get a focal plane. Thus, varying the filter orientation for each pixel of the output image corresponds to moving the focal surface in 3D space.

The paper Dynamically Reparameterized Light Fields [11] discusses blurring of objects at the focal surface. Since the intersecting rays that are mapped to a pixel will come from different projection centers on the uv plane, their radiance values will not necessarily be the same if we inhabit the free space with objects as in Figure 2.12. The difference of these radiance values directly affects the amount of blur. As shown in the figure, the closer the surface of an object is to the focal surface, the less disparity there is between the captured radiances. This tends to correspond to depth of field: the farther away an object is from the focal surface, the blurrier it appears to the viewer. This can also be observed in the EPIs in terms of filter orientation in Figure 2.11. In

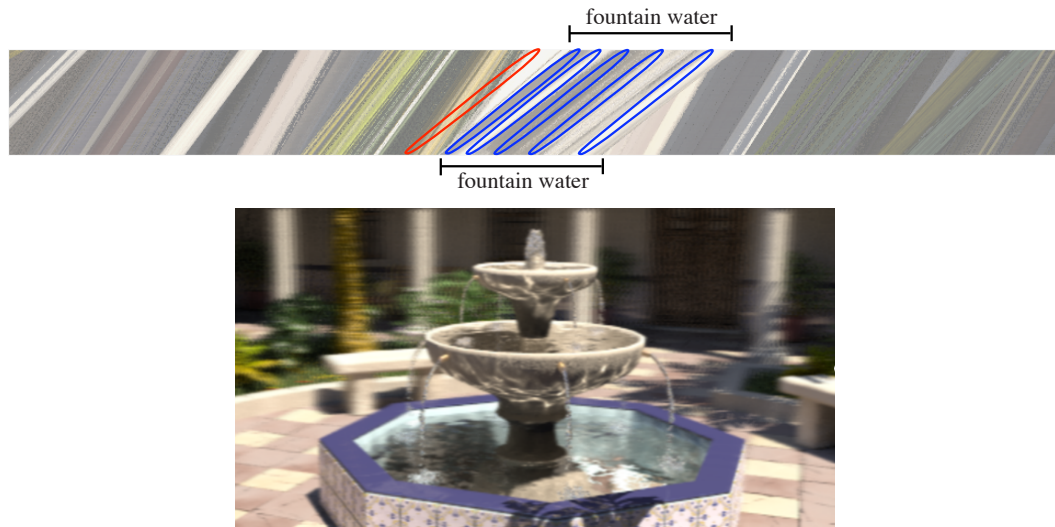


Figure 2.11: Each filter outlined in blue lines up with the pixels that correspond to a point on the fountain. This keeps the fountain detail intact in the final image. The filter outlined in red is in an area with a different depth than the fountain, thus many different colors from different depths get combined in the filter, and is seen as blurring in the final image.

an EPI, adjacent collinear pixels of similar color have the same depth, due to light rays originating from the same point on the surface of an object in 3d space. Only with a true lambertian material would the collinear pixels have the exact same color value. Stewart suggests using long thin filters, oriented to match the slope of an EPI line, to minimize blurring artifacts [16]. The more the EPI values differ within the filter kernel, the blurrier a point appears on the imaging surface, due to mixing radiance values originating from different depths, ie. different objects in the scene.

2.2 3D Displays

A traditional flat display shows one 2D image at a time from any viewing position. This means the same image is sensed on the same viewing plane in both eyes. This type of display captures the geometry, color, and texture of a 3D scene. Some subtle depth cues are available in terms of occlusion, perspective projection, lighting and shad-

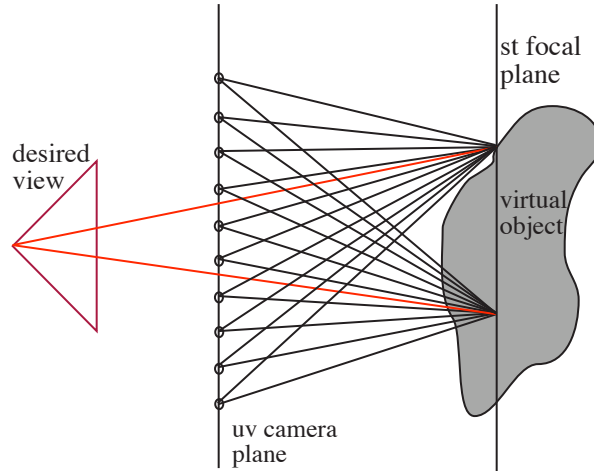


Figure 2.12: This shows multiple rays intersecting at the focal plane in a light field. The rays will be equivalent if the virtual object surface intersects with the focal plane, giving a sharp focused pixel color. The rays will have differing values if the object surface does not intersect with the focal plane, giving a blurred pixel color.

ows, and atmospheric fog. However, other major depth cues from stereo vision, motion parallax, convergence of eyes, and depth of focus are lost [5]. When humans look at a real scene, each eye has a different perspective projected onto the retina. The horizontal disparity due to distance between a pair of eyes, along with correspondences between these two images are processed by the brain to determine the depth of each part of the scene. In order to recreate this, a flat display must be able to show two images at once, and ensure each eye of the viewer sees only one of the images. These are defined as 3D displays, summarized by Dodgson [6] and Cruz-Neira [5].

2.2.1 Stereo Displays

There are two general families of 3D displays which apply to this project. The first is the stereo display. This uses specialized projectors or specialized pixel rendering and some sort of device worn by the viewer to separate the left and right images. The most well known device is the two-color anaglyphic glasses. These have a red filter for one eye and a cyan filter for the other. Any fully red pixels will not be seen by the

eye with the red filter, and vice versa for the cyan filter. Unfortunately, this method does not preserve the color gamut [10]. To address this, linearly or circularly polarized glasses are used as an alternative for separating images. By using at least one projector per eye, and putting a different filter over each projector, the light is polarized uniquely for each of the superimposed images. When the viewer wears corresponding polarized filters over each eye, only the desired projected image reaches the correct eye. Circularly polarized glasses have the extra advantage of retaining separate eye images even when the viewer's head tilts to a 90 degree angle [2].

Allowing substantial head movement also introduces more complexity to 3D displays, since stereo vision is not the only depth cue humans depend on. Movement parallax, seeing different images when the viewing position alters, is also expected. A tracking device for head or eye-gaze can be used in conjunction with stereo displays to simulate this sensory. The realtime information on the translation and rotation of the user allows the onscreen images to update accordingly. In other words, whenever a viewer changes position, by moving their head, walking to a different location, or moving their eyes, the perspective of the screen image changes to match the change in the viewer's perspective. This creates an immersive 3D environment in which users have the freedom to move around and look around occluding objects in the virtual world.

A characteristic of movement parallax is most noticeable when simplified to horizontal parallax-only. This is equivalent to walking while looking in the same direction. Objects closest to the viewer will seem to move out of the field of view more quickly than objects far away. In the extreme case, objects on the horizon appear to remain stationary. If the viewer fixates on an object and tries to walk around it, anything lying in front of the focal point will appear to move in the opposite direction than everything else in the scene, while the focal plane will remain stationary.

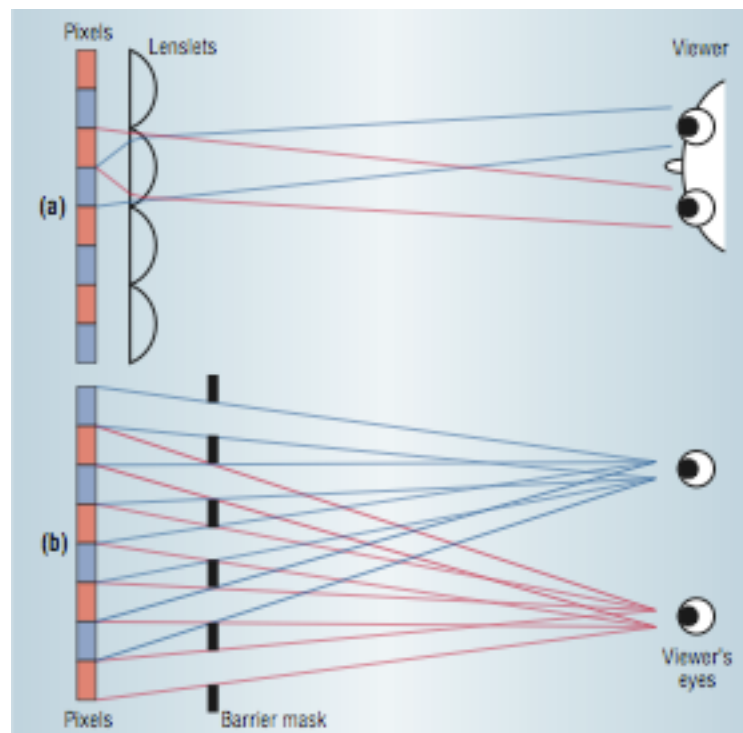


Figure 2.13: This shows how multi-view displays select out pixels to show for different view positions, and for autostereoscopic usage, for each eye[6].

2.2.2 Autostereoscopic Displays

Another family of 3D displays is the multi-view display. More than two different images can be viewed on the monitor at the same time, depending on the viewing angle. Lenticular sheets or parallax barriers placed in front of the screen then filter out strategically placed image pixels from being viewed at certain positions (Figure 2.13). The display can also be used to create an autostereoscopic display, a stereo display which does not require any head or eye gear [13]. By showing images from the same scene with the correct disparity between each view, each eye will receive the proper perspective. A disadvantage of current rendering techniques for the multi-view screen is the inability to show the exact perspective of each eye without an inordinate amount of captured views. This can be observed in Figure 2.4 and 2.9, where each view frustrum along the camera plane represents each view rendered to the screen. When the viewer is at the rendering camera position, they will see the two closest views, not necessarily the exact perspective. This creates an unwanted discretization as the viewer translates their head horizontally. We can solve this problem by generating new views from the correct perspective for each eye in realtime, thereby making the perspective change seamless to the user.

2.2.3 Tracking Devices

There are two popular types of trackers, optical and magnetic. Optical trackers use a number of cameras to determine position and direction of the viewer. This can be used for headtracking through face recognition [3], eyetracking [17], or anything with enough features to follow. Shortcomings of this system are line-of-sight requirements or insufficient illumination to pick up features to track. In comparison, magnetic trackers do not have these problems, although they have their own inherent limitations. A magnetic tracking system has a component that emits a magnetic field in three orthogonal directions, which is then picked up by sensors attached to the object or person being tracked. The sensors are made up of three coils, one in the direction of each axis, to

capture information for six degrees of freedom, three axes of translation, and three axes of rotation. Since the magnetic field goes through (most) objects without distortion, line-of-sight is not an issue. However, if there are nearby metallic objects or electronics that generate their own magnetic field, this throws off the magnetic field created, thereby compromising accuracy.

Chapter 3.

System Overview

This application is a plugin integrated into OpenSceneGraph [21] and rendered with OpenCOVER, the virtual reality rendering standalone module of the COVISE [15] package. To understand how this plugin improves the system, we will first describe this system's hardware and software. Then a guide on the plugin will describe how to use the extended features and options for the system.

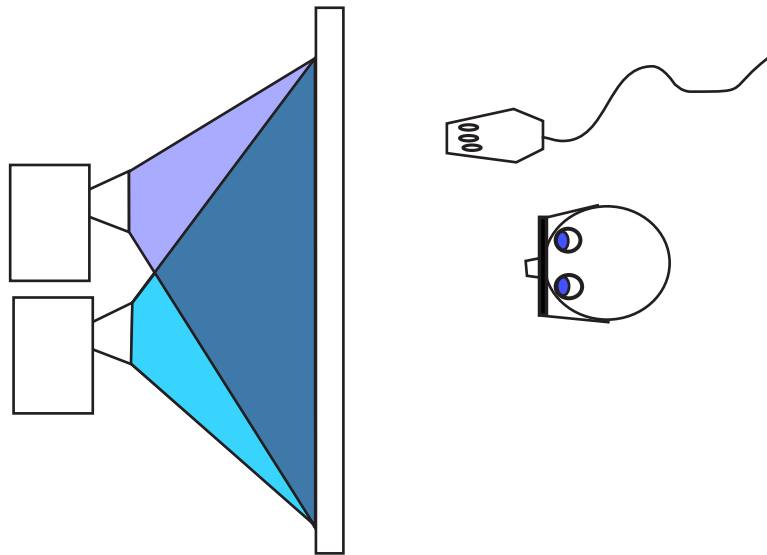


Figure 3.1: The system consists of a screen, two projectors, a wand, and polarized glasses with an attached tracker for the user to wear.

3.1 Display

The display setup, shown in Figure 3.1 involves back projection onto an 8x4 ft screen with a unobstructed viewing and walking area in front of it. Two JVC HD2k projectors, each with a resolution of 1920 x 1080 have circular polarization filters mounted. This allows two separate images to be projected with differently polarized light for the left and right eye. When the user wears polarized glasses, the images are separated for proper stereo vision, with each respective image viewed in the appropriate eye. Users normally stand around four feet away from the screen and we assume an eye separation of 60mm.

3.2 Hardware

The C-Wall features a single screen, stereo wall, driven by a PC with dual 2.6GHz AMD Opteron CPUs, an Nvidia GeForce 7900 video card, 4 GB RAM, and 1.7 TB HDD space, running under Suse Linux 10.1. It drives two JVC HD2k projectors with a resolution of 1920x1080 pixels each. The Nvidia cards have GLSL support, which is used to write shaders to accelerate the computation of finding the depth of a pixel patch and filtering the final image. We use circular polarization filters and glasses to separate the stereo images. The concept of the C-Wall is described in the paper COVISE in the CUBE[15], and a similar setup is described by Bresnahan [2]. The software driving our C-Wall is COVISE [15]. Writing new software applications for the C-Wall can be done by implementing a C++ module for COVISE, of which the underlying graphics API is OpenSceneGraph.

3.3 Tracker

The Flock of Birds tracking system by Ascension [19] is used to get realtime information on position, direction, and rotation of the user's head. This is done through

magnetic fields and sensors attached to the viewing glasses, and navigation wand.

The user holds a Wanda [23], a three-button mouse like device to click-and-drag to move objects and menus, as well as click to select menu items, which also has six degrees of freedom for tracking in 3d space. This device is rendered as a long wand in the virtual space. Whatever the wand intersects with in the virtual world, gets selected upon a click.

3.4 Software Platform

This project was implemented for OpenCOVER, the VR renderer for COVISE, using OpenSceneGraph [21]. COVISE supplied the interface with the tracking systems and ready-to-use GUI objects such as menus, dials, and checkboxes. OpenSceneGraph is an open source 3D graphics library, that optimizes rendering on high performance hardware. This library builds the scene graph data structures for OpenCOVER to render from. This library also has image and file loading support, as well as basic GLSL functions for shader program support.

For early development, other environments were used. Matlab helped to quickly determine a range of effective filter parameters. Qt 3.0 was used to write a prototype 2D version of the plugin. ShaderDesigner also eased fragment shader coding and debugging with its standalone compiler.

3.5 Plugin GUI Overview

Choosing one of the data set names in the submenu labeled "Load Scene" will load a scene. This will display an image focused on a default focus of 14, with a focal length of 40, and filterWidth of 32. The image will automatically fit to the screen width. If "Enable DOF" is selected in the Depth Of Focus plugin menu, any clicks on the loaded image will focus the image to the depth of the intersection pixel clicked upon. The parallax will also be set to remain stationary at the new focal point. This means all

objects behind the focal point will move in the direction of the user while the objects at a shallower depth will move in the opposite direction. Dials or potimeters are available to the user to adjust focal length, focal depth, and filter width.

3.6 Input Data Collection

The data sets were obtained in two ways – either by a horizontal camera array, or by a camera taking numerous shots of an actual static scene at constant intervals all along a straight horizontal path. The other option is capturing images in a virtual scene, at one moment in time, from a row of virtual cameras. Camera parameters are assumed to be constant across all images. Camera viewing angle remains perpendicular to the camera plane. Only the horizontal position varies between images. Focal length of the cameras are also set to be as long as possible, capturing data with as little blur as possible.

Chapter 4.

Rendering Algorithm

Given a series of images representing a horizontal-parallax-only light field, we want to render a pair of stereoscopic images for a 3D display. The user should have the ability to adjust the focal plane, the axis of parallax motion, and focal depth interactively. This involves manipulating the orientation and length of the light field filter. If we formulate our filter process such that we can change and control these characteristics independently and quickly, we can generate these stereoscopic pairs in realtime.

In the following sections, we discuss the formulation of the rendering algorithm. First, when given a focal plane, we derive how to construct a corresponding filter. Then we describe how to filter in order to render a new image projected to that focal plane. Finally, given a user-designated pixel, we outline the method used to find the focal plane of this pixel.

4.1 Elliptical Filters

Image processing techniques have long used Gaussians as the filter of choice. This is because the shape and orientation of Gaussians are easy to control through a series of linear transformations. If we formulate our filter as a Gaussian, we can manipulate the desired filter characteristics quickly enough for realtime applications. To

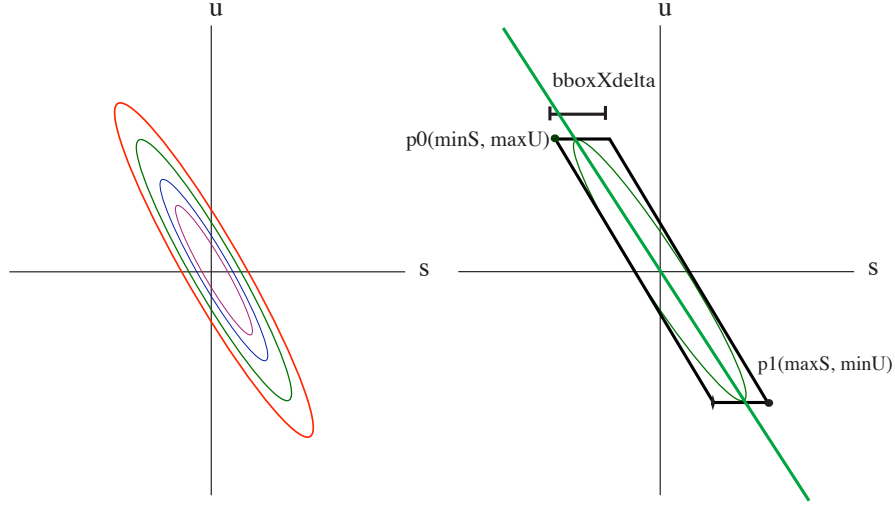


Figure 4.1: The left shows ellipses, isocontours of the function, with the same transformation matrix but with varying r values. The largest ellipse in red corresponds to the largest r value. The right figure shows the ellipse corresponding to the r chosen to define a bounding box for the filter kernel.

demonstrate this, let's begin with the simplest normalized Gaussian distribution function:

$$f(r) = \frac{1}{\sqrt{2\pi}} e^{-\frac{r^2}{2}} \quad (4.1)$$

Since r represents the distance from the symmetric center of the function, we can generalize this to a 2D elliptical Gaussian function by writing r in terms of a 2D coordinate $\mathbf{p} = \begin{bmatrix} s \\ u \end{bmatrix}$, using the implicit form of an ellipse:

$$\mathbf{p}^T \mathbf{M} \mathbf{p} = r^2 \quad (4.2)$$

With this form, the shape and orientation of an ellipse is determined in $\mathbf{M}$. The r value can then independently determine the size of the ellipse, shown in Figure 4.1.

The original radial Gaussian function would have $\mathbf{M}$ as the identity matrix. Substituting equation 4.2 into our original Gaussian function (Equation 4.1) we get an elliptical version:

$$f(\mathbf{p}) = \frac{1}{\sqrt{2\pi|\mathbf{M}^{-1}|}} e^{-\frac{1}{2}(\mathbf{p}^T \mathbf{M} \mathbf{p})} \quad (4.3)$$

By using the implicit form, we can use the 2x2 matrix, M , to hold our transformations, which uniquely define a 2D Gaussian filter in the subset we are generating.

To determine the exact entries of this 2x2 matrix, we follow Heckbert's reformulation in [9], so that we can rewrite the implicit form in terms of the transformed points. If X is defined as the concatenation of all transformations applied to the original Gaussian function, then $\mathbf{p} = X^{-1}\mathbf{p}'$ relates the transformed point to the original. This can be substituted into the original implicit ellipse equation, giving:

$$\begin{aligned}\mathbf{p}^T \mathbf{I} \mathbf{p} &= [X^{-1}\mathbf{p}']^T \mathbf{I} [X^{-1}\mathbf{p}'] \\ &= \mathbf{p}'^T X^{-1T} X^{-1} \mathbf{p}' \\ &= \mathbf{p}'^T M \mathbf{p}'\end{aligned}$$

With a little rearranging, we get a clean definition for our filter by the matrix M :

$$\begin{aligned}M &= [X^{-1}]^T X^{-1} \\ M &= \{X X^T\}^{-1}\end{aligned}$$

In our application, the concatenated transforms of X are limited to a scaling and a shearing, written in terms of two scaling factors, σ_s and σ_u , and a shearing direction b as so:

$$X = \begin{pmatrix} 1 & b \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \sigma_s & 0 \\ 0 & \sigma_u \end{pmatrix} \quad (4.4)$$

And then we can write the ellipse's M in terms of scaling and shearing.

$$M = \begin{pmatrix} \frac{1}{\sigma_s^2} & -\frac{b}{\sigma_s^2} \\ -\frac{b}{\sigma_s^2} & \frac{1}{\sigma_u^2} + \frac{b^2}{\sigma_s^2} \end{pmatrix} \quad (4.5)$$

This makes manipulation of the filter straightforward for our purposes. As we narrow the filter along the s-direction, by decreasing σ_s , and lengthen along the u-direction by increasing σ_u , the Gaussian filter becomes shaped more like a line. The b affects the amount of shear transformation applied, which changes the orientation of the filter as shown in Fig 4.2.

Why use a shearing transform instead of a rotational transform? Both transforms change the orientation of a line, but a rotational transform bundles horizontal and vertical translation together. Since we are working with a horizontal parallax-only light field, vertical pixel disparity is not captured. The EPIs show horizontal translation of pixels, with the translation amount varying directly proportional to its u position. This is analogous to shearing. A shear transformation in only the s -direction properly produces this effect on the pixel transformations.

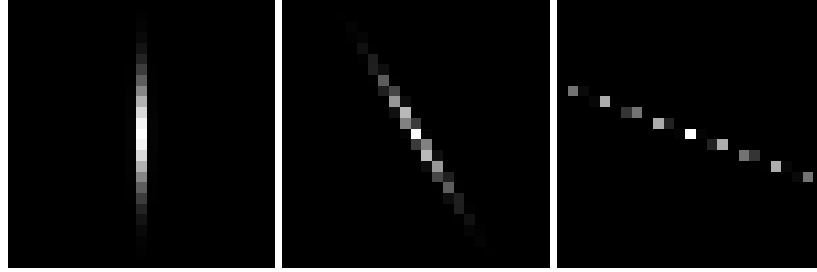


Figure 4.2: All filter parameters are held constant except for shearing. From left to right, the shear values are $b = 0, 6$, and 27 . $\sigma_s = 0.8$, $\sigma_u = 10$

4.2 Filter Bounding Box

In order to turn the continuous Gaussian function into a filter kernel, we must set cutoff values on the function, to define a bounding box. To do so, we choose a bounding ellipse by fixing the r value from equation 4.2, and then find a bounding box for this ellipse. This r will affect performance as well as robustness. Choose too small a radius, and the filter output will not be smooth due to undesirable blocking artifacts. Choose too large a radius, and too many near-zero weighted pixels will be added into the calculation, slowing performance. Typical r values falls between 1 and 2 [24]. Our plugin uses $r = \sqrt{3} \cong 1.73$ as the bounding radius.

We can now find a shear-aligned bounding box for the ellipse by taking derivatives in the direction of the ellipse's axes. Unlike traditional axis-aligned bounding boxes, we do not align to the coordinate axes. Because of the slender nature of elliptical

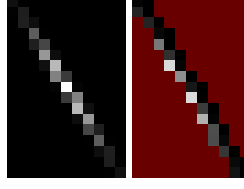


Figure 4.3: Red pixels outside the sheared bounding box are skipped over, while all pixels in left image are processed with a traditional rectangular bounding box.

filters, the rectangular bounding box would encompass more and more pixels far beyond the cutoff as the shear of the filter increased. The area of the rectangular filter kernel would grow in size quickly as the shear increased, while the area of a skewed kernel would stay constant as shown in Fig 4.3. By keeping the area constant, the number of pixels that fall into the area will not grow with an increasing shear value. Since this algorithm's computing time depends on the number of pixels in the filter kernel, we keep the calculation time independent of the filter orientation.

To define a skewed bounding box, we need three parameters: $maxU$, slope of the sheared axis, and $bboxXdelta$ as shown in Figure 4.1. The maximum u value is half of the filter height. To find the $maxU$ and $bboxXdelta$ values, we follow the method described by Zwicker et al.[24], setting directional derivatives to the slope of the ellipse's axes, and plugging those constraints into the original ellipse equation. The following simplified formulas determine a skewed bounding box for the user-specified cut-off ellipse:

$$\begin{aligned} maxU &= r\sigma_u \\ slope &= \frac{1}{b} \\ bboxXdelta &= r(\sigma_s^2 + b^2\sigma_u^2) \end{aligned}$$

4.3 Rendering Stereoscopic Images from Filtered EPIs

Now that we have a filter that we can manipulate, we must apply it to the light field to generate a pair of stereoscopic images. This requires three parts: convolving the correct subset of the light field, shearing the view direction so that the fields of view between the two images line up as in Figure 2.9, and pairing two perspectives with correct eye spacing.

4.3.1 Convolution

We orient our filter kernel parallel to the s-u plane, centered over each corresponding area of the light field in order to generate pixel color values in the final view. This convolves the filter over a slice of the light field, by placing a filter over the EPIs, as in Figure 2.10.

Each scanline of the final image is generated from convolution of a single scanline in the EPI. Each interpolated pixel color $\bar{u}$ is determined by:

$$\bar{u}(s, t) = \sum_{\forall u} \sum_{s=-\frac{w_f}{2}}^{\frac{w_f}{2}} f(s, u) imageCube(s, t, u), \bar{u} \in (r, g, b)$$

The u represents the specific image in the image stack, and (s, t) is the current pixel of the final image being generated. The function $f(s, u)$ is the Gaussian function defined in equation 4.3. And *imageCube* returns the corresponding RGB values from the s-t-u space illustrated in Figure 2.3. We use w_f to represent the width of the filter's bounding box. (Although the double summation denotes that $w_f \times u$ values will be summed together, similar to using a traditional bounding box, I am just simplifying notation.)

As the filter elongates in the u-direction, more images are combined in the final image. This means the range of depths in focus decreases. In photography, this corresponds to increasing the aperture. Also, as the filter widens along the s-direction, artifacts which come from interpolating between viewpoints that are too far apart dimin-

ishes. Widen the filter too much, then detail at the focal depth no longer appears crisp. The minimum sample interval required to capture the full range of frequencies in the images is beyond the scope of this paper, but can be found in the Chai [4] and Stewart [16] papers.

4.3.2 Varying the Parallax Axis

The application also changes the parallax axis of the scene to the focal plane. When a focal distance is given, that depth plane automatically moves into the plane of the screen. Setting the parallax axis in this plane means objects that intersect with the focal plane will appear stationary. Anything behind the focal plane will appear to move in the same direction as the viewer's movement, while objects in front will move in the opposite direction. This recreates the feeling of walking around a focal object instead of walking past a scene without fixating on a specific focal point.

To achieve this effect, the viewing frustrum must be sheared to match the slope of the focused pixels in the 2D slice. Since we know the original viewing frustrum of the acquired images is perpendicular to the plane of cameras, we can apply the same formula as the above section to find the leftmost pixel for each image. In order to change the focal center of the camera viewing frustrum, we must choose the subset of pixels in the scanlines that fall in this field of view. Whatever pixels lie inside this slanted viewing frustrum are used to determine which pixels are part of the convolution (Figure 2.10).

4.3.3 Parallax versus Depth with Stereo

One constraint that relates parallax depth perception to stereo depth perception is the equality between images seen in differing eyes in two viewing positions. Fig 4.4 illustrates this equality. From position A, the user's right eye should see the same image as the left eye when the user is at position B, (ie. when the user has shifted his position slightly to the right). This aligns the B_{left} image with the A_{right} image. Since we know the average eye spacing to be 60 mm apart, we can scale our walkable range to this.

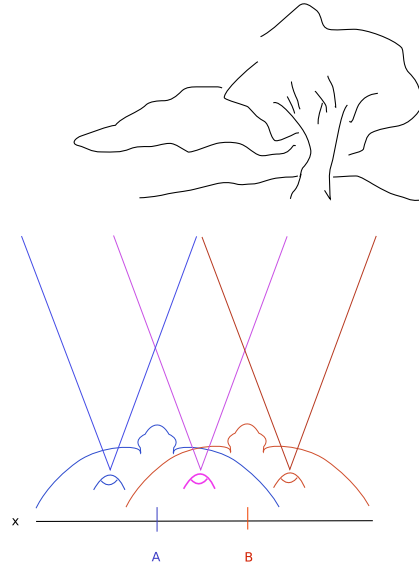


Figure 4.4: This will show a diagram of the two head positions and the relating images seen in each eye.

The number of images divided by the walkable distance will give us the real distance between input images. Since the distance between images in the light field is 1, we can convert the real distance between eyes into the light field space by dividing by the real distance between images.

Alternatively, we can scale the change in head movement. This means the range of movement that generates new views could be greatly diminished. Instead, we can change the range of images used by increasing the interval between the sampling camera positions, but this may lead to ghosting artifacts. In any case, these methods allow us to control the amount of parallax sensed by the viewer. The exact value often depends on the subject of the scene, the viewer's prior knowledge of an object's absolute size. Also the fundamental matrix can be calculated and actual distance from users can be determined to decide the amount of parallax that should be present.

There is still a scaling factor that decides just how large objects are in the scene. However, this can be decided by other constraints, such as the number of input images given and the screen width.

4.4 Interactive Focus

So far we have assumed a focal plane, i.e., the slope of the Gaussian filter, is given. However, the plugin allows a user to select a pixel to quickly change focal planes, versus manually changing focus with a dial. To do so, we must find the focal plane that intersects the depth of the selected pixel.

Determining the depth of a user selected pixel is done by finding the filter orientation that returns an image with minimum color variance over the area of interest. This requires two steps – comparing a range of filters, and deciding the area of interest.

4.4.1 Determining Filter Orientation

As described earlier, we want to find the slope of the line made up of similarly colored pixels in our EPIs. In order to determine its slope, we must match a filter of similar shape and orientation. We create many filters with the same σ_u and σ_s , but varying shear values b . To choose the shear that best matches up with the slope of the line is similar to finding the filter that minimizes the total variance of the pixel values inside the filter kernel. Minimum variance of zero means the pixel values are the same, ie. the same color. This is illustrated in Figure 4.6. Variance of a single pixel in the area of interest is defined as follows:

$$Var(s_f, t_f) = \sum_{\forall u} \sum_{s=-\frac{w_f}{2}}^{\frac{w_f}{2}} f(s, u) (imageCube(s_f + s, t_f, u) - \bar{u}(s_f, t_f))^2 \quad (4.6)$$

We represent each ordered image in the imagestack of Figure 2.3 in terms of its u position, and the user-specified focal point as (s_f, t_f) , while $s \in s_f \pm \frac{w_f}{2}$, and $f(s, u)$ gives the weight from the Gaussian function. The variance of every pixel in the area of interest is then averaged.

Variance comparisons are done by stepping through a range of shear transforms. We begin with a shear of zero, find the variance, increase the value by one, and iterate until the maximum shear value is reached. This maximum is a user-defined constraint

found in the configuration file. Each time a smaller variance is found, that shear is saved. Unfortunately, stepping through every possible shear is necessary, since there could be local minima. These minima exist in scenes that have transparent or reflective surfaces. In these areas, pixels could have multiple legitimate focal distances. These possibilities have been left for future improvements since stereo eye gaze would be needed to deduce which depth the user has fixated upon. We must disregard variances of 0 because these

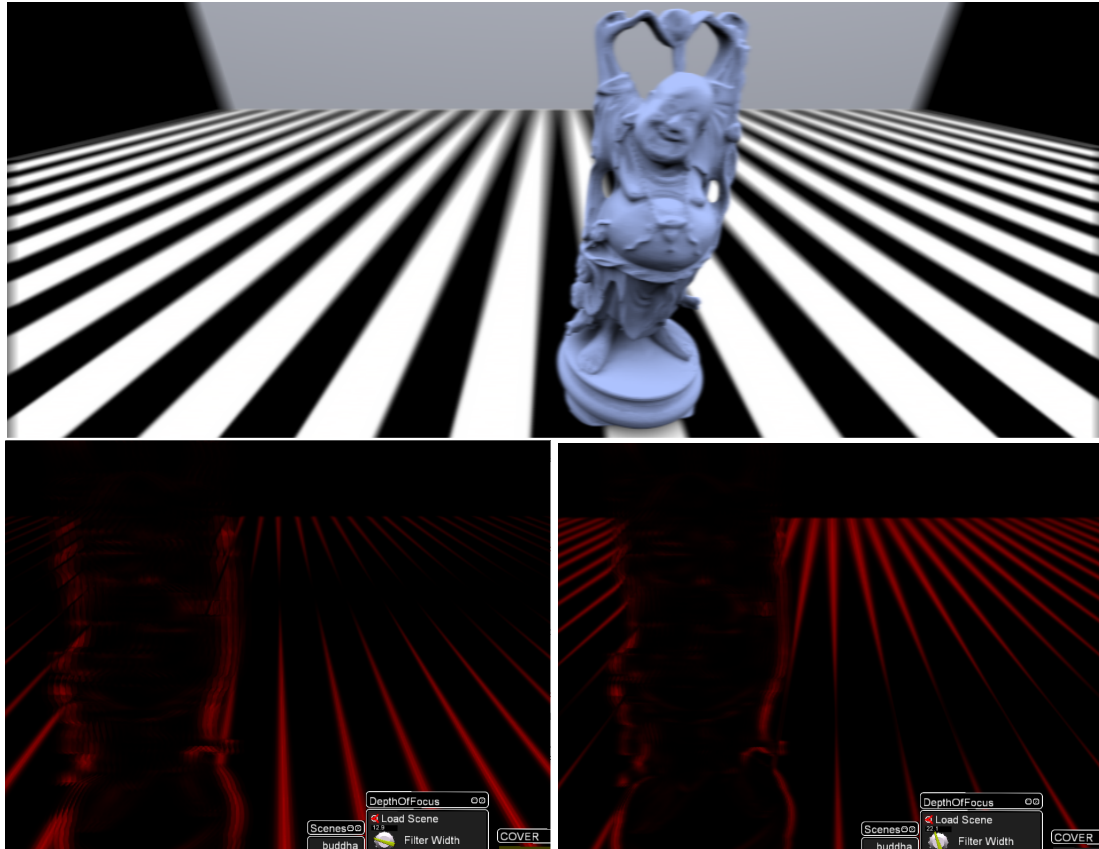


Figure 4.5: This shows variance computed over an entire image for the original scene, top. The left variance image is calculated for a filter shear of 12 pixels. The right variance image is done with a shear of 22 pixels. Note the large number of black pixels which represent zero variance in areas without strong edge features.

usually indicate the filter has been centered in an area devoid of adequate detail to determine an accurate slope. This is especially problematic in simplistic computer generated scenes where pixel colors might be too uniform. If the filter is not centered close enough

to an edge feature, many differently sheared filters will have zero variance, so that we cannot find a uniquely sheared filter through minimum variance. This can be seen in Figure 4.6 and 4.5 where black pixels represent variances of zero.

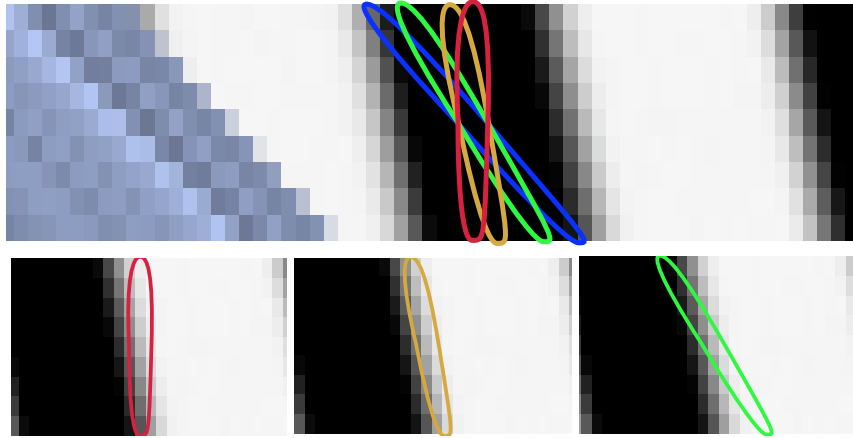


Figure 4.6: In areas of the EPI with uniform color, multiple orientations can have a minimum variance of 0. By ignoring zeroed variance, the blue orientation will be found, but this is still the wrong orientation. To get a unique orientation that corresponds to the area, we check for variance near edge features, so that the values between filters are more dissimilar and the correct orientation (middle) can be found.

4.4.2 Determining the Search Area

We determine the image patch for testing variance with a simple edge detector. Ideally, this patch would encompass pixels from the same object at the same depth. The edges found would be the boundaries of the object.

Because large contrast in color or texture is also necessary for identifying the depth robustly, at least two edges should fall in the relevant pixel patch. Figure 4.6 shows how robustness can be compromised without an edge feature. A simple one threshold edge detector is sufficient to determine two edges, to the left and right of the chosen focal pixel. In particular, this method ensures that background depth can be identified accurately. If only one edge pixel is used to determine the depth, the foreground depth will always block the background depth from being found with the minimum variance

method. Only when pixels between the two edges are averaged will the background depth be the found minimum variance. Although the edge threshold actually depends on the data set, I have found 0.0125 works well across the data sets used for testing.

Chapter 5.

Implementation

The highest priority of our implementation was achieving seamless realtime performance. This was counterbalanced by the need for robustness when calculating the user selected depth. Other factors that weighed in were the resolution of the final rendered image, minimization of the number of input images, and accuracy of the depth and parallax perceived compared to the original scene. This chapter details the implementation decisions of the plugin based on these priorities and any hardware or software compatibility issues that arose from integration into COVISE and translation of code into GLSL for fast filtering computations.

5.1 Scene Graph

The main graph of OpenCOVER has a root node with three children, an Objects root, a Scene root, and a Tracker root. For this plugin, we only want the image pixels to change value, not position. This means we do not want the entire image canvas to shift, based on user movement. In order to keep the entire image canvas from shifting due to headtracking, we attach our application's root node to the scene node, which is not affected by the tracker nodes. This way headtracking values are still updated, but the plugin's onscreen image does not move.

Although the plugin requires only adding one Geode node, its associated Geometry class variable is set to a StereoGeometry, a type created just for this application. StereoGeometry encapsulates all the shader interaction for rendering and determining pixel depth. The class also inherits from OpenSceneGraph's Geometry in order to override OpenSceneGraph's Geometry drawImplementation function with my own OpenGL calls. This allows control over which perspective to render, which shader to run, which buffer to write and read from, as well as which shader variables to update.

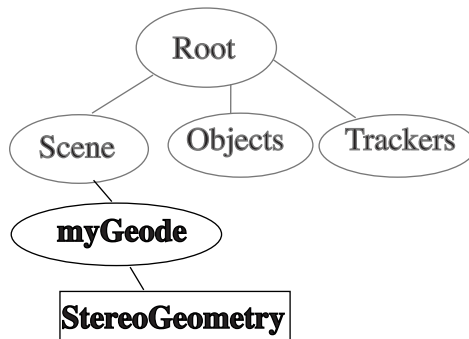


Figure 5.1: This is the main scene graph. The new nodes associated with this plugin are in bold.

5.2 Initialization

Initialization of the Geode and StereoGeometry occurs when the user selects an entry from the load submenu. This ensures that the plugin will not affect the performance of other plugins if the user has not entered this plugin's submenu. Inside of the main plugin class called DepthOfFocus, the function menuEvent is called and instantiates a new ImageSet object. This ImageSet encapsulates an ordered array of images that represents the image cube. If an image set was loaded before, the previous one is first destroyed. A new StereoGeometry is then created and initialized with the ImageSet. The first image of the data set becomes the default texture. Two fragment shaders, depthFilter and varianceFinder, are created, loaded, compiled, and linked. All shader

textures are initialized on the GPU at this time, too. Only the filter parameters and the user's position are yet to be initialized in the shader.

5.3 Loading and Updating Filter Parameters

Filter parameters are updated on the GPU every time before a new frame is rendered inside StereoGeometry's drawImplementation function. The filter parameters passed as uniform variables are M , the filter's corresponding matrix, $bboxXdelta$, the number of pixels in one row of the filter kernel, $slope$, the inverse slope of the filter's longer axis, and $boundError$, the r value of the isocontour ellipse used to determine the bounding box. Note that the $slope$ value is actually the inverse to avoid NaN errors when the slice lines are vertical. In addition to these parameters, the horizontal resolution of the acquired images is passed as $width$, and the current position of the viewer is passed as $viewerU$. The data is transferred to the GPU through a series of glUniform calls. The $wbsbE$ variable packs in the given order $width$, $bboxXdelta$, $slope$, and $boundError$ into one uniform vector. This snippet of code from StereoGeometry's drawImplementation sets the filter parameters in the depthFilter shader:

```
//update uniform variables of shader
paramLoc = myGLSL->glGetUniformLocation(depthFilter, "M");
float* M = StereoGeometry::mFilter.getM();
myGLSL->glUniformMatrix2fv( paramLoc, 1, GL_FALSE, M);

paramLoc = myGLSL->glGetUniformLocation(depthFilter, "wbsbE");
myGLSL->glUniform4f( paramLoc, (float)width,
                    StereoGeometry::mFilter.getBboxXdelta(),
                    StereoGeometry::mFilter.getInvSlope(),
                    StereoGeometry::mFilter.getBoundError());

if (right){          //setting right eye image
    paramLoc = myGLSL->glGetUniformLocation(depthFilter, "view");
    myGLSL->glUniform2f( paramLoc, viewerU + EYE_DIST, viewShear);
}
else {               //setting left eye image
    paramLoc = myGLSL->glGetUniformLocation(depthFilter, "view");
```

```

myGLSL->glUniform2f( paramLoc, viewerU - EYE_DIST, viewShear);
}

```

Determining right or left eye is done by identifying which machine is running the code with a COVISE call, `coVRMSController::instance()→isMaster()`. However, if two screens are run by the same machine, the `RenderInfo` instance passed into the `drawImplementation` can identify which eye to render.

5.4 MouseClick Event

When in `EnableDOF` mode, a mouse click on the scene changes the focal plane to the depth of the pixel. The mouse click is caught inside of the `preFrame` function of `DepthOfFocus`, and the following code is executed once before each frame is rendered:

```

if(enableDOF->getState()) {
    float updatedShear = canvas->findDepth(clickS, clickT);
    filterShearPoti->setValue(updatedShear);
}

if(imageset->isValid()){
    stereogeom = myGeode->getDrawables(0);
    stereogeom->updateFilter(scalingPoti->getValue(),
                           filterShearPoti->getValue(),
                           filterWidthPoti->getValue(),
                           filterHeightPoti->getValue());
    stereogeom->updateImages(imageset);
}

```

The function `updateImages` loads the set of textures onto the GPU if not already loaded. This only occurs when the user tries to load a new data set in order to render each frame efficiently enough for realtime performance.

The image processing is done in three main phases, determining the patch borders, calculating variance values, and comparing variance values. First we find the left and right edges of an image patch with a simple edge detector:

```

RGB pc = currentSet->getPixel(clickT, clickS, viewerU);
float prevColor = (0.3 * c.r) + (0.59 * c.g) + (0.11 * c.b);

for (int i=1; i<end; i++){
    RGB c = currentSet->getPixel(clickT, clickS+i, viewerU);
    currColor = (0.3 * c.r) + (0.59 * c.g) + (0.11 * c.b);

    if (pow(currColor - prevColor,2) > EDGE_THRESHOLD){
        rightEdge = i; //the pixel offset from the clicked pixel
        i = end; //end loop
    }
    prevColor = currColor; //compare adjacent pixel
}
//repeat for left edge

```

Then we set the current shader program to the varianceFinder, and test out different depths by changing filter shear and the corresponding uniform variables while keeping track of the minimum variance found so far. The following code in StereoGeometry which updates the filter variables in the shader is used inside a loop over the range of shear values, similar to the code in drawImplementation to update the depth-Filter shader uniform variables:

```

GLint paramLoc;
paramLoc = myGLSL->glGetUniformLocation(varianceFinder, "view");
myGLSL->glUniform2f( paramLoc, viewerU, 0.0);

StereoGeometry::mFilter.setShear(-1.0 * testShear);
StereoGeometry::mFilter.initVars();
paramLoc = myGLSL->glGetUniformLocation(varianceFinder, "M");
float* M = StereoGeometry::mFilter.getM();
myGLSL->glUniformMatrix2fv( paramLoc, 1, GL_FALSE, M);
paramLoc = myGLSL->glGetUniformLocation(varianceFinder, "wbsbE");
myGLSL->glUniform4f( paramLoc, (float)width,
                    StereoGeometry::mFilter.getBboxXdelta(),
                    StereoGeometry::mFilter.getInvSlope(),
                    StereoGeometry::mFilter.getBoundError());

```

Then the calculated variance values of the image patch are rendered to an off-screen buffer, GL_AUX0, and read back into the CPU via a glReadPixels call. The

average of these values is then compared with the last minimum variance found. Rendering at the correct resolution and texture offset is essential to reading back values correctly and efficiently. Code for rendering to the auxiliary buffer and reading the data back in is given below:

```
GLfloat glsRasterPos[4];
// Save and set viewing matrix states:
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
glOrtho(-1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f);
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glLoadIdentity();

// Store raster position:
glGetFloatv(GL_CURRENT_RASTER_POSITION, glsRasterPos);

// Draw image:
// origin is bottom left corner of output window
glRasterPos2f(-1.0f, -1.0f);

//draw aux canvas
//Choose portion of processed img to avg thru texcoords
if (currBuffer == variancebuffers[0]){
    glBegin( GL_QUADS );

    glTexCoord2d(1.0*(clickS - (patchWidth>>1) - 1)/width,
                 1.0 - 1.0*(clickT + (patchHeight>>1))/height);
    glVertex2f(-1.0, -1.0 + 1.0*patchHeight/height);

    glTexCoord2d(1.0*(clickS + (patchWidth>>1) + 1)/width,
                 1.0 - 1.0*(clickT + (patchHeight>>1))/height);
    glVertex2f(-1.0 + 1.0*patchWidth/width,
                 -1.0 + 1.0*patchHeight/height);

    glTexCoord2d(1.0*(clickS + (patchWidth>>1) + 1)/width,
                 1.0 - 1.0*(clickT - (patchHeight>>1))/height);
    glVertex2f(-1.0 + 1.0*patchWidth/width, - 1.0);

    glTexCoord2d(1.0*(clickS - (patchWidth>>1) - 1)/width,
                 1.0 - 1.0*(clickT - (patchHeight>>1))/height);
```

```

        glVertex2f(-1,-1);
        glEnd();
    }

    // Restore raster position:
    glRasterPos4fv(glsRasterPos);

    // Restore matrix states:
    glPopMatrix();
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    glMatrixMode(GL_MODELVIEW);
    glFinish();

    glReadPixels(0, 0, patchWidth, patchHeight,
                 GL_RGB, GL_FLOAT, variances);

    float avg = 0.0;
    for (int i=0; i<patchHeight*patchWidth; i++){
        avg += variances[i*3];    //shader wrote variance value to red channel
    }
    avg = avg/(patchWidth*patchHeight); //normalizing avg

```

Once all depths have been stepped through, the shear that returned the minimum variance is saved as a StereoGeometry class variable called `currShear`. The filter parameters of StereoGeometry's filter variable, *mFilter*, are then updated with an `initVars` call. These values will be updated in the `depthFilter` shader as part of each frame update. The draw buffer should also be reset to the default and the `depthFilter` shader should be re-enabled.

5.5 GLSL Shaders

The biggest performance boost comes from removing the pixel filtering computation from the CPU, and reassigning it to the GPU. This requires deciding which parts of code to translate into shaders and when to initialize the shader variables.

Since a GPU does computation on a per pixel basis and only handles certain

data manipulation more efficiently than the CPU, we just parallelize filter-image convolutions onto the GPU. The pixel computation occurs in two separate parts of the application, so a fragment shader is implemented for each portion. One filters the light field (depthFilter), and the other finds the variance of a clicked pixel (varianceFinder). Filter parameters are determined outside the shader, then passed in as parameters, since they stay the same for all pixels.

The following shader code is executed per pixel on the display:

```
vec4 sumColor = vec4(0,0,0,0);
float sum = 0.0;
vec2 soffset = vec2(-slope*viewerU/width, 0);
float startS = ((8.0 + viewerU)*slope) - float(bboxXdelta)/2.0;

for each image (TextureUnitN):
    float startSN = startS - N*slope;
    for (float w=0.0; w<bboxXdelta; w++){
        curroffset = vec2((startSN + w) * slope)/width, 0);
        float wtN = filterValue((startSN + w), (float(NUM_CAMS/2) - N + viewerU));
        sumColor += wtN * texture2D(TextureUnitN,
                                   gl_TexCoord[0].st + curroffset + soffset);

        sum += wtN;
    }
gl_FragColor = sumColor/sum;
```

Although Chapter 4 discusses finding the height of the sheared bounding box by using *maxU*, our implementation uses a constant height of 16 pixels. This is the maximum textures allowed for a fragment shader program on the Nvidia cards and is denoted by NUM_CAMS. N represents each image that is loaded in GPU texture memory. In our implementation, $0 \leq N \leq \text{NUM_CAMS}$. Adding if-statements into the shader to decide which range of images to weigh into the convolution would be less efficient than simply applying each image, so we use a constant height for the filter kernel. This allows us to unroll the image (TextureUnitN) loop.

Instead of displacing the filter, we displace the images with texture coordinate offsets so that the current pixel being rendered is centered on the filter. In order to do this, we use the original *width* resolution of the image set to determine the correct

decimal offsets. This was passed in as the first entry of the uniform variable *wbsbE*.

The *soffset* determines at what pixel in a row of an EPI to begin filtering. This is similar to shearing the camera frustum, allowing the pivot center of the parallax to match the focused depth. This mimics walking around an object, while centering your gaze on an object instead of just looking straight ahead into a scene. By shearing the view frustum, sometimes the sides of the image will go beyond the bounds of the image data (Figure 2.10), producing areas where horizontal lines are seen, since the OpenGL textures are set to clamp and not wrap around. This can be fixed by shrinking the number of pixels across, but often these artifacts do not show up. Only if the view frustum is sheared to an extreme, will this affect the final image.

For each offset of the filter in the EPI, we multiply each pixel in the filter kernel in the following way. The *startSN* finds the leftmost pixel that lies inside the sheared filter kernel while the *w* steps through the rest of the pixels in the current row. The variable *wtN* is the weight of the current pixel, calculated by the Gaussian function *filterValue*.

Once all the pixel values have been added together in *sumColor*, it is divided by the total weight, *sum*, to normalize the final value. This takes care of boundary cases where the filter kernel may fall partially outside of the 2D image data. A noticeable side effect from this occurs when the user moves too far to the edges of the perspective range. The focal depth diminishes until only the first or last of the original captured images is displayed. This can be easily fixed by decreasing the width resolution of the final rendered images.

The variance is calculated similarly, passing in the same uniform variables and first finding the average by running the above shader code, followed with this:

```
for each image:
    startSN = ((float(NUM_CAMS)/2.0 + viewerU - N)*slope) - float(bboxXdelta)/2.0;
    curroffset = vec2(startSN/width, 0);
    float wtN = filterValue(startSN, ((float(NUM_CAMS/2)) + viewerU));
    for (float w=0.0; w<bboxXdelta; w++){
        curroffset = vec2((startSN + w)/width, 0);
        wtN = filterValue((startSN + w), ((float(NUM_CAMS/2)) + viewerU));
```

```

pColor = texture2D(TextureUnitN, gl_TexCoord[0].st + curroffset);
var += pow(pColor - avg, vec4(2.0, 2.0, 2.0, 0.0)) * vec4(wtN, wtN, wtN, 1.0);
}

```

This code steps through each pixel inside the sheared filter kernel to sum up the squared difference between the actual value and the average value found, following the definition of variance in section 4.4.1. Normalizing the variance did not seem to change the precision of the variance finder, so normalization was skipped.

Instead of inefficiently passing every pixel weight to the shader as a uniform variable, we calculate them on the GPU via the passed filter parameters. The weight of each pixel for both shaders is determined by a function inside the shader called `filterValue`. This takes an (s,u) coordinate as an argument. It then uses the M and the elliptical filter definition in Chapter 4 to determine the weight. Note that the normalizing factor in Equation 4.1 is unnecessary because normalization is done after the pixels in the kernel have been summed up. The following code implements the function:

```

float filterValue(in float s, in float u){
    vec2 p = vec2(s,u);
    vec2 pM = p * M;
    float pMp = dot(pM, p);
    return pow(2.718281828, -pMp);
}

```

5.6 Workarounds

Numerous issues were encountered when porting my code to the COVISE and OpenSceneGraph environment. Changes and updates to software, hardware, and their configuration accounted for many difficulties. Some changes were to accommodate my plugin, limited by the original setup.

I had some misunderstandings about the way OpenSceneGraph handles its textures. Because of this, I hoped to update the texture quickly through a `preFrame` call. Unfortunately, `preFrame` did not have enough access to the lower layers of abstraction and OpenGL to allow pushing alternating textures to screen from two `preFrame` calls.

Instead, one OpenGL flush is called per frame, after processing both calls to `preFrame`. This did not allow two different images to render simultaneously to the screen.

Initially, the hardware configuration was such that I needed to create separate Geometry objects and render a different texture of each perspective to each Geometry. Eventually, I created my own Geometry implementation that could handle two textures. The corresponding texture would be pushed to the display depending on which screen was currently being rendered. This would work with the `HORIZONTAL_SPLIT` stereo mode. If the configuration did not allow this (in `LEFT_EYE` or `RIGHT_EYE` stereo modes), then I would introduce a static boolean or integer variable to switch off rendering between left and right textures. However, this method would only work for configurations that use one machine to control two projectors.

When inheriting from the class `Geometry`, the `drawImplementation` function can encapsulate any direct calls to OpenGL. Since this is `const` typed, no class variables could change value inside this function. This kept me from keeping certain filter values as class variables. Instead, I used `glUniform` calls to change values inside the fragment shader. This inevitably led to putting more functionality and computation inside the shader than originally intended. This meant some simple arithmetic had to be done at the beginning of the shader, to initialize variables used later. Since shaders run faster when fewer variables need to be looked up, this implementation choice could have slowed the performance of the shader. However, this was necessary in order to correctly render a different image from each projector for each screen.

One of the pitfalls of working with `OpenSceneGraph` is working directly with OpenGL instead of built-in functions of `OpenSceneGraph`. When the functions do not provide the exact behavior or access needed, these pitfalls become relevant. The biggest problem I encountered was getting the plugin to work with machines with two unbridged graphics cards. I would suggest working with `OpenSceneGraph`'s functions to fix initialization errors.

Since GLSL does not allow variable sized arrays, I was unable to write an efficient shader that could benefit from variable height for the bounding box. If I incorpo-

rated this parameter, extra if-statements would be required before each texture operation. Since more textures for the GPU means a larger range of images that can be generated, I used the maximum number of texture handles supported by the graphics card. Instead of checking for the vertical bounds on the bounding box of the filter, each image was weighted into the final image, giving a simpler, faster implementation.

Working with Nvidia graphics cards on the Linux platform presented more challenges. Although certain functions were listed as supported, I would run into segmentation faults when a valid function address could not be linked at runtime. For example, the Nvidia drivers for Linux did not support `glUniform1f`, despite being listed under available OpenGL extension functions. Because of that, I decided to pass variables of the same type as a vector to the GPU, using `glUniform4v`.

Also, framebuffer objects were not implemented for the Nvidia Linux drivers for the graphics card on the lower end developer machines, so I had to write my GPU computed variance data to a different buffer. I ended up drawing to an auxiliary draw buffer provided by OpenGL and using `glReadPixels` to retrieve the data.

Chapter 6.

Results

This section gives an overview of the range of parameter values used in this plugin and the results they achieved visually and performance-wise. We will break down the runtime bottlenecks and tweak the program parameters according to these limitations.

6.1 Performance

The average frame rate for this plugin was 5.0 fps. Without user movement, refresh rate could be as fast as 5.3 fps while viewing the train data set of resolution 1920 x 724. With the user walking in front of the screen while changing focal parameters, this rate could dip down to 3.9 fps when displaying the patio or buddha data set at a resolution of 1920 x 760. We used 16 images for all data sets, and filter parameters of a boundError of 3.0. The viewer's position, the focal plane, and filterHeight do not affect the performance. This was measured on the 2.6GHz AMD Opterons and Nvidia GeForce 7900s with memory bandwidths of 42.2 GB/sec [20].

The main bottleneck of the shader-driven plugin is the number of pixels in the final displayed image. Specifically, the time complexity can be described as $O(\text{number of pixels in output image} \times \text{number of pixels in filter kernel})$. This is because for each

output pixel, we average every pixel in the filter kernel with the desired Gaussian weight function. Since the number of output pixels easily exceeds the number of pixels in the filter kernel, the output resolution becomes the main restriction on the performance of the image filtering.

To demonstrate the direct link between output resolution and render time, we tested different resolutions in realtime. As the resolution of the StereoGeometry decreased, the frames per second increased, directly proportional to this. However, assuming that users always want the widest field of view possible, we automatically fitted our output image to the width of the screen.

By executing the image filtering on hardware, we accelerate the rendering frame rate but also become limited by the number of pixel calculations that the graphics card can parallelized. One of the graphics card is able to parallelize calculations for a maximum of 24 pixels per clock cycle. Newer graphics cards do not have an upper bound on pixel parallelization, so they will be able to achieve even better performance.

Another bottleneck of the shader comes from updating its uniform variable values with tracking information. Without any head movement, a quad takes 79 milliseconds to render. With head movement, the quad takes 82 milliseconds.

In order to make the frame rate highest when an image set is shown on screen, we preload our images. Disk retrieval takes the longest time of all functions in the plugin. However, this time doesn't affect the frame rate when an image set is visible on the display. Loading an imageset takes approximate 3 seconds. This includes the time to copy the image from the CPU to the GPU texture memory.

6.2 Sampling

We want to maximize the horizontal walkable distance that generates new views of a scene. To do so, we need the outer images to be captured from as far from each other as possible. The more sample images of a scene we have, the greater this distance. However, we are hardware-limited to 16 image textures per fragment shader program,

so we must choose a sampling distance between the views. If the disparity between the images is too great, many ghosting artifacts cannot be alleviated with filtering. Some minimum values are determined in the papers [4] and [16]. In the end, every third image is used from the original data set without pronounced ghosting. The original distance between the camera positions is unknown.

6.3 Filter Parameters

After writing a prototype application in C++, using the Qt 3.0 graphics library [22], I found the effect of increasing or decreasing certain parameters. From this information, I determined optimal ranges for filter values and scene sampling. Fine tuning some of these parameters was then done through dials, which were implemented in the menu system to change each parameter in realtime.

Filter width affects performance time, sharpness at the focal plane, and the amount of ghosting artifacts. As filter width increases, ghosting artifacts decrease, but sharpness at the focus also decreases. The best value found for this parameter was 0.8 pixels wide.

Filter height affects the focal length of the final image, but not the refresh rate. A filter height of at least 1.5 pixels is best in order to notice the depth of field effect. The larger this value, the shallower the focal depth. This can be seen in Figure 6.1.

The shear of a filter affects the focal plane of the final image. Like filter height, it also does not affect refresh rate. The smaller the shear, the further into the scene lies the focal plane. This is illustrated in Figure 6.3. However, when focusing on a depth very close to the viewer, large shears introduce artifacts from rectification on the borders of the final image. To minimize this onscreen, the final image canvas width can be decreased.

Since filters of different shears are iteratively used to test for depth on a patch of pixels, we can enhance performance by minimizing the number of filters used. Maximum range of shear values has been found to be from 0 pixels to 50 pixels. Since the

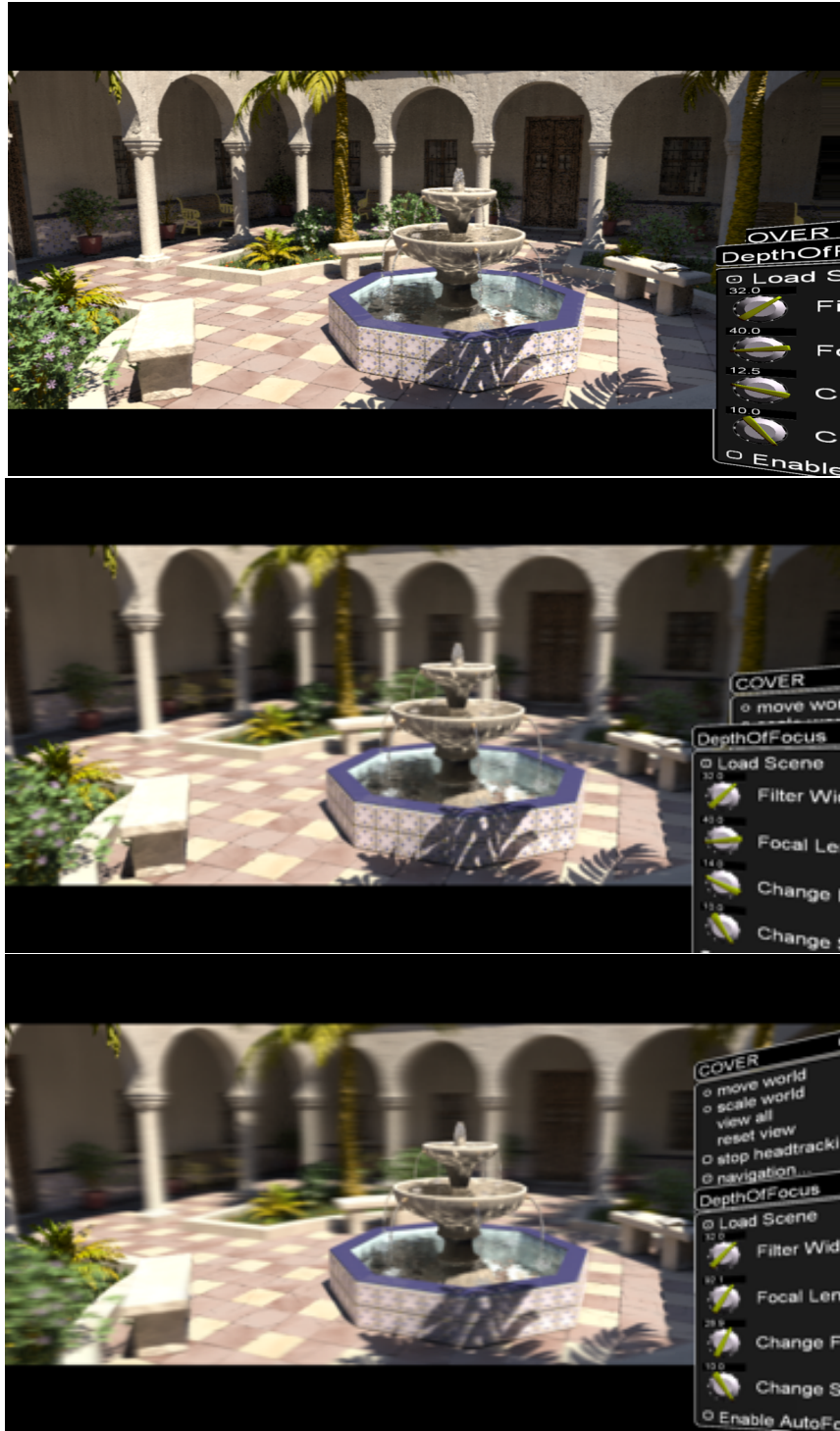


Figure 6.1: This shows different filterHeight (σ_u) values in increasing order. A shorter filterHeight corresponds to a longer focal depth in the simulated camera. In the extreme case, of filterHeight = 1, everything in the image is in focus, as seen in the top image.

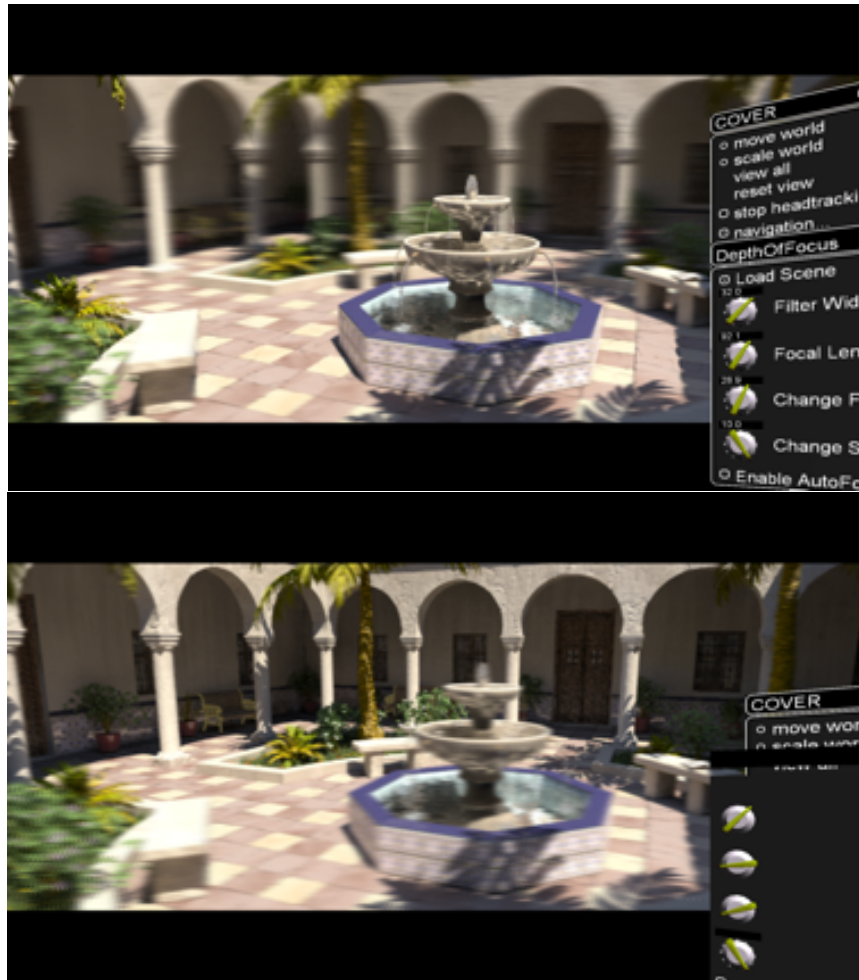


Figure 6.2: This shows different focal planes based on filter shear. The image on the left is focused on the fountain, so it has a larger shear than the image on the right, focused on the background tree.



Figure 6.3: This shows different focal planes on a different data set. Notice how the crispness of the text on the train is preserved in the lower image.

sampling of shears is done at an interval of 1 pixel, there are 50 filter comparisons total. The maximum value is based on the range of depth in the scenes and the distance between sample images used. This was simply found by using the dial to find the minimum shear value required to make every point in the scene out of focus. Intuitively, this is when the focal plane has moved behind the viewer. Also note that if the filter height is increased, a wider range of shear values is needed.

The *bboxXdelta* of a sheared bounding box affects the frame rate as well as robustness. Make the filter kernel cutoff too narrow, then many relevant pixels are unaccounted for, which makes the orientation of the filter less accurate. This translates into less accuracy in the depth detected by the filter. Make the filter kernel cutoff too wide, then too many near-zero weighted pixels are retrieved, slowing performance. I found that a filter bounding box of at least 3 pixels wide per row gives the most robust results.

6.3.1 Edge Threshold Parameters

For each scene, a different edge threshold is needed to find an important feature at the desired depth. The algorithm depends on variations in color to detect a depth, but noise in images could be picked up accidentally as a scene feature. Therefore, the edge threshold needs to be large enough to ignore noise, and small enough to identify important features.

The edge threshold is also used to decide the size of the area of interest. The height of the patch is fixed at 3 pixels, but the edge threshold determines the left and right cutoffs. This directly affects frame rate since each pixel in the patch runs through the variance shader. The more pixels that need to be processed, the longer the total patch variance calculation takes. Because of this, we do not want too large a value for the edge threshold.

To simplify the plugin, one threshold value is used for all scenes. An edge threshold of 0.0125 works best over all three datasets to create a small enough patch that will not impact performance and still determines the depth relatively accurate. However,

in areas with little color variation such as the sky in the train set or the back wall of the buddha set, a foreground depth is found instead. This is due to a strong edge from a foreground object that lies in the row of the selected pixels.

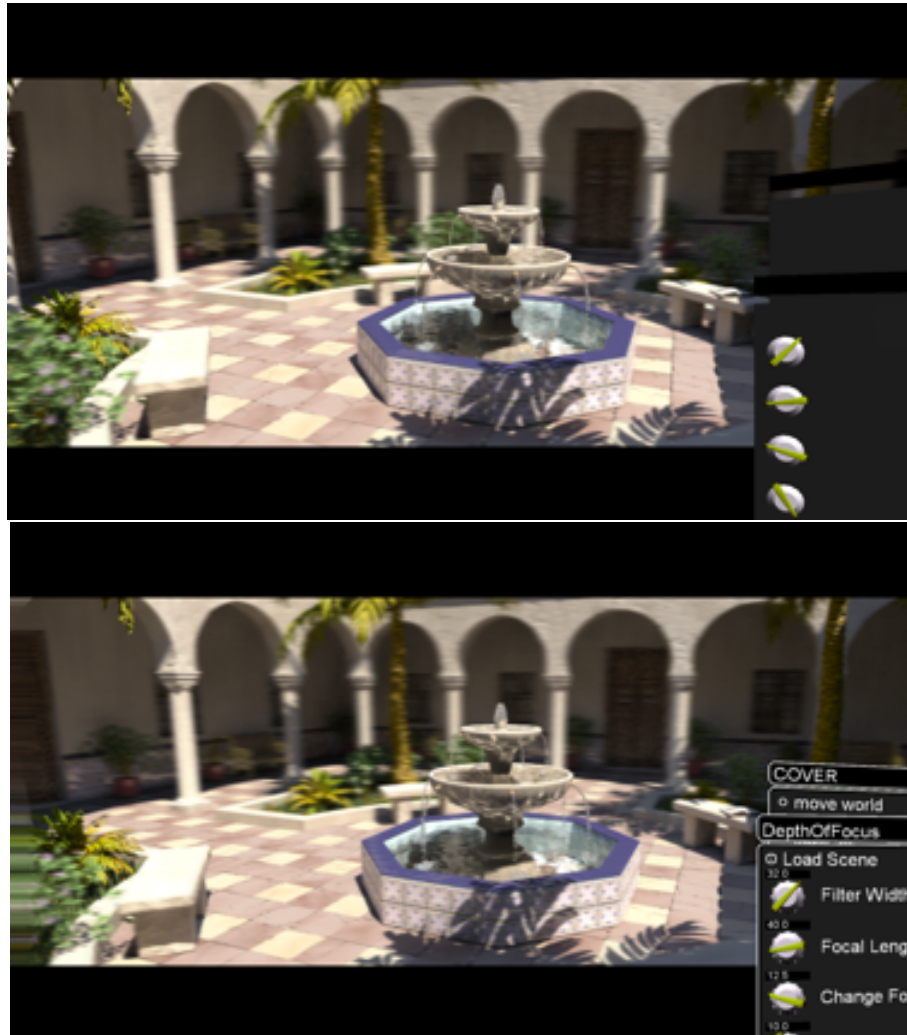


Figure 6.4: This shows two different view perspectives rendered with our algorithm. It shows the parallax axis as centered with the focal plane. The door right behind the center of the fountain moves from the left to the right of the spouting water. However, the bench on the left appears to move the opposite direction, from right to the left. The fountain lies in the plane of the parallax axis, therefore it does not appear to move.

Chapter 7.

Future Work

The original intent of this rendering algorithm was to use realtime video input with gaze-tracking devices to implement autofocusing on 3D autostereoscopic displays. Now that we have a robust application that can determine the depth of any pixel within a displayed image, and can create many views with different focus from images captured from a horizontal camera array, all of which are adjustable in realtime, we can easily extend our project to work with other trackers and video streams. Future work would be to render video streams acquired from camera arrays in realtime. Adding gaze-tracking or automatic tracking of objects of interest, such as faces, could continually adjust the focal plane as a means of autofocusing. By incorporating these aspects, we can use our algorithm for image compression to reduce bandwidth usage in video conferencing, and improve depth perception for remote operation.

Using our algorithm to prefilter video data before sending it over a network can decrease bandwidth usage without compromising the relevant detail for interactive video applications. By introducing depth of field into the rendered image, we can compress the output image due to the blurred areas of the image. The focal depth also isolates out distracting areas of the image, highlighting which part of the image viewers should concentrate on. This would be helpful in video conferencing and presentations when a face would normally be the object of interest.

Alternative ways of compressing images based on area-of-interest have used foveated blurring [14], based on human vision. We do not account for this in our implementation. However, we could add this additional step to our final image should there be pixels in our output image still in focus, though they are outside the user's area-of-interest. This would create an even more natural image, although the additional savings in compression would depend on the scene.

When operating remote controlled robots or surgical tools for minimally invasive medical procedures, horizontal parallax-only information may not provide enough freedom in viewing perspective. However, our system can easily be extended to work with vertical parallax. This is done by using an $m \times n$ grid of cameras as input. The viewer would be able to also see a different perspective when changing their viewing height.

By substituting headtracking for eye tracking, the focal plane can be constantly moved to the depth of the user's object of interest. This way, the remote operator can experience even more natural vision. Eye trackers would give the application enough information to match the viewer position and focus through continual updating.

Alternatively, eye trackers let users specify the focal depth without using their hands. This is important since during remote operation, hands are often holding tools. Remote operators can also benefit from the extra depth cues re-created by depth-of-focus. This would allow them to judge the 3D space better and not be confused or distracted by objects at different depths.

Another possibility with eye tracking data is eliminating focal plane ambiguity of single pixels with multiple depths. The distance from the focal plane to the user can be deduced and varied depending on where their eyes converge. For translucent, transparent objects, the foreground material or the background could become the focal point. The intersection plane of the two lines of sight can be deduced from the eye tracker data, and thus differentiate between focusing on the foreground or the background.

Since the performance bottleneck of the application is based on the output resolution, not the size of input data, we are able to extend this algorithm to many appli-

cations with additional information. Also, because the image processing can be parallelized across pixels, scalability is not an issue. Should higher resolution be desired, additional machines can be used to balance the pixel load to maintain the realtime aspect of the application.

Appendix A.

Implementation Details

A.1 Image Data Structures

One of the most important parts of making the application run in realtime is the structuring of the data and its flow through the system. By minimizing the amount of information written, or retrieved repeatedly, between disk and main memory, CPU memory and GPU memory, and even just between memory and caches, we can get large performance gains. A large part of this depends on the organization of the data so that an optimal size gets cached or read in blocks. We will look at each of the data types used in our implementation: Image, ImageSet, and Filter.

A.1.1 Image

The wrapper class, Image, was created to mirror the osg library class of the same name. This was meant to make porting between 3D environments easier. (However, this could have been the cause of problems with loading textures on two separate graphics cards on the same machine. These get initialized with OpenGL calls now, not with osg's calls.) The main difference from osg's Image class is that image data is stored as an array of RGBs, a class that encapsulates three floats r,g,b, for red, green, and blue information. This was originally meant to have an overloaded constructor to allow

changing many different pixel formats into my own type RGB. In the end, I only dealt with png files, so I only had to do the conversion once, inside the `initImage` function.

Class Variables

RGB* myImage This is a pointer to the dynamically allocated array of RGBs that holds all the pixel data of an image

int maxIndex This is the maximum value of a single channel. For example, an 8-bit png would have `maxIndex = 255`. This is used to convert input values to floats as well as convert internal type RGB back to the original input type. The last usage is more for debugging and comparing generated views with original views.

int myWidth This is the number of pixels in a row

int myHeight This is the number of pixels in a column

GLuint texID This is the `texID` generated by OpenGL from `ImageSet`. We need this for `glBindTexture` if we do a non-GPU version of `StereoGeometry`

Constructor

The constructor takes a string as the filename. This filename is then used to load the image by calling the `osgDB` routine `readImageFile`. The file is returned as a char array. Then the private helper `initImage` function converts and stores pixel data as triplet floats of type RGB. This is done in the `initImage` routine, which handles both 8 bit or 16 bit RGB. By using floats, we do not lose precision during filtering. This affects the robustness of finding the correct depth of the focused pixel.

The `Image` constructor takes a string for the filename, a number for the index for its corresponding place in the `ImageSet` array, and the OpenGL texture ID.

Initializer: `initImage`

The initializer function, `initImage`, for the `Image` class has three main jobs: initialize class variables that are image attributes, convert different input image data to our own RGB-type, and initialize the texture directly in OpenGL. The following code demonstrates our direct initialization of an OpenGL `texImage2D`. Since the textureIDs are generated in `ImageSet`, we need to designate our own of the `texID`. This value is passed in from `ImageSet` with the constructor. In this way, every `Image` in our `ImageSet` is quickly allocated and is guaranteed contiguous textureIDs. This is necessary for the shader computation later.

```
//Finally, initialize the textures directly in OpenGL for the shader
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, texID);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
...
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, myWidth, myHeight, 0,
GL_RGB, GL_UNSIGNED_BYTE, myOsgImage->data());
glDisable(GL_TEXTURE_2D);
```

Additional Functions

`bool isNull()` This function checks if `myImage` has been initialized yet.

`int load(const char* filename, const char* fmt)` This loads an image into memory by filename and file format. Then it calls `initImage` to update all the class variables to match this image. The return value is the `texID` that the texture gets bound to in OpenGL. (This function is called by `ImageSet`)

`RGB pixel(int x, int y)` This returns the pixel value as an RGB.

`bool valid(int pickS, int pickT)` This checks if the pixel (`pickS`, `pickT`) exists in this image.

`void clearImage()` This frees the data in the `myImage` array.

GLuint ID() This returns the texID associated with this texture. It is used to initialize textures on the GPU for shaders.

int width() This returns the image attribute, myWidth.

int height() This returns the image attribute, myHeight.

A.1.2 ImageSet

ImageSet essentially is an array of Images that corresponds to an image stack such as in Figure 2.3. It has some helpful accessor functions such as retrieving a pixel value in terms of the image stack subspace. However, most of these are not used except by the now defunct DepthProcessor. ImageSet also keeps track of sampling parameters that characterize the currently loaded image samples, such as interval between cameras and the number of cameras.

Class Variables

Image* dataset[SLICE_FILE_WIDTH] This is the array of Images used for interpolating views. The array size is chosen as the maximum number of texture units the graphics card can support for a fragment shader. For all the Nvidia cards I worked with, this value is 16. Note that this is just a subset of the original light field sample data.

double viewX This is the horizontal position of viewer when they are looking straight at the screen. An analogous viewY class variable is not necessary because we took horizontal parallax-only samples. With HPOs, change in viewing height will not affect the final rendered image.

double camInterval This specifies the "distance" between adjacent cameras. More accurately, it samples the set of ordered images with this interval. The default value can be quickly set in the config file as the CAMERA_DIST.

int imgCount This is always equal to SLICE_FILE_WIDTH, set in the config file. This was supposed to be variable. However, once the image processing was moved into a shader program, changing this value in runtime was too difficult to code without adversely affecting performance. In order to write a shader that performs the best, as many variables should be turned into constant hard coded values. This saves variable look up and branch decision costs. Also, to get the best results for image reconstruction, the more data samples, the better. Once shaders were used, the size of input data was no longer the bottleneck. Instead, the final image resolution posed a bigger performance bottleneck.

const char* fileset This string denotes the current loaded scene and is used to generate filenames to send to the Image constructor.

Constructor

The constructor takes the current viewer position, viewX as a parameter. It then initializes the camInterval to CAMERA_DIST and imgCount to SLICE_FILE_WIDTH as specified in the config file.

Initializer: load

The bulk of initialization is done inside the function load(const char* sceneName, int interval, int imageCount, Image* currentImage), called by DepthOfFocus. The main job of the ImageSet initializer is to generate the desired filenames of images. It then generates an array of texture IDs with the OpenGL call glGenTextures. These parameters are used to instantiate Images and load them into the array, dataset[], ordered by their camera positions. The filename depends on the sceneName and (sampling) interval. The imageCount parameter tells us how many images to choose from the original light field sample image set. Finally, the currentImage was intended to pass in the latest onscreen image if using the non-interpolation mode.

Additional Functions

bool isValid() This checks if the dataset array has been loaded with images yet.

void setViewPosition(double x, double y) This is used by DepthOfFocus in menuEvent to set the viewerU value, basically giving access to this parameter all the way down at the shader level.

RGB getPixel(int y, int x, int slice) This function indexes into the imagestack 2.3 when given the (s,t,u coordinates), which correspond to the (x,y,slice) parameters. and returns the pixel value as an RGB. It originally was used by DepthProcessor. Now, it's only used by StereoGeometry to find edges in EPIs to determine patch size.

int getWidth() This returns the number of pixels in a row of an input image.

int getHeight() This returns the number of pixels in a column of an input image.

A.1.3 Filter

This class holds all the data to uniquely define a sheared elliptical Gaussian filter as described in Chapter 4. It also precomputes the data needed by the shader to compute values of a filter and to efficiently step through the relevant pixels. The shaders need the following: matrix M , $bboxXdelta$, $boundError$, and slope. This is mostly computed from a shear value, scaling values in both x and y directions, and the bounding ellipse radius, described in Chapter 4. This decreases the amount of time spent in the shader to determine what pixels lie inside the filter's sheared bounding box, and to compute the filter value of these pixel.

Class Variables

double shear This represents the shear matrix entry b from Equation 4.4.

double ctrX, ctrY, maxY, maxX, minY, minX These correspond to the vertices values of the filter's sheared bounding box. Note that ctrX and ctrY are always = 0. Instead of changing the center of the filter, I offset my x,y values inside the shader code. This approach runs faster than recomputing the filter parameters and passing that to the shader. Because the filter stays centered, $\text{maxY} = -\text{minY}$.

double sigmaX This corresponds to the σ_u scaling matrix entry from Equation 4.4.

double sigmaY This corresponds to the σ_s scaling matrix entry from Equation 4.4.

Matrix2x2 M This corresponds to the final matrix computed in Chapter 4

double bboxXdelta This corresponds to the number of pixels per row inside the sheared bounding box.

double boundError This is the user specified value in Section 4.1

double invslope This is the computed inverse slope of the long axis of the filter. This corresponds to the focal plane position as discussed in Chapter 2.

Constructor

The constructor sets the class variables corresponding to the constructor arguments: shear, filtercenterX, filtercenterY, sigmaX, sigmaY, and boundErrorRadius. Then initVars function is called to initialize the rest of the class variables.

Initializer: initVars

This function initializes or "refreshes" the matrix, bounding box, and slope values of a Filter. It calculates all the class variable values. M is first calculated, then its corresponding slope. Finally, the vertices of the sheared bounding box are computed. It is called by the Filter constructor and StereoGeometry whenever a filter parameter is changed by the potimeters.

Additional Functions

float getInvSlope() This returns the invSlope value to pass to the shaders. We return the inverse, not the actual slope, so that we will not have any trouble with values going to infinity. Possible inverse slope values range from 0 to 50, depending on the range of depth in the captured imageset.

float getBboxXdelta() This returns the bboxXdelta value to pass to the shaders.

float getBoundError() This returns the boundError value to pass to the shaders.

float* getM() This returns the matrix M as an array of 4 floats in column-order to pass to the shaders.

void setSigmaX(double sigX) This is used as a hook to only change the sigmax value, used by the "Filter Width" potimeter in DepthOfFocus.

void setSigmaY(double sigY) This is used as a hook to only change the sigmay value, used by the "Focal Depth" potimeter in DepthOfFocus.

void setShear(double sh) This is used as a hook to only change the slope of the filter, used by the "Change Focus" potimeter in DepthOffocus.

A.2 OpenSceneGraph and COVISE Object Extensions

This section covers the objects that inherit from OpenSceneGraph. These are for interfacing with the 3D system. We will discuss how each one builds upon their parent object.

A.2.1 DepthOfFocus

DepthOfFocus is the main plugin class. It handles mouse button interactions with the main menus and scene objects. It inherits from COVISE's coMenuListener

and coValuePotiActor classes. This means filling in the functionality of the abstract functions, preFrame, menuEvent, and potiValueChanged.

Class Variables

ImageSet* imageset This is a pointer to the currently loaded image set.

osg::Geode* imageNode This is attached to the scene node and holds our StereoGeometry

Constructor

The ImageSet constructor is called by opencover before the plugin is selected by the user so very little should go in here. Only menus and menu objects are instantiated here. Therefore, all my potimeters and listeners are created in here.

Initializer: initGeode

The function initGeode is called after an imageset is loaded. It instantiates a new Geode and sets its Geometry to a new StereoGeometry object. This Geode is then attached to the scene graph at the scene root node.

Inherited Virtual Function

The function preFrame is inherited from coMenuListener, which registers up and cleans up user interface listeners. This includes determining what pixel a mouse clicked on, initiating the depthFinder, calling updateEyeImage when potimeters have been touched, or a new imageset has been chosen for loading by the user.

Additional Functions

void updateEyeImage() This updates the filter values and the imageset if the user selected or changed any menu values. It is called after an event occurs.

bool calcTransImageToPlane(image) This determines if a mouseclick occurred on the image.

A.2.2 StereoGeometry

This class inherits from the OSG class Geometry. By creating our own class extension, we can specify how to draw a Geometry. This is done through implementing the abstract function drawImplementation. This low level of access to the renderer was needed in order to get information on which screen was currently being rendered to.

Normally, COVISE handles the mapping of perspectives to screens for rendering. However, because this plugin is texture-based, no actual geometry changes between the two views. COVISE does not handle texture updates based viewing angle, so knowing this state, we can choose which perspective, left or right eye, to render. Inside of drawImplementation we define a static variable alternating between two values. This is used to render two different images to the proper screen from one machine.

A texture object is not necessary for creating this type of Geometry because we will be using the imageset to generate new textures on the fly. We will not go through OpenSceneGraph's OpenGL rendering setup. Instead, we will write our own OpenGL drawing calls inside drawImplementation. This is a simple glBegin with type GL_QUAD.

Class Variables

static Filter mFilter This holds the parameters of the current filter.

ImageSet* currentSet This is a pointer to the current ImageSet.

float viewerU This is the viewer's (horizontal) position.

GLuint depthFilter, gl_shader, varianceFinder These are the IDs created by OpenGL when initializing shaders

GLuint gl_vs, gl_fs, gl_fsVF These are the IDs created by OpenGL when initializing the shaders

GLint compiled, linked These are the IDs created by OpenGL when initializing the shaders

Constructor

This only needs a width and height to construct, while the imageset can be set later with the function `updateImages`.

Inherited Virtual Functions

The `drawImplementation` is where you can overwrite the renderer with your own OpenGL function calls. The majority of my rendering code is housed in this function.

Additional Functions

float findDepth(int x, int y) This calls the `varianceFinder` shader and returns the shear value that corresponds to the depth of the pixel at (x,y).

void updateFilter(float sc, float sh, float sigX, float sigY) This changes the values of the filter that are linked to potimeters.

void updateImages(ImageSet* currSet) This loads a new `ImageSet` and initializes class variables to match it.

Originally, we wanted to keep a viewing mode without depth of focus for comparison. In this case, a texture object should be used to keep track of the current images shown to be shown on screen. This helps performance since the textures do not need to be constantly refreshed if the user does not move far enough to change to the next available perspective image. Of course this is unnecessary if performance does not take a visually noticeable hit.

References

- [1] ADELSON, E. H., AND BERGEN, J. R. The plenoptic function and the elements of early vision. *Computational Models of Visual Processing* (1991), 3–20.
- [2] BRESNAHAN, G., GASSER, R., ABARAVICHYUS, A., BRISSON, E., AND WALTERMAN, M. Building a large-scale high-resolution tiled rear-projected passive stereo display system based on commodity components. A. J. Woods, M. T. Bolas, J. O. Merritt, and S. A. Benton, Eds., vol. 5006, SPIE, pp. 19–30.
- [3] CASCIA, M. L., ISIDORO, J., AND SCLAROFF, S. Head tracking via robust registration in texture map images. In *CVPR '98: Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (Washington, DC, USA, 1998), IEEE Computer Society, p. 508.
- [4] CHAI, J.-X., CHAN, S.-C., SHUM, H.-Y., AND TONG, X. Plenoptic sampling. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2000), ACM Press/Addison-Wesley Publishing Co., pp. 307–318.
- [5] CRUZ-NEIRA, C., SANDIN, D. J., AND DEFANTI, T. A. Surround-screen projection-based virtual reality: the design and implementation of the cave. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1993), ACM Press, pp. 135–142.
- [6] DODGSON, N. A. Autostereoscopic 3d displays. *Computer* 38, 8 (2005), 31–36.
- [7] GORTLER, S. J., GRZESZCZUK, R., SZELISKI, R., AND COHEN, M. F. The lumigraph. *Computer Graphics* 30, Annual Conference Series (1996), 43–54.
- [8] HAMAGUCHI, T., FUJII, T., AND HONDA, T. Real-time view-interpolation system for super multi-view 3d display(regular section). *IEICE transactions on information and systems* 86, 1 (20030101), 109–116.
- [9] HECKBERT, P. S. Fundamentals of texture mapping and image warping. Master's thesis, University of California, Berkeley, 1989.

- [10] IDESES, I., AND YAROSLAVSKY, L. P. New methods to produce high quality color anaglyphs for 3-d visualization. In *ICIAR (2)* (2004), A. C. Campilho and M. S. Kamel, Eds., vol. 3212 of *Lecture Notes in Computer Science*, Springer, pp. 273–280.
- [11] ISAKSEN, A., MCMILLAN, L., AND GORTLER, S. J. Dynamically reparameterized light fields. In *Siggraph 2000, Computer Graphics Proceedings* (2000), K. Akeley, Ed., ACM Press / ACM SIGGRAPH / Addison Wesley Longman, pp. 297–306.
- [12] LEVOY, M., AND HANRAHAN, P. Light field rendering. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), ACM Press, pp. 31–42.
- [13] PERLIN, K., PAXIA, S., AND KOLLIN, J. S. An autostereoscopic display. In *SIGGRAPH* (2000), pp. 319–326.
- [14] PERRY, J. S., AND GEISLER, W. S. Gaze-contingent real-time simulation of arbitrary visual fields. In *Proc. SPIE Vol. 4662, p. 57-69, Human Vision and Electronic Imaging VII, Bernice E. Rogowitz; Thrasyvoulos N. Pappas; Eds.* (June 2002), B. E. Rogowitz and T. N. Pappas, Eds., vol. 4662 of *Presented at the Society of Photo-Optical Instrumentation Engineers (SPIE) Conference*, pp. 57–69.
- [15] RANTAZAU, D., FRANK, K., LANG, U., RAINER, D., AND WOESSNER, U. Covise in the cube: An environment for analyzing large and complex simulation data. In *Proceedings of 2nd Workshop on Immersive Projection Technology (IPTW '98)* (Ames, Iowa, USA, 1998).
- [16] STEWART, J., YU, J., GORTLER, S., AND MCMILLAN, L. A new reconstruction filter for undersampled light fields. In *Proceedings of Eurographics Symposium on Rendering 2003 (2003)* (2003), pp. 150–156.
- [17] STIEFELHAGEN, R., YANG, J., AND WAIBEL, A. Tracking eyes and monitoring eye gaze. In *Proceedings of the Workshop on Perceptual User Interfaces (PUI'97)* (Alberta, Canada, 1997), pp. 98–100.
- [18] TALMI, K., AND LIU, J. Eye and gaze tracking for visually controlled interactive stereoscopic displays. *Signal Processing: Image Communication* 14, 10 (August 1999), 799–810.
- [19] Ascension.
<http://www.ascension-tech.com/products/flockofbirds.php>.
- [20] Nvidia geforce 7900 specifications.
http://www.nvidia.com/page/geforce_7900.html.

- [21] Openscenegraph.
<http://www.openscenegraph.org/projects/osg>.
- [22] Trolltech qt.
<http://trolltech.com/products/qt>.
- [23] Wanda.
<http://www.ascension-tech.com/products/wanda.php>.
- [24] ZWICKER, M., RÄSÄNEN, J., BOTSCH, M., DACHSBACHER, C., AND PAULY, M. Perspective accurate splatting. In *GI '04: Proceedings of Graphics Interface 2004* (School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004), Canadian Human-Computer Communications Society, pp. 247–254.