Real-Time Consensus-Based Scene Reconstruction using Commodity Graphics Hardware

Ruigang Yang, Greg Welch, Gary Bishop
Department of Computer Science
University of North Carolina at Chapel Hill

Abstract

We present a novel use of commodity graphics hardware that effectively combines a plane-sweeping algorithm with view synthesis for real-time, on-line 3D scene acquisition and view synthesis. Using real-time imagery from a few calibrated cameras, our method can generate new images from nearby viewpoints, estimate a dense depth map from the current viewpoint, or create a textured triangular mesh. We can do each of these without any prior geometric information or requiring any user interaction, in real time and on line. The heart of our method is to use programmable Pixel Shader technology to square intensity differences between reference image pixels, and then to choose final colors (or depths) that correspond to the minimum difference, i.e. the most consistent color.

In this paper we describe the method, place it in the context of related work in computer graphics and computer vision, and present some results.

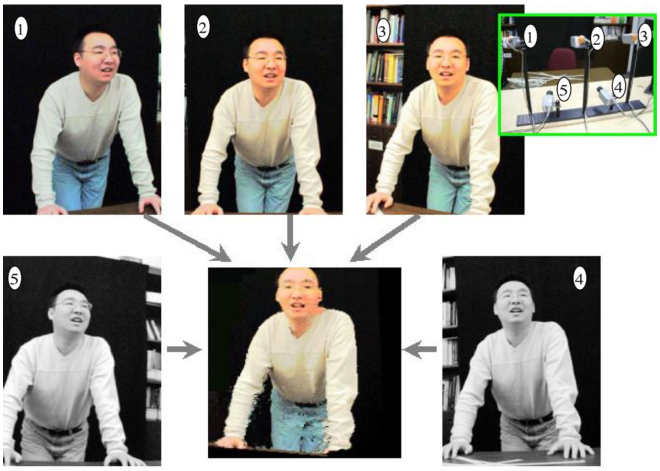

Figure 1. Example setup for 15 frame-per-second online reconstruction using five cameras (also see Figure 8 in color plate).

1 Introduction

This work is motivated by our long standing interest in tele-collabration, in particular 3D tele-immersion. We want to display high-fidelity 3D views of a remote environment in real time to create the illusion of actually looking through a large window into a collaborator's room.

One can think of the possible real-time approaches as covering a spectrum, with geometric or polygonal approaches at one end, and Light-Field Rendering [19, 8] at the other. For real scenes, geometric approaches offer tremendous challenges in terms of creating, representing, and transmitting a model. Even for static regions of a scene, it would be impractical for an individual to model manually every piece of paper, pen and pencil, and pile of junk on a typical desk. As such, researchers pursuing geometric methods have explored automated approaches, for example using computer vision techniques. Typically one constructs a 3D geometric model of the scene, and then renders the model from any desired view. However robust recovery of 3D geometry from 2D images remains an open computer vision problem. Many algorithms exist, yet they are relatively specialized or fragile, and too computationally expensive for real-time applications.

Rather than striving to understand or reproduce a geometric model of a scene, Light-Field Rendering uses a collection of 2D image samples to reconstruct a function that completely characterizes the flow of light through unobstructed space [19]. With the function in hand, view synthesis becomes a simple table lookup. Photo-realistic results can be rendered at interactive rates on inexpensive personal computers. However collecting, transmitting, and processing such dense samples from a real environment, in real time, is impractical.

We present a novel use of commodity graphics hardware that effectively combines a plane-sweeping algorithm with view synthesis for real-time, online 3D scene acquisition and view synthesis. Using real-time imagery from a few calibrated cameras, our method can generate new images from nearby viewpoints, estimate a dense depth map from the current viewpoint, or create a textured triangular mesh. We can do each of these without any prior geometric information or requiring any user interaction, and thanks to the parallelization and tremendous performance of commodity graphics hardware, we can do them interactively, in real time and on line.

In this paper we describe the method, place it in the context of related work in computer graphics and computer vision, and present some results. When describing the method we keep open the choice of either estimating color or depth, and perhaps corresponding geometry.

2 Related Work

Here we discuss some related work in image-based rendering and real-time reconstruction/rendering systems. We also discuss some related work in Section 3 where appropriate.

2.1 Vision-based Methods

Virtualized Reality^{TM} System [23, 30]. Kanade's Virtualized Reality system uses a sea of cameras in conjunction with vision techniques to extract models of dynamic scenes. These methods require significant off-line processing, so strictly speaking, it is not a real-time system yet. Recently, they are exploring to use special-purpose hardware to speed up the computation.

Real-time Depth from Stereo. Depth from stereo techniques [6] seem to be the most available option for computing depth from images because of their unobtrusive nature and the ease of data acquisition. However, these techniques are computationally intensive and typically require special hardware to operate in real time [13, 31, 15]. Recently, Mulligan and Daniilidis proposed a new trinocular stereo algorithm in software [22]. They used a number of techniques to accelerate the computation, such as motion prediction and assembly level instruction optimization. However, the stereo matching part is still the bottleneck in their method (1-2 frames/second).

2.2 Image-based Methods

Image-based Visual Hull. Matusik et. al. presented an efficient method for real-time rendering of a dynamic scene [21]. They used an image-based method to compute and shade visual hulls [18] from silhouette images. A visual hull is constructed by using the visible silhouette information from a series of reference images to determine a conservative shell that progressively encloses the actual object. Unlike previously published methods, they constructed the visual hulls in the reference image space and used an efficient pixel traversing scheme to reduce the computational complexity to $O(n^2)$, where n^2 is the number of pixels in a reference image. Their system uses a few cameras (four in their demonstration) to cover a very wide field of view and is very effective in capturing the dynamic motion of objects. However, their method, like any other silhouette-based approach, cannot handle concave objects, which makes close-up views of concave objects less satisfactory.

Hardware-assisted Visual Hull. Based on the same visual hull principle, Lok proposed a novel technique that leverages the tremendous capability of modern graphics hardware [20]. The 3D volume is discretized into a number of parallel planes, the segmented reference images are projected onto these planes using projective textures. Then, he makes clever use of the stencil buffer to rapidly determine which volume samples lie within the visual hull. His system benefits from the rapid advances in graphics hardware and the main CPU is librated for other high-level tasks. However, this approach suffers from a major limitation – the computational complexity of his algorithm is $O(n^3)$. Thus it is difficult to judge if his approach will prove to be faster than a software-based method with $O(n^2)$ complexity.

Generalized Lumigraph with Real-time Depth. Schirmacher et. al. introduced a system for reconstructing arbitrary views from multiple source images on the fly[26]. The basis of their work is the two-plane parameterized Lumigraph with per-pixel depth information. The depth information is computed on the fly using a depth-from-stereo algorithm in software. With a dense depth map, they can model both concave and convex objects. Their current system is primarily limited by the quality and the speed of the stereo algorithm (1-2 frames/second).

3 Our Method

We believe that our method combines the advantages of previously published real-time methods in Section 2, while avoiding some of their limitations as follows.

- We achieve real-time performance without using any special-purpose hardware.
- We can deal with arbitrary objects, both concave and convex.
- We do not use silhouette information, so there is no need for image segmentation, which is not always possible in a real environment.
- We use graphics hardware to accelerate the computation without increasing the symbolic complexity; our method is $O(n^3)$ while the lower bound of depth-from-stereo algorithms is also $O(n^3)$.
- Our proposed method is more versatile. A number of systems listed here, such as the Virtualized-Reality system and the generalized lumigraph system, could benefit from our method since we can rapidly compute a geometric model using the graphics hardware.

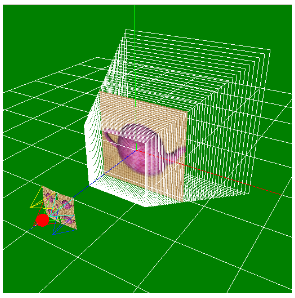

Figure 2. A configuration where there are five input cameras, the red dot represents the new view point. Spaces are discretized into a number of parallel planes.

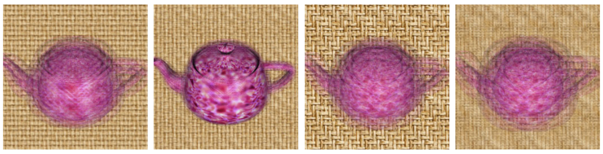

Figure 3. Depth plane images from step 0, 14, 43, 49, from left to right; The scene, which contains a teapot and a background plane, is discretized into 50 planes.

3.1 Overview

We want to synthesize new views given calibrated input images. We can discretize the 3D space into parallel planes. For each plane D_i, we project the calibrated input images onto it, as shown in Figure 2. If there is indeed a surface on D_i, the projected images on that spot should be the same color if two assumptions are satisfied: (A) the surface is visible, i.e. there is no occluder between the surface and the images; and (B) the surface is diffuse, so the reflected light does not depend on the 3D position of the input images.

If we choose to accept that two above assumptions are satisfied, the *color consistency* on plane D_i is a measure if there is a surface on D_i. If we know the surface position, then we can trivially render it from any new view point.

So here is our proposed method to render new views from calibrated input images. For a desired new view C_n (the red dot in Figure 2), we discretize the 3D space into planes parallel to the image plane of C_n. Then we step through the planes. For each plane D_i, we project the input images on these planes, and render the textured plane on the image plane of C_n to get a image (I_i) of D_i. While it is easy to conceptually think of these as two separate operations, we can combine them into a single homography (planar-to-planar) transformation. In Figure 3, we show a number of images from different depth planes. Note that each of these images contains the projections from all input images, and the area corresponding to the intersection of objects and the depth plane remains sharp. For each pixel location (u,v) in I_i, we compute the mean and variance of the projected colors. The final color of (u,v) is the color with minimum variance in $\{I_i\}$, or the color most consistent among all camera views.

The concept of sweeping a plane through a discretized space is not new, it has been used in a number of computer vision algorithms [4, 27, 16, 29] for scene reconstruction. However, we have developed a means to combine scene reconstruction and view synthesis into a single step. We do so using the programmable Pixel Shader technology typically available on modern graphics hardware. The use of graphics hardware for real-time online acquisition, reconstruction, and rendering is central to this paper.

3.2 Relationship with other Techniques

Stereo Techniques. From a computer vision perspective, our method is *implicitly* computing a depth map from the new viewpoint C_n (readers who are unfamiliar with computer vision terms are referred to Faugeras' textbook [7]). If we only use two reference cameras and make C_n the same as one of them, say the first one, we can consider our method as combining depth-from-stereo and 3D image warping in a single step. The projection of the view ray $P(u,v)$ into the second camera is the epipolar line (its projection in the first camera reduces to a single point). If we clip the ray $P(u,v)$ by the near and the far plane, then its projection defines the disparity search range in the second camera's image. Thus stepping along $P(u,v)$ is equivalent to stepping along the epipolar line on the second camera, and computing the color variance is similar to computing the Sum-of-Squared-Difference (SSD) over a 1×1 window in the second camera. Typically, the SSD score over a single pixel support window does not provide sufficient disambiguating power, so it is essential for our method to use several cameras. As such our method can be considered in effect a multi-baseline stereo algorithm [25] operating with a 1×1 support window, with the goal being the best color estimate rather than depth.

Conceptually, our method distributes the correlation support among input cameras. In binocular stereo, the support is distributed around the neighborhood of the pixel of interest within a image. And in multi-baseline stereo, the support is distributed both among cameras and within images. There are certain tradeoffs among these choices. While a big support area in the input images is more economical, it does make a strong assumption about surface smoothness and favors frontal-parallel surfaces. On the other hand, a small support size requires more cameras to provide reliable estimates.

Light Field Rendering. Isaksen et. al. introduced the idea of dynamically re-parameterizing the light field function that allows the user to *select* a single depth value for all the view rays[12]. We extend this dynamic parameterization to the extreme–we allow each ray to have an implicit depth value that our method automatically *estimates* on the fly.

Space Carving. Our consensus function using inter-camera variance is similar to the photo-consistency check first proposed by Seitz et. al. [27, 16]. However, they used it to create a volumetric model of a static scene, while we apply it directly for view synthesis. We believe our approach brings a number of advantages. First, the photo-consistency check is ambiguous in textureless (uniformly colored) regions, which usually leads to errors in the volumetric model. But such situations are generally fine for view synthesis, since using any color in the textureless region (in fact they are all the same) will create the correct image. Secondly, since we know the new view point and the desired output image resolution at the time of synthesis, we could use the method in [3] to select the minimum number of depth steps needed. Thus, we do the minimum amount of work to generate a image. For a low resolution image, we could use fewer depth steps. This ”lazy evaluation” is best suited for real-time applications. We also reduce the quantization artifacts since the space is implicitly quantized according to the output resolution.

3.3 Hardware Acceleration

While it is straightforward to use texture mapping functionalities in graphics hardware to project the input images on to the depth planes, our hardware acceleration scheme does not stop there. Modern graphic cards, such as the NVIDIAs GeForce series [24], provide a programable means for per-pixel fragment coloring through the use of *Register Combiner* [14]. We exploit this programmability, together with the texture mapping functions, to carry out the entire computation on the graphics board.

3.3.1 Real-time View Synthesis

In our hardware-accelerated renderer, we step through the depth planes from near to far. At each step (i), there are two stages of operations, *scoring* and *selection*. In the *scoring* stage, we set up the transformation according to the new view point. We then project the reference images onto the depth plane D_i. The textured D_i is rendered into the image plane (the frame buffer).

We would like to program the *Pixel Shader* to compute the RGB mean and a luminance “variance” per pixel. Computing the true variance requires two variables per pixel. Since the current hardware is limited to four channels and not five, we opt to compute the RGB mean and a single-variable approximation to the variance. The latter is approximated by the sum-of-squared-difference (SSD), that is

$$SSD = \sum_i (Y_i - Y_{base})^2 \tag{1}$$

where Y_i is the luminance from an input image and Y_{base} is the luminance from a base reference image selected as the input image that is closest to the new viewpoint. This allows us to compute the SSD score sequentially, an image pair at a time. In this stage, the frame buffer acts as an accumulation buffer to keep the mean color (in the RGB channel) and the SSD score (in the alpha channel) for D_i. In Figure 4, we show the SSD score images (the alpha channel of the frame buffer) at different depth steps. The corresponding color channels are shown in Figure 3.

In the next *selection* stage, we need to select the mean color with the smallest SSD score. The content of the frame buffer is copied to a temporary texture (Tex_{work}), while another texture (Tex_{frame}) holds the mean color and minimum SSD score from the previous depth step. These two textures are rendered again into the frame buffer through orthogonal projection. We reconfigure the *Pixel Shader* to compare the alpha values on a per pixel basis, the output color is selected from the one with the minimum alpha (SSD) value. Finally the updated frame buffer’s content is copied to Tex_{frame} for use in the next depth step. The complete pseudo code for our hardware-accelerated render is provided here. Details about the settings of the *Pixel Shader* are in Appendix A. Note that once the input images are transferred to the texture memory, all the computations are performed on the graphics board. There is no expensive copy between the host memory and the graphics broad, and the host CPU is essentially idle except for executing a few OpenGL commands.

```
createTex(workingTexture);
createTex(frameBufferTexture);
for (i = 0; i< steps; i++) {
   // the scoring stage;
   setupPerspectiveProjection();
   glEnable(GL_BLEND);
   glBlendFunc(GL_ONE, GL_ONE);
   setupPixelShaderForSDD();
   for (j = 0; j< inputImageNumber; j++)
     projectImage(j, baseReferenceImage);

  //  the selection stage;
  if (i == 0) {
   copyFrameToTexture(frameBufferTexture);
   continue;
  } else
   copyFrameToTexture(workingTexture);

  setupPixelShaderForMinMax();
  setupOrthogonalProjection();
```

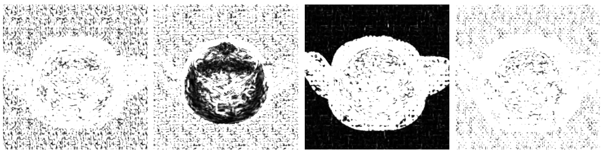

Figure 4. SSD scores (encoded in the alpha channel) for different depth planes in the first *scoring* **stage. We use the same setup as in Figure 3. From left to right, the corresponding depth steps are 0, 14, 43, 49.**

```
  renderTex(workingTexture,
            frameBufferTexture);
  copyFrameToTexture(frameBufferTexture);
}
```

3.3.2 Real Time Depth

As explained in Section 3.2, our technique is *implicitly* computing a depth map from a desired view point, we just choose to keep the best color estimate for the purpose of view synthesis. If we choose to trade color estimation for depth estimation, we can use almost the same method above (Section 3.3.1) to compute a depth map in real-time, and liberate the CPU for other high level tasks. The only change necessary is to configure the graphics hardware to keep the depth information instead of the color information. The color information can then be obtained by re-projecting the reconstructed 3D points into input images. From an implementation standpoint, we can encode the depth information in the RGB portion of the texture image.

It should be noted that it is not robust to estimate the depth information based on the variance of a single pixel, especially for real images in which there is camera noise, errors in the calibration, or surfaces which are not perfectly diffuse. That is why most stereo algorithms use an area-based approach to aggregate support spatially by summing over a window in the disparity (image) space. The latest official OpenGL Specification (Version 1.3) [1] has an Imaging Subset that supports convolution functions. By convolving the frame buffer with a blurring filter, we can sum up the SSD scores from neighboring pixels to make the depth estimate more robust. This can be done after the *scoring* stage, during the first copy from the frame buffer to the texture memory(`copyFrameToTexture()`).

```
glEnable(GL_CONVOLUTION_2D);
copyFrameToTexture(workingTexture)
glDisable(GL_CONVOLUTION_2D);
```

If the convolution function is implemented in hardware, there will be little performance penalty since the convolution is already a part of the pixel transfer pipeline. Unfortunately, hardware-accelerated convolutions are only available on expensive graphics workstations such as SGI's Oynx2[1], but these expensive workstations do not have a programable pixel shader . Given the tremendous advances in commodity graphics hardware in recent years, we are optimistic that new hardware that supports both the complete function set of OpenGL 1.3 and programmable shaders will soon emerge. When this becomes true, our algorithm provides a practical and inexpensive way for real-time stereo vision, which to date has only been achieved by using either special hardware [13, 31, 15], or highly optimized software at a very limited resolution.

Real-time Depth Postprocessing Without hardware support for convolution functions, the depth map computed from real data with 1-pixel kernels can be quite noisy. However, we can filter out the majority of outliers in software (the CPU is otherwise idle). This postprocessing involves three steps: (I) the depth data with their SSD scores are read back into the main memory; (II) SSD Test: Any point with a SSD score larger than a threshold (T_{ssd}) is rejected; (III) Correlation Test: If a point passes the SSD test, it will be projected into input images. For each pair between the base reference images and the others, we compute a normalized correlation score over a small window. If the average of the correlation scores is smaller than a threshold (T_{nc}), that depth point will be rejected.

When dealing with real data, we observed that not many points will survive after the postprocessing. However, the survived points contain very few outliers. In fact, we can use the filtered depth map to create a textured mesh model using Delaunay triangulation.

Although this optional postprocessing falls back to the difficult problem of reconstructing a 3D model, it does have a few advantages:

- It decouples the modeling and the rendering loop. The textured geometric model can be rendered at a faster rate, using the same graphics hardware.
- The model can be used with traditional compute graphics objects, or as input to other image-based rendering methods, such as [5, 2, 8], to provide more compelling results.
- We have observed that view extrapolation deteriorates rapidly, which seems to concur with the report by Szeliski [28]. The textured model extends the effective view volume, even allowing oblique views from the reference images.
- It is more efficient for stereo viewing. Furthermore, a geometric representation that is consistent between left

[1] There are ways for performing convolution using PC hardware. One could use multiple textures, one for each of the neighboring pixels; or render the scene in multiple passes and perturb the texture coordinate in each pass. However these tricks significantly decrease the speed.

and right-eye views makes stereo fusion easier for the human visual system.

3.4 Tradeoffs Using the Graphics Hardware

There are certain tradeoffs we have to make when using the graphics hardware. The lack of hardware accelerated convolution functions is one of them. Another common complaint about current graphics hardware is the limited arithmetic resolution. Our method, however, is less affected by this limitation. Computing the SSD scores is the central task of our method. SSD scores are always non-negative, so they are not affected by the unsigned nature of the frame buffer. (The computation of SSD is actually performed in signed floating point on the latest graphics card, such as the GeForce4 from NVIDIA.) A large SSD score means there is a small likelihood that the color/depth estimate is correct. So it does not matter if a very large SSD score is clamped, it is not going to affect the estimate anyway.

A major limitation of our hardware acceleration scheme is the inability to handle occlusions. In software, we could use the method introduced in the Space Carving algorithm [16] to mark off pixels in the input images, however, there is no such “feedback” channel in the graphics hardware. To address this problem in practice, we use a small baseline between cameras, a design adopted by many multi-baseline stereo algorithms. However, this limits the effective view volume, especially for direct view synthesis. Our preliminary experiments indicate that a large support size (more filtering) could successfully address this problem in practice.

The bottleneck in our hardware acceleration scheme is the fill rate. This limitation is also reported by Lok in his hardware-accelerated visual hull computation [20]. More detailed analysis can be found there.

4 System Implementation and Results

We have implemented a distributed system using four PCs and five calibrated 1394 digital cameras (SONY DFW-V500). Camera exposure is synchronized by use of an external trigger. Three PCs are used to capture the video streams and correct for lens distortions. The corrected images are then compressed and sent over a 100Mb/s network to the rendering PC with a graphics accelerator. We have tested our algorithm on two NVIDIA cards, a Quadro2 Pro and GeForce3. Performance comparisons are presented in Table 1. On average, the GeForce3 is about 75 percent faster than Quadro2 Pro for our application.

We discussed in Section 3.3.2 how the support size plays an important role in the quality of reconstruction. Our experiments show that even a small support size (3×3 or 5×5) can improve the results substantially (Figure 5). Note that

	128^2	256^2	512^2
20	16, 40	31, 55	82, 156
50	31, 85	70, 130	211, 365
100	62, 140	133, 235	406, 720

Table 1. Rendering time per frame in milliseconds (number of depth planes vs output resolutions). The first number in each cell is from GeForce3; the second from Quadro2 Pro. All numbers are measured with five 320×240 input images.

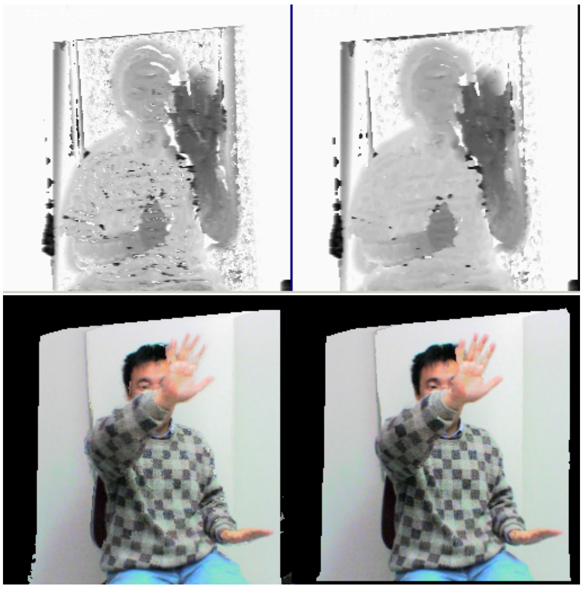

Figure 5. Impact of support size on the depth and color reconstruction; Left: 1×1 support (real-time results); Right: 5×5 support (also in color plate).

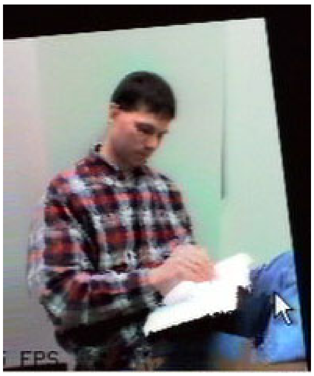

Figure 6. An live view directly captured from the screen.

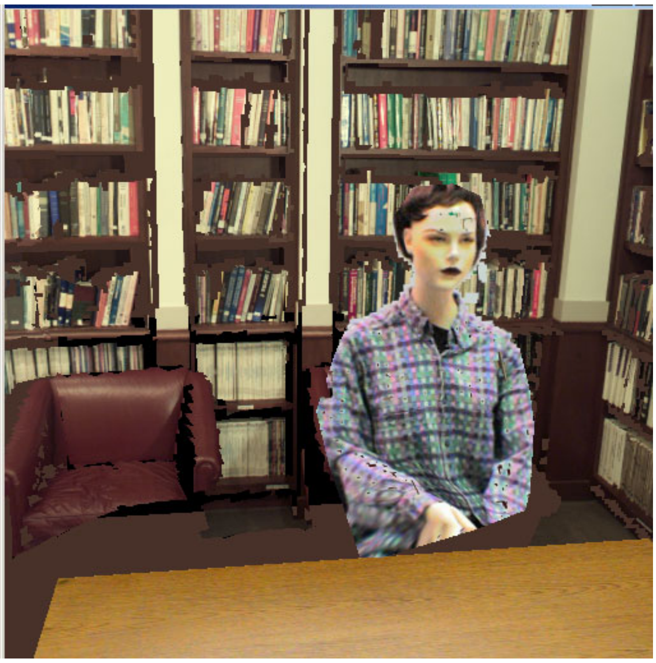

Figure 7. A dynamic mesh model of the mannequin is evaluated and rendered with a static 3D background.

these test results are not based on software simulation; they are obtained using OpenGL functions on real hardware. These results further demonstrate the viability and potential of our method.

Figure 6 shows a live image computed online in real time. More lives images can be found in the color plate. Figure 7 shows a textured, mesh model (foreground mannequin) created in real time from postprocessing the raw depth map. The mesh model is rendered with a static 3D background model.

5 Conclusions

As stated in the introduction, this work is motivated by our long standing interest in 3D tele-immersion. An inverse or image-based approach to view-dependent rendering or scene reconstruction has appeal to us for this particular application for several reasons. For one, the finite and likely limited inter-site bandwidth motivates us to consider methods that "pull" only the necessary scene samples from a remote to a local site, rather than attempting to compute and "push" geometry from a remote to a local site. In addition we would like to be able to leverage existing video compression schemes, so would like our view/scene reconstruction to be as robust to sample artifacts (resulting from compression for example) as possible. We are working with systems and networking collaborators to explore new encoding schemes aimed specifically at this sort of reconstruction.

We look forward to wider implementation of the OpenGL extensions that facilitate hardware accelerated convolution filters on commodity hardware. In any case we intend to explore higher-order or larger-support convolution using hardware as in [9]. We are considering extending our method to allow for likelihood-based global constraints. For example we would like to allow for the likelihood of a surface along one ray to affect the likelihoods of individual samples along other rays, to support transparencies, etc. We are also thinking about formulating the reconstruction more in 3D, perhaps leveraging work in 3D convolution using graphics hardware [10].

Like other recent work that makes use of increasingly powerful graphics hardware for unusual purposes [17, 11] we hope that our method inspires further thinking and additional new methods.

6 Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant No. 0121657, "Electronic Books for the Tele-Immersion Age". We made use of equipment provided by DOE under contract B519834, and personal computers donated by the Intel corporation.

We also acknowledge the support of our close collaborators at UNC-CH (www.cs.unc.edu/˜stc/office/index.html), in particular we thank Herman Towles for his continuous leadership and gracious support. We thank John Thomas and Jim Mahaney for their engineering assistance, and Scott Larsen and Deepak Bandyopadhyay for their help in shooting video.

References

[1] OpenGL Specification 1.3, August 2001. `http://www.opengl.org/developers/documentation/version13/glspec13.pdf`; accessed January 8, 2002.

[2] Chris Buehler, Michael Bosse, Leonard McMillan, Steven Gortler, and Michael Cohen. Unstructured Lumigraph Rendering. In *Proceedings of SIGGRAPH 2001*, Los Angeles, August 2001.

[3] Jin-Xiang Chai, Xin Tong, Shing-Chow Chan, and Heung-Yeung Shum. Plenoptic Sampling. In *Proceedings of SIGGRAPH 2000*, page 307318, New Orleans, August 2000.

[4] R. T. Collins. A Space-Sweep Approach to True Multi-image Matching. In *Proceedings of Conference*

on Computer Vision and Pattern Recognition, pages 358–363, June 1996.

[5] P. Debevec, C. Taylor, and J. Malik. Modeling and Rendering Architecture from Photographs. In *Proceedings of SIGGRAPH 1996*, Annual Conference Series, pages 11–20, New Orleans, Louisiana, August 1996. ACM SIGGRAPH, Addison Wesley.

[6] U. Dhond and J. Aggrawal. Structure from stereo: a review. *IEEE Transactions on Systems, Man, and Cybernetics*, 19(6):14891510, 1989.

[7] O. Faugeras. *Three-Dimensional Computer Vision: A Geometric Viewpoint*. MIT Press, 1993.

[8] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen. The Lumigraph. In *Proceedings of SIGGRAPH 1996*, pages 43–54, New Orleans, August 1996.

[9] Markus Hadwiger, Thomas Theul, Helwig Hauser, and Eduard Grller. Hardware-Accelerated Hiqh-Quality Filtering on PC Graphics Hardware. In *Proceedings of Vision, Modeling, and Visualization 2001*, Stuttgart, Germany, November 2001.

[10] Matthias Hopf and Thomas Ertl. Accelerating 3D Convolution using Graphics Hardware. In *Proceedings of IEEE Visualization 99*, San Francisco, USA, October 1999.

[11] Kenneth E. Hoff III, Andrew Zaferakis, Ming Lin, and Dinesh Manocha. Fast and Simple 2D Geometric Proximity Queries Using Graphics Hardware. In *Proceedings of ACM Symposium on Interactive 3D Graphics*, March 2001.

[12] Aaron Isaksen, Leonard McMillan, and Steven J. Gortler. Dynamically Reparameterized Light Fields. In *Proceedings of SIGGRAPH 2000*, pages 297–306, August 2000.

[13] T. Kanade, A. Yoshida, K. Oda, H. Kano, and M. Tanaka. A Stereo Engine for Video-rate Dense Depth Mapping and Its New Applications. In *Proceedings of Conference on Computer Vision and Pattern Recognition*, pages 196–202, June 1996.

[14] Mark J. Kilgard. A Practical and Robust Bump-mapping Technique for Today's GPUs. In *Game Developers Conference 2000*, San Jose, California, March 2000.

[15] K. Konolige. Small Vision Systems: Hardware and Implementation. In *Proceedings of the 8th International Symposium in Robotic Research*, page 203212. Springer-Verlag, 1998.

[16] K. Kutulakos and S. Seitz. A Theory of Shape by Space Carving. Technical Report TR692, Computer Science Dept., U. Rochester, 1998.

[17] Scott Larsen and David McAllister. Fast Matrix Multiplies using Graphics Hardware. In *Proceedings of ACM Supercomputing 2001*, Denver, CO, USA, November 2001.

[18] A. Laurentini. The Visual Hull Concept for Silhouette Based Image Understanding. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(2):150–162, February 1994.

[19] M. Levoy and P. Hanrahan. Light Field Rendering. In *Proceedings of SIGGRAPH 1996*, pages 31–42, New Orleans, August 1996.

[20] B. Lok. Online Model Reconstruction for Interactive Virtual Environments. In *Proceedings 2001 Symposium on Interactive 3D Graphics*, pages 69–72, Chapel Hill, North Carolina, March 2001.

[21] W. Matusik, C. Buehler, R. Raskar, S. Gortler, and L. McMillan. Image-Based Visual Hulls. In *Proceedings of SIGGRAPH 2000*, pages 369–374, New Orleans, August 2000.

[22] J. Mulligan, V. Isler, and K. Daniilidis. Trinocular stereo: A new algorithm and its evaluation. *to appear in International Journal of Computer Vision, Special Issue on Multi-baseline Stereo*, 2002.

[23] P. Narayanan, P. Rander, and T. Kanade. Constructing Virtual Worlds using Dense Stereo. In *Proceedings of International Conference on Computer Vision*, pages 3–10, June 1998.

[24] Nvidia. http://www.nvidia.com.

[25] M. Okutomi and T. Kanade. A Multi-baseline Stereo. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(4):353–363, April 1993.

[26] Hartmut Schirmacher, Li Ming, and Hans-Peter Seidel. On-the-Fly Processing of Generalized Lumigraphs. *EUROGRAPHICS 2001*, 20(3), 2001.

[27] S. M. Seitz and C. R. Dyer. Photorealistic Scene Reconstruction by Voxel Coloring. In *Proceedings of CVPR 1998*, pages 1067–1073, 1997.

[28] R. Szeliski. Prediction Error as a Quality Metric for Motion and Stereo. In *Proceedings of International Conference on Computer Vision*, pages 781–788, Sept 1999.

[29] R. Szeliski and P. Golland. Stereo Matching with Transparency and Matting. In *Proceedings of International Conference on Computer Vision*, pages 517–524, Sept 1998.

[30] Sundar Vedulay, Simon Baker, Peter Randeryz, Robert Collinsy, and Takeo Kanadey. Three Dimensional Scene Flow. In *Proceedings of International Conference on Computer Vision*, pages 722–729, Sept 1999.

[31] John Woodfill and Brian Von Herzen. Real-time stereo vision on the PARTS reconfigurable computer. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 201–210, Los Alamitos, CA, 1997. IEEE Computer Society Press.

A Pixel Shader Pseudo Code

The following code is written roughly following the syntax of `nvparser`, a generalized compiler for NVIDIA extensions. Documentation about `nvparser` can be found on NVIDIA's web site at http://www.nvidia.com.

A.1 Code to compute the squared difference

This piece of code assumes that there are m input images. The alpha channel of the input images contains a gray scale copy of the image, and the base reference image is stored in tex0. The squared difference is computed on the gray scale images. The scales in the code are necessary because the unsigned char values are converted to floating point values between [0, 1] within Pixel Shader. If no scale is applied, the output squared value (in unsigned char) will be floor($(a-b)^2/256$), where a and b are the input values (in unsigned char). In our implementation, we use a combined scale factor of 32, effectively computing floor($(a-b)^2/32$).

```
const1 = {1/m, 1/m, 1/m, 1};
// the base reference image
// will be added
// m-1 times more than
// the other images;
const0 = {1/((m)(m-1)), 1/((m)(m-1)),
          1/((m)(m-1)), 1};

// **** combiner stage 0;
{
    rgb {
      spare0 = tex1*const1
           + tex0*const0;
    }
    alpha {
      spare0 = tex1 - tex0;
      scale_by_four ();
    }
}
// **** combiner stage 1
{
    alpha{
        spare0 = spare0*spare0;
        scale_by_four ();
    }
}
// **** final output
{
    out.rgb = spare0.rgb;
    out.alpha = spare0;
}
```

A.2 Code to do the minimum alpha test

This piece of code assumes that the mean colors are stored in the RGB channel while the SSD scores are stored in the alpha channel.

```
// **** combiner stage 0;
{
    alpha {
        // spare0 = tex1 - tex0 + 0.5
      spare0 = tex1 - half_bias(tex0);
    }
}
// **** combiner stage 1
{
    rgb{
    // select the color with
    // the smaller alpha;
    // spare0 =
    //  (spare0.alpha < 0.5) ?
    //        (tex1) : (tex0);
        spare0 = mux();
    }
    alpha{
    // select the smaller
    // alpha value
        spare0 = mux();
    }
}
// **** final output
{ out = spare0; }
```

Real-Time Consensus-Based Scene Reconstruction using Commodity Graphics Hardware

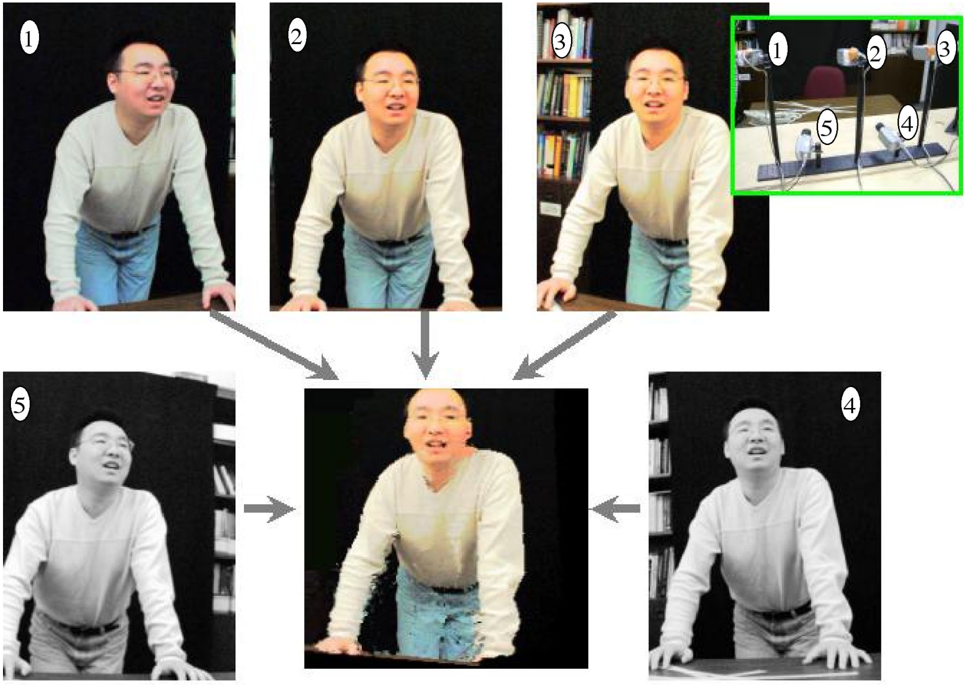

Figure 8. Example setup for 15 frame-per-second online reconstruction using five cameras.

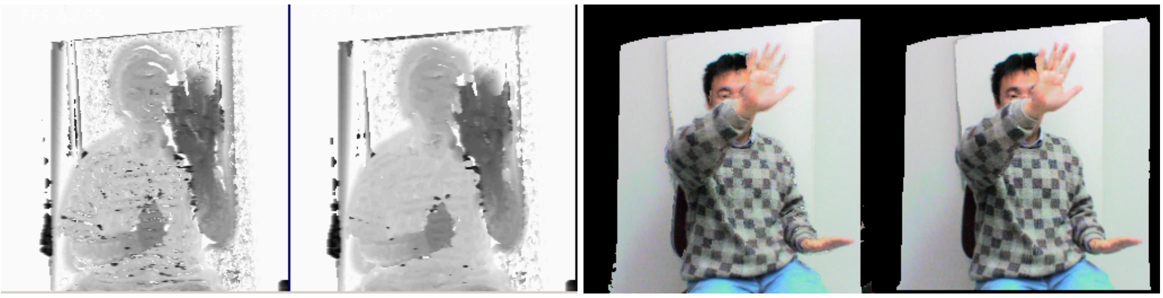

Figure 9. Impact of support size on the depth and color reconstruction; For each pair, the one on the left uses a 1×1 support window (real-time results) while the one on the right uses a 5×5 support window.

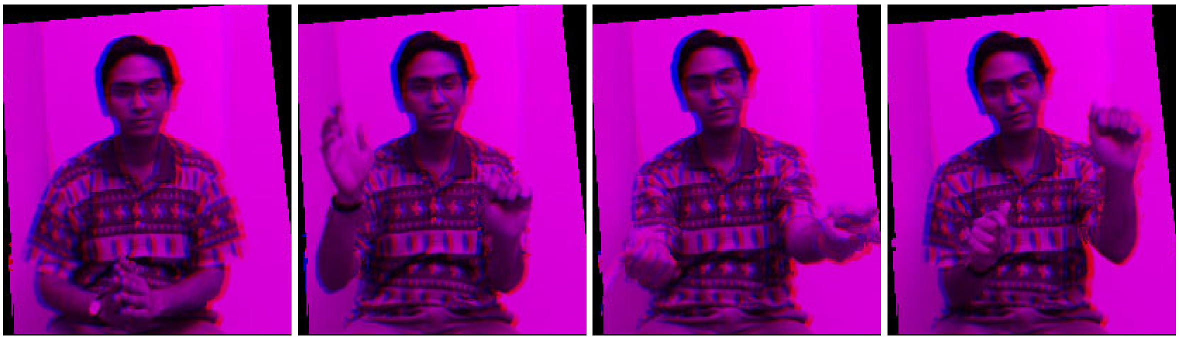

Figure 10. Red-blue stereo images from a live sequence. Use red-blue eyeglasses for stereo viewing.