To appear in the Special Issue of the International Journal of Image and Graphics (IJIG), 2003

A Unified Approach to Real-Time, Multi-Resolution, Multi-Baseline 2D View Synthesis and 3D Depth Estimation using Commodity Graphics Hardware*

Ruigang Yang[†], Marc Pollefeys, Hua Yang, Greg Welch
Department of Computer Science,
University of North Carolina at Chapel Hill,
Chapel Hill, NC 27599, U.S.A.
{ryang, marc, yanghua,welch}@cs.unc.edu

August 6, 2003

Abstract

We present a new method for using commodity graphics hardware to achieve real-time, on-line, 2D view synthesis or 3D depth estimation from two or more calibrated cameras. Our method combines a 3D plane-sweeping approach with 2D multi-resolution color consistency tests. We project camera imagery onto each plane, compute measures of color consistency throughout the plane at multiple resolutions, and then choose the color or depth (corresponding plane) that is most consistent. The key to achieving real-time performance is our use of the advanced features included with recent commodity computer graphics hardware to implement the computations *simultaneously* (in parallel) across all reference image pixels on a plane.

Our method is relatively simple to implement, and flexible in term of the number and placement of cameras. With two cameras and

*This paper is a combined and extended version of two previous papers, [36] resented at Pacific Graphics 2002 in Beijing China, and [37] presented at CVPR 2003.

†Currently at the University of Kentucky (ryang@cs.uky.edu).

an NVIDIA GeForce4 graphics card we can achieve 50-70M disparity evaluations per second, including image download and read-back overhead. This performance matches the fastest available commercial software-only implementation of correlation-based stereo algorithms, while freeing up the CPU for other uses.

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 *Light-Field* Rendering at one end, and geometric or polygonal approaches at the other. Light-Field Rendering uses a collection of 2D image samples to reconstruct a function that completely characterizes the flow of light through unobstructed space[20, 8]. 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.

Alternatively, some researchers use computer vision techniques (typically correlation-based stereo) to reconstruct a 3D scene model, and then render that model from the desired view. It is only recently that real-time implementations of stereo vision became possible on commodity PCs, with the help of rapid advances in CPU clock speed, and assembly level optimizations utilizing special extensions of the CPU instruction set. One such example is the MMX extension from Intel. Software implementations using these extensions can achieve up to 65 million disparity estimations per second (Mde/s) [11, 12, 14]. While this is impressive, there are typically few CPU cycles left to perform other tasks, such as processing related to user interaction. Further if the primary goal is 2D view synthesis, a complete geometric model (depth map) is not necessary.

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 in a single step. Using real-time imagery from two or more calibrated cameras, our method can generate new

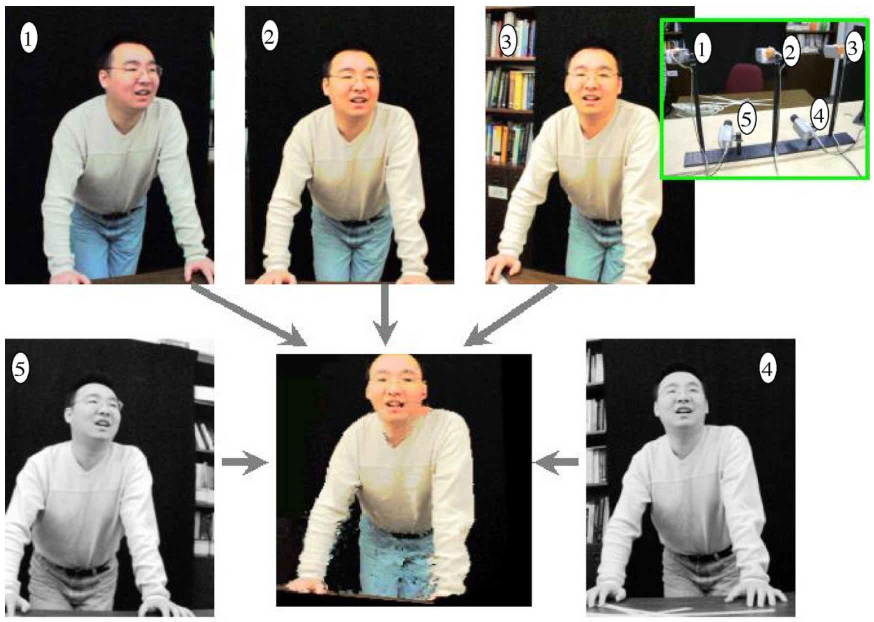

Figure 1: Example setup for real-time online reconstruction using five cameras.

images from nearby viewpoints, or if desired, estimate a dense depth map from the current viewpoint in real time and on line.

At the heart of our method is a multi-resolution approach to find the most consistent color (and corresponding depth) for each pixel. We combine sum-of-square-differences (SSD) consistency scores for windows of different sizes. This approach is equivalent to using a *weighted correlation kernel* with a *pyramidal* shape, where the tip of the pyramid corresponds to a single-pixel support and each square horizontal slice from below the tip to the base corresponds to a doubling of the support. This multi-resolution approach allows us to achieve good results close to depth discontinuities as well as on low texture areas. By utilizing the programmable Pixel Shader and mipmap functionality [33] on the graphics hardware, we can compute multi-resolution consistency scores very efficiently, enabling interactive viewing of a dynamic real scene. Parts of the results have been previously presented at Pacific Graphics 2002 in Beijing China [36] and CVPR 2003 [37].

In the next section we discuss related work in computer vision and computer graphics. After that we present an overview of our method followed by a detailed description. Finally, we present experimental results for novel view synthesis and depth estimation.

2 Related Work

Recently, with the increasing computational power on inexpensive personal computers and the wide availability of low-cost imaging device, several real-time methods have been proposed to capture and render dynamic scenes. They are of particular interest to this thesis and are reviewed here.

2.1 Stereo Vision Methods

Stereo vision is one of the oldest and most active research topics in computer vision. It is beyond the scope of this paper to provide a comprehensive survey. Interested readers are referred to a recent survey and evaluation by Scharstein and Szeliski [28]. While many stereo algorithms obtain high-quality results by performing global optimizations, today only correlation-based stereo algorithms are able to provide a dense (per pixel) depth map in real time on standard computer hardware.

Only a few years ago special hardware had to be used to achieve real-time performance with correlation-based stereo algorithms [6, 15, 35]. It was until recently that, with the tremendous advances in computer hardware, software-only real-time systems began to merge. For example, Mulligan and Daniilidis proposed a new trinocular stereo algorithm in software [24] to achieve 3-4 frames/second on a single multi-processor PC. Hirschmuler introduced a variable-window approach while maintaining real-time suitability [12, 11]. There is also a commercial package from Point Grey Research [14] which seems to be the fastest one available today. They report 45 Mde/s on a 1.4GHz PC, which extrapolates to 64Mde/s on a 2.0GHz PC.

These methods use a number of techniques to accelerate the calculation, most importantly, assembly level instruction optimization using Intel's MMX extension. While the reported performance of 35-65 Mde/s is sufficient to obtain dense-correspondences in real-time, there are few CPU cycles left to perform other tasks such as high-level interpretation of the stereo results. Furthermore, most approaches use an equal-weight box-shaped filter to aggregate the correlation scores, so the result from the previous pixel location can be used in the current one. While this simplifies the implementation and greatly reduces computational cost, the size of the aggregation window has a significant impact on the resulting depth map.

2.2 Image-based Methods

Image-based Visual Hull. Matusik et. al. presented an efficient method for real-time rendering of a dynamic scene [23]. They used an image-based method to compute and shade visual hulls [19] 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. Another disadvantage of image-based visual hull is that it completely relies on successful background segmentation.

Hardware-assisted Visual Hull. Based on the same visual hull principle, Lok presented a novel technique that leverages the tremendous capability of modern graphics hardware [22]. 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 liberated for other high-level tasks. However, this approach suffers from a major limitation – the computational complexity of his algorithm is $O(n^3)$, where n is the width of the input images (assuming a square image). 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 [29]. 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, and our method, based on the OpenGL specification, is relatively easy to implement.
- We can deal with arbitrary object shapes, including concave and convex objects.
- 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 $O(n^3)$, the same as most correlation-based stereo algorithms.
- Our proposed method is more versatile. We can use two or more cameras in a casual configuration, including configurations where the images contain epipolar points. This case is problematic for image-pair rectification [5], a required pre-processing step for most real-time stereo algorithms.

3.1 Overview

Our primary goal is to synthesize a new view, given two or more calibrated input images. We begin by discretizing the 3D space into parallel planes orthogonal to the view direction. We then project the calibrated input images onto each plane D_i as shown in Figure 2. If there is indeed a surface at a point on D_i, the projected images at that point 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 Lambertian—the reflected light does not depend on the 3D position of the input images.

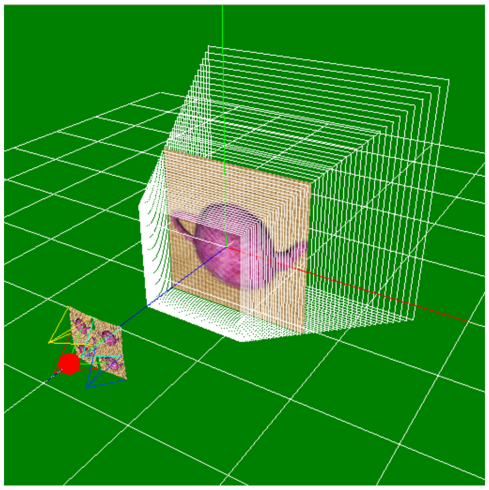

Figure 2: A configuration where there are five input cameras, the red dot represents the new view point. Spaces are discretized into a number of parallel planes.

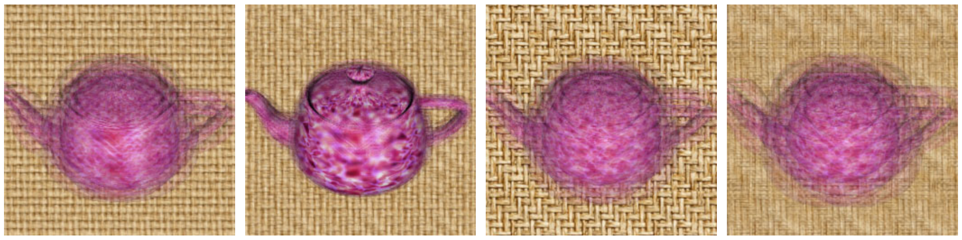

Figure 3: Depth plane images from step 0, 14, 43, 49, from left to right; The scene, which contains a teapot and a background plane, is discretized into 50 planes.

If we choose to accept that the two above assumptions are satisfied, then the *color consistency* at each point on plane D_i can be used as an indication of the likelihood of a surface at that point on D_i. If we know the surface position (depth) and most likely color, then we can trivially render it from any new view point.

3.1.1 Basic approach (single-pixel support)

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 an image (I_i) of D_i. While it is natural to think of these as two sequential operations, in practice we can combine them into a single homography (plane-to-plane) 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.

From a computer vision perspective, our method is *implicitly* computing a depth map from the new viewpoint C_n using a plane-sweep approach [4, 30, 17, 31]. (Readers who are unfamiliar with computer vision terms are referred to [5].). If we only use two reference cameras and make C_n the same as one of them, for example the first one, one 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 in the second camera, and computing the color variance is similar to computing the sum-of-square-difference (SSD) over a 1×1 window in the second camera. If we use multiple images, our method can be considered in effect a multi-baseline stereo algorithm [27] operating with a 1×1 support window, with the goal being an estimate for the most likely color rather than the depth.

3.1.2 Hardware-based aggregation of consistency measures

Typically, the consistency measures computed over a single pixel support window do not provide sufficient disambiguating power, especially when using only two to three input images. In these cases, it is desirable to aggregate the measures over a large support window. This approach can be implemented very efficiently on a CPU, and by reusing data from previous pixels, the complexity becomes independent of the window size. However on today's graphics hardware, which uses Single-Instruction-Multiple Data (SIMD) parallel architectures, it is not so simple to efficiently implement this type of optimization.

We have several options to aggregate the color consistency measures. One is to use convolution functions in the Imaging Subset of the official OpenGL Specification (Version 1.3) [1]. By convolving a blurring filter with the contents of the frame buffer, we can sum up the consistency measures from neighboring pixels to make color (depth) estimates more robust.

If the convolution function was implemented as a part of the pixel transfer pipeline in hardware, as in the OpenGL specification, there would be little performance penalty. Unfortunately, hardware-accelerated convolutions are only available on expensive graphics workstations such as SGI's Oynx2, but these expensive workstations do not have programmable pixel shaders. On commodity graphics cards available today, convolution is only implemented in software, requiring pixels to be transferred between the main memory and the graphics board. This is not an acceptable solution since such out-of-board transfer operations will completely negate the benefit of doing computation on the graphics board.

There are ways to perform convolution using standard graphics hardware [34]. One could use multiple textures, one for each of the neighboring pixels, or render the scene in multiple passes and perturb the texture coordinates in each pass. For example, by enabling bilinear texture interpolation and sampling in the middle of 4 pixels, it is possible to average those pixels. Note that in a single pass only a summation over a 2×2 window can be achieved. In general such tricks would significantly decrease the speed as the size of the support window becomes larger.

A less obvious option is to use the mipmap functionality available in today's Graphics Processing Units (GPUs). This approach is more general and quite efficient for certain types of convolutions. Modern GPUs have built-in box-filters to efficiently generate all the mipmap levels needed for texturing. Starting from a base image J^0 the following filter is recursively applied:

$$J^{i+1}_{u,v} = \frac{1}{4} \sum_{q=2v}^{2v+1} \sum_{p=2u}^{2u+1} J^{i}_{p,q},$$

where (u, v) and (p, q) are pixel coordinates. Therefore, it is very efficient to sum values over $2^n \times 2^n$ windows. Note that at each iteration of the filter the image size is divided by two. Therefore, a disadvantage of this approach is that the color consistency measures can only be evaluated exactly at every $2^n \times 2^n$ pixel location. For other pixels, approximate values can be obtained by interpolation. However, given the low-pass characteristics of box-filters, the error induced this way is limited.

3.1.3 Multi-resolution approach

Choosing the size of the aggregation window is a difficult problem. The probability of a consistency mismatch goes down as the size of the window increases [25]. However, using large windows leads to a loss of resolution and to the possibility of missing some important image features. This is especially so when large windows are placed over occluding boundaries. This problem is typically dealt with by using a hierarchical approach [7], or by using special approaches to deal with depth discontinuities [12].

Here we will follow a different approach that is better suited to the implementation on a GPU. By observing correlation curves for a variety of images, one can observe that for large windows the curves mostly only have a single strong minimum located in the neighborhood of the true depth, while for small windows often multiple equivalent minima exist. However, for small windows the minima are typically well localized. Therefore, one would like to combine the global characteristics of the large windows with the well-localized minima of the small windows. The simplest way to achieve this in hardware consist of just adding up the different curves. In Figure 4 some example curves are shown for the Tsukuba dataset.

Summing two variance images obtained for windows differing by only a factor of two (one mipmap-level) is very easy and efficient. It suffices to enable trilinear texture mapping and to set the correct mipmap-level bias. Additional variance images can easily be summed using multiple texturing units that refer to the same texture data, but have different mipmap-level biases.

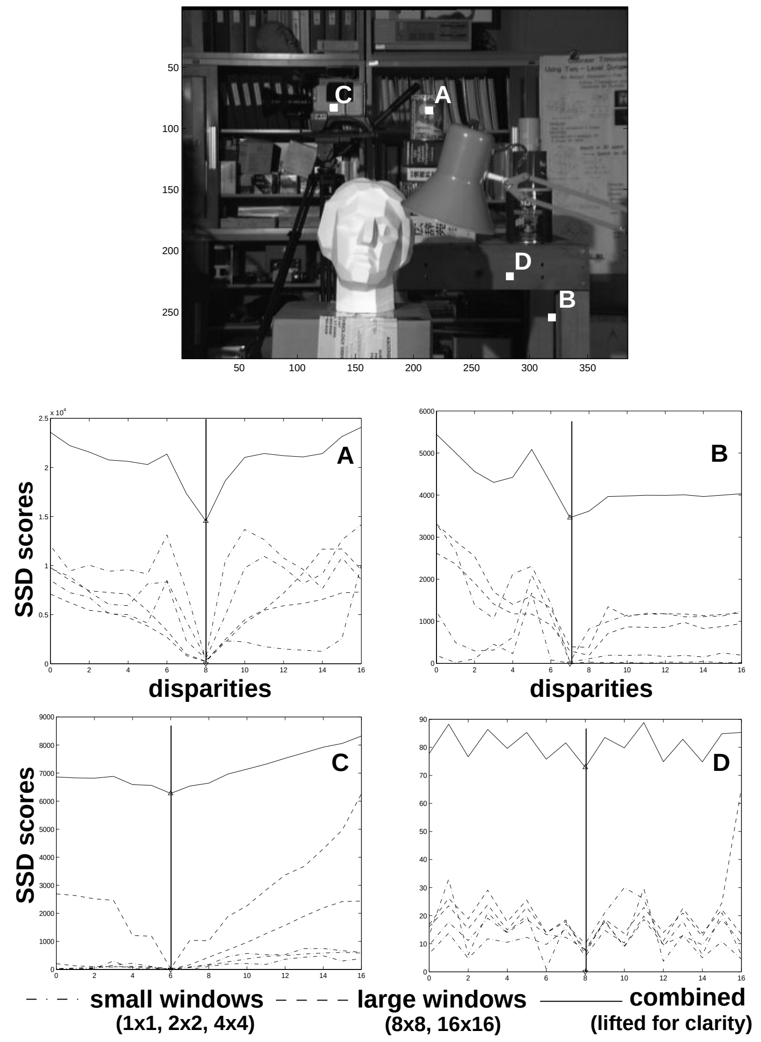

Figure 4: Correlation curves for different points of the Tsukuba stereo pair. Case A represents a typical, well-textured, image point for which SSD would yield correct results for any window size. Case B shows a point close to a discontinuity where SSD with larger windows would fail. Case C and D show low-texture areas where small windows do not capture sufficient information for reliable consistency measures.

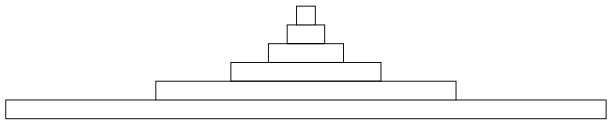

Figure 5: Shape of kernel for summing up six levels.

In fact, this approach corresponds to using a large window, but with larger weights for pixels closer to the center. An example of a kernel is shown in Figure 5. The peaked region in the middle allows good localization while the broad support region improve robustness.

We will call this approach the Multiple Mip-map Level (MML) method. In contrast, the approach that only uses one mip-map level will be called the Single Mip-map Level (SML) method.

In addition to these two variations, we also implemented an additional *min-filtering* step for depth computation only. The use of *min-filter* was proposed in [2, 3]. It is equivalent to replacing each pixel's depth value with the one from its local neighborhood with the best matching measure (the most consistent) in the *final* disparity map. So it suffices to overlay the final depth map with different offsets and select the minimum at each pixel location. This can be applied to both the SML method and the MML method, and it incurs little additional computation.

3.2 OpenGL Implementation Details

While it is straightforward to use texture mapping facilities 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 NVIDIA's GeForce series [26], provide a programable means for per-pixel fragment coloring through the use of *Register Combiner* [16]. We exploit this programmability, together with the texture mapping functions, to carry out the entire computation on the graphics board.

3.2.1 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 three stages of operations, *scoring*, *aggregation*, and *selection*. In the *scoring* stage, we first project the reference images onto the depth plane D_i, then compute the per-pixel mean color and consistency measure. In the second optional *aggregation* stage, consistency measures are aggregated through different mipmap levels. Note that if we only want to use a single mipmap level, which is equivalent to aggregating the consistency measures over a fix-sized support region, we only need reduce the rendered image size proportionally (by changing the viewport setting) in the last selection stage. The automatic mipmap selection mechanism will select the correct mipmap level. A sub-pixel texture shift can also be applied in the last selection stage to increase the effective support size. In the final *selection* stage, the textured D_i, with a RGB color and a consistency measure for each pixel, is rendered into the the frame buffer, and the most consistent color is selected. We show an outline of our implementation in Algorithm 1.

In the first *scoring* stage, 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-square-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 6, 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 optional *aggregation* stage, we first have to copy the frame buffer to a texture, with automatic mipmap generation *enabled*. Then we use the multi-texturing functionalities to sum up different mipmap levels. Note that all texture units should bind to the same texture object but with different settings of the mipmap bias. This is possible because the mipmap bias, as defined in the OpenGL specification, is associated with per texture unit, not per texture object.

In the last *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

Algorithm 1 Pseudo Code for an OpenGL implementation

```
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 OPTIONAL aggregation stage
   // MaxML is the Maxinum Mipmap Level
   if (MaxML > 0)
       sumAllMipLevels(MaxML);

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

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

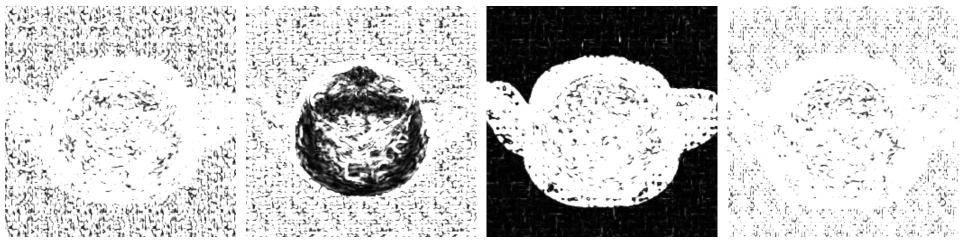
Figure 6: SSD scores (encoded in the alpha channel) for different depth planes in the first *scoring* stage. We use the same setup as in Figure 3. From left to right, the corresponding depth steps are 0, 14, 43, 49.

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.

Complete pseudo code for our hardware-accelerated render is provided in Algorithm 1. 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 aside from executing a few OpenGL commands.

3.2.2 Real-time depth

As explained in Section 3.1.1, 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 3.2.1) 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. In addition, an optional *min-filter* can be applied to the final depth map.

3.3 Tradeoffs and Future Work Using the Graphics Hardware

There are certain tradeoffs we have to make when using the graphics hardware. One 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 recent graphics cards, 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. This is a common problem for most real-time stereo algorithms. In software, we could use the method introduced in the Space Carving algorithm [17] 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 systems. However, this limits the effective view volume, especially for direct view synthesis. We are also exploring a multi-pass approach to address the occlusion problem in general.

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 [22]. In each stage, there is a texture copy operation that copies the frame buffer to the texture memory. We found that texture copies are expensive operations even within the graphics board, especially when the automatic mipmap generation is enabled. We have explored the use of *P-buffer*, an OpenGL extension that allows to render directly to an off-screen texture buffer. There was no performance gain, in fact performance was worse in some cases. We suspect that this extension is still a "work in progress" in the current drivers from NVIDIA. We expect to see a substantial performance boost when this work is done.

Furthermore, the last texture copy for the selection stage can be eliminated with small changes in the graphics hardware. There exists an alpha test in the graphics pipeline, but it only allows comparison of the alpha value (which is the SSD score in our case) to a constant value. It would be ideal to change the alpha test to compare to the current alpha value in the frame

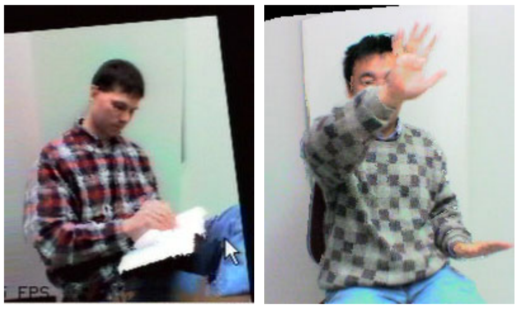

Figure 7: Some live views directly captured from the screen. They are synthesized using five cameras

buffer, in a fashion similar to the depth test. We believe these are opportunities for graphics hardware venders to improve their future products.

4 Experimental Results

We have implemented a distributed system using four PCs and up to five calibrated 1394 digital cameras (SONY DFW-V500). These cameras are roughly 70 mm apart, aiming at a scene about 1200 mm away from the cameras. The camera exposures are synchronized using 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. We have tested our algorithm on three NVIDIA cards, a Quadro2 Pro (a professional version of the GeForce2 card), a GeForce3, and a GeForce4, using all five cameras for view synthesis. Figure 7 shows some live images computed online in real time. Performance comparisons are presented in Table 1 and Figure 8. On average, a GeForce3 is about 75 percent faster than a Quadro2 Pro for our application, and a GeForce4 is about 60 percent faster than a GeForce3.

For comparison purposes, we also ran some tests using different support sizes. The results are shown in Figure 9. Even a small support window (4×4) can make substantial improvements, especially in low texture regions, like the cheek or forehead on the face. The black areas in the depth map are due to the lack of textures on the background wall. They will not impact the synthesized views.

Table 1: Rendering time per frame in milliseconds (number of depth planes (rows) vs. output resolutions (columns)). The numbers in each cell are from a GeForce4, a GeForce3, and a Quadra2 Pro, respectively. All numbers are measured with five 320×240 input images.

	128^2	256^2	512^2
20	9, 16, 40	18, 31, 55	51, 82, 156
50	20, 31, 85	42, 70, 130	120, 211, 365
100	40, 62, 140	84, 133, 235	235, 406, 720

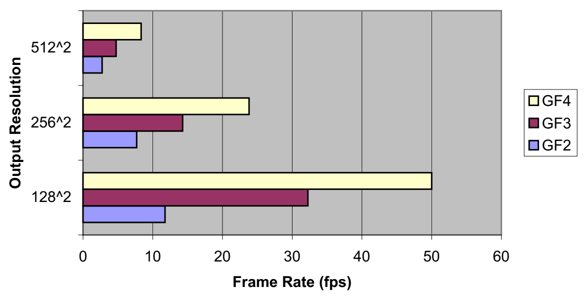

Figure 8: Frame rates from three cards at different output resolutions with 50 depth planes—corresponding to the second row in Table 1.

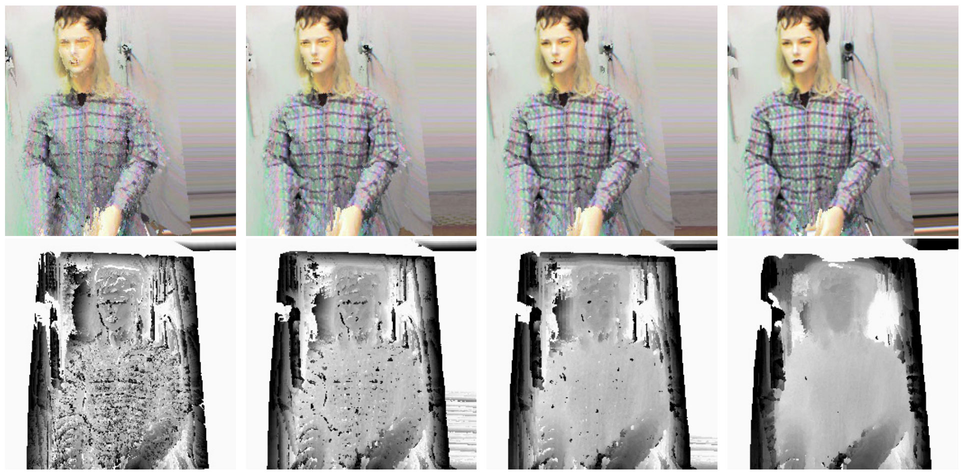

Figure 9: Impact of support size on the color or depth reconstruction(using all five cameras); First row: Synthesized images from an oblique viewing angle (view extrapolation) with different levels of aggregation using the MML method. Second row: extracted depth maps using the MML aggregation method. The erroneous depth values in the background are due to the lack of textures in the image. The maximum mipmap level is set from zero to four, corresponding to a support size of 1×1, 2×2, 4×4, and 8×8, respectively.

4.1 Stereo Results

We also tested our method under the minimal condition—using only a pair of images. Our first test set is the Tsukuba set, which is used in computer vision literature. Note we only tested depth output since this data set includes a ground truth disparity map. The results are shown in Figure 10, in which we show the disparity maps with different aggregation methods. For disparity maps on the left column, we used the SML method so that the SSD image was only rendered at a single mipmap level to simulate a fix-sized box filter. Note that texture shift trick effectively doubles the size of the filter. So it is equivalent to use a 2×2 kernel at mipmap level zero, and a 4×4 kernel at mipmap level one, etc. The right column shows results from using the MML method, i.e., summing up the different mipmap levels. We computed the different disparity maps in each subsequent row by varying the maximum mipmap level(abbreviated as $MaxML$) from zero up to five. We can see that the results using a 1×1 kernel (second row) are almost meaningless. If we use a higher mipmap level, i.e. increase the support size, the results improve for both methods. But the image resolution drops dramatically with the SML method (left column). The disparity map seems to be the best when using the MML method with $MaxML = 4$ (i.e. a 16×16 support). Little is gained when $MaxML > 4$. Results from another widely used stereo pair using $MaxML = 4$ are shown in Figure 11.

In term of performance, we tested our implementation on an NVIDIA GeForce4 Card—a card with four multi-texture units. We found virtually no performance difference when $MaxML$ was changed from one to four. This is not surprising since we can use all four texture units to sum up all four levels in a single pass. If $MaxML$ is set to over four, another additional rendering pass is required, which results in less than 10% increase in calculation time[1]. In practice, we find that setting $MaxML$ to four usually strikes a good balance between smoothness and preserving small details for stereo. Details of the performance data for the MML method can be found in Table 2. Plotting these data in Figure 12, we can see that our algorithm exhibits very good linear performance with respect to the image size.

We also tested our SML method (results shown in Table 3). In this case the frame-rates are higher, especially when going to a higher mipmap level. Note that for higher mimpmap levels the number of evaluated disparities

[1]We do not use trilinear interpolation in our performance testing, and it seems that in practice setting $MaxML$ over four has a detrimental effect on the final result.

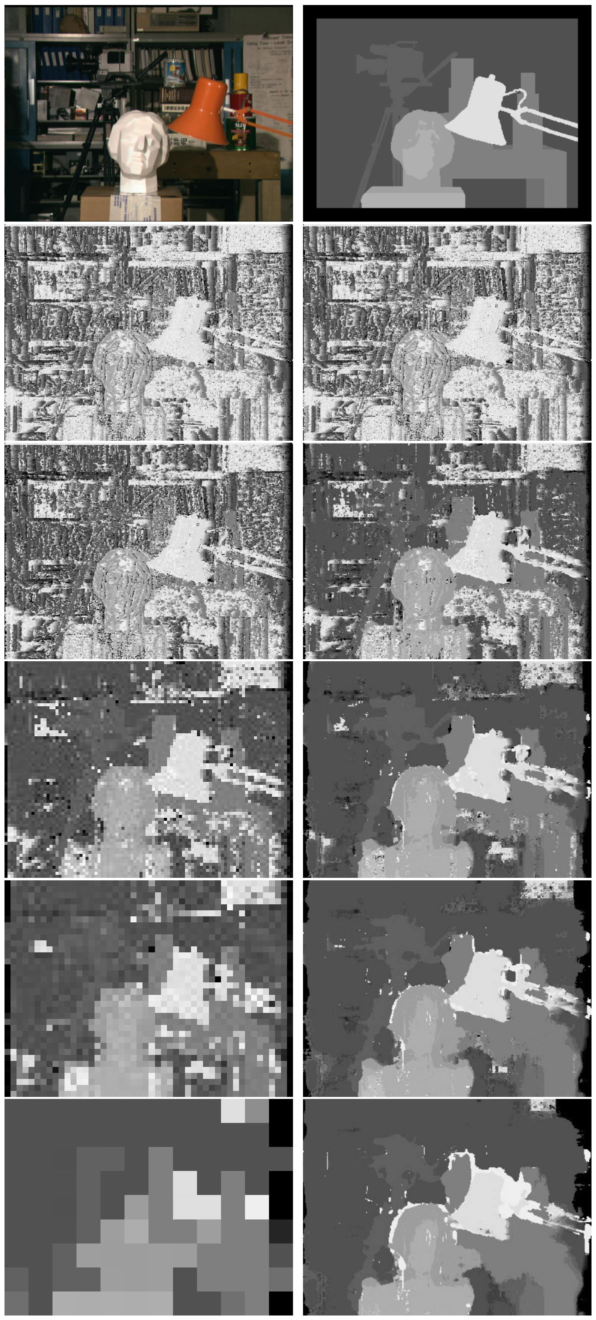

Figure 10: Depth Results on the Tsukuba data set using only two images. The top-right image shows the ground-truth disparity map. For the remaining rows, on the left column we show the disparity maps from the SML method, while on the right we show the ones from the MML method. The mipmap levels are set to zero to five, corresponding to a support window of size 1×1, 2×2, 4×4, etc.

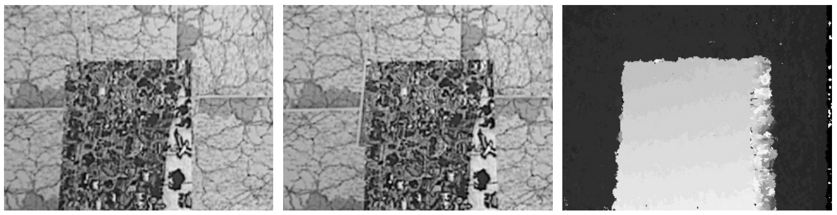

Figure 11: Calculated disparity map from another widely-used stereo pair.

Table 2: Stereo Performance on an NVIDIA GeForce4 card when summing all mipmap levels. The two input images are 640×480, the maximum mipmap level ($MaxML$) is set to 4 in all tests.

Output Size	# of Depth Planes	Times (ms)	Times (Hz)	Img. Update (ms)	Read (ms)	Disp. Calc. (M/sec)
512^2	20	71.4	14	(VGA)		58.9
	50	182	5.50	5.8×2	6.0	65.6
	100	366	2.73			68.3
256^2	20	20.0	50	(QVGA)		53.1
	50	49.9	20	1.6×2	1.5	60.0
	100	99.0	10.1			63.2

per seconds drops because in this case the output disparity map has a lower resolution. This method might be preferred for some applications where speed is more important than detail.

When running under the stereo configuration, our current real-time prototype performs a few additional steps in software, such as radial distortion correction and segmentation[2]. As a proof of concept, these yet-to-be-optimized parts are not fully pipelined with the reconstruction. These overheads slow down the overall reconstruction rate to 6-8 frames per second at 256×256 resolution with 100 depth planes. In Figure 13, we show a sample stereo pair and a reconstructed depth map. In our real-time system, darker colors in the depth map mean that the object is closer to cameras while brighter colors mean further. To better illustrate our results, we also show the reconstructed 3D point cloud from different perspectives in Figure 13. More scenes and their depth maps can be found in Figure 14.

[2]The cameras are facing a white wall with little texture. So we segment the images to fill the background with different colors.

Table 3: Stereo Performance on an NVIDIA GeForce4 card when using only a single mipmap level with texture shift enabled. Throughput decreases proportionally to the output resolution because the majority of the time is spent on computing the SSD score. The overhead includes both the image update time and the time to read back the depth map from the frame buffer.

Base Size	Output Size	# of Depth planes	Times (ms)	Times (Hz)	Overhead (ms)	Disp. Calc. (M/sec)
512^2	128^2 (4×4)	20	2.5	40	12.0	11.7
		50	6.4	15.6		10.7
		100	12.8	7.8		8.86
512^2	256^2 (2×2)	20	28.3	35.3	13.1	31.7
		50	71.4	14.0		38.8
		100	144	6.9		41.7
512^2	512^2	20	40.8	24.5	17.6	89.8
		50	106	9.4		106
		100	207	4.8		117
256^2	128^2 (2×2)	20	12.7	78.7	3.58	20.1
		50	31.6	31.6		23.3
		100	63.1	15.8		24.6
256^2	256^2	20	16.2	61.7	4.7	62.7
		50	40.3	24.8		72.8
		100	80.7	12.4		76.7

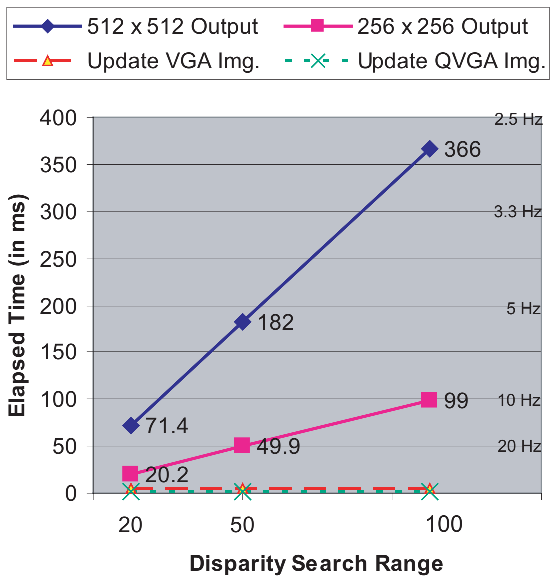

Figure 12: Stereo Performance on a NVIDIA GeForce4 Card. The data are from Table 1.

5 Conclusion

We have presented a view synthesis/depth estimation scheme suitable for implementation on commodity graphics hardware. The heart of our method is to use advanced features in modern graphics hardware to efficiently compute weighted sum-of-square-differences consistency measures from all reference images, and choose the most consistent color (or depth) for each output pixel. We have demonstrated a real-time system implemented solely using OpenGL. When running on a PC with an NVIDIA Geforce4 graphics card, our method exhibits speed and accuracy comparable to some of the fastest commercial stereo systems available.

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.

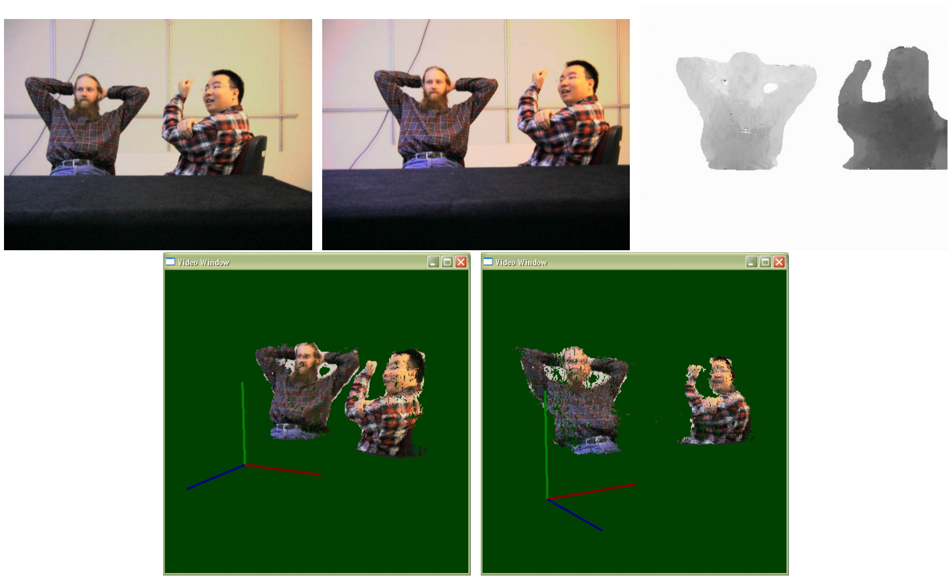

Figure 13: Typical results from our real-time online stereo system. The first row shows the two input images and the disparity map. The second row shows the reconstructed 3D point cloud from different perspectives. Some holes in the 3D views are caused by the rendering. We simply render fix-size (in screen space) points using the GL_POINT primitive.

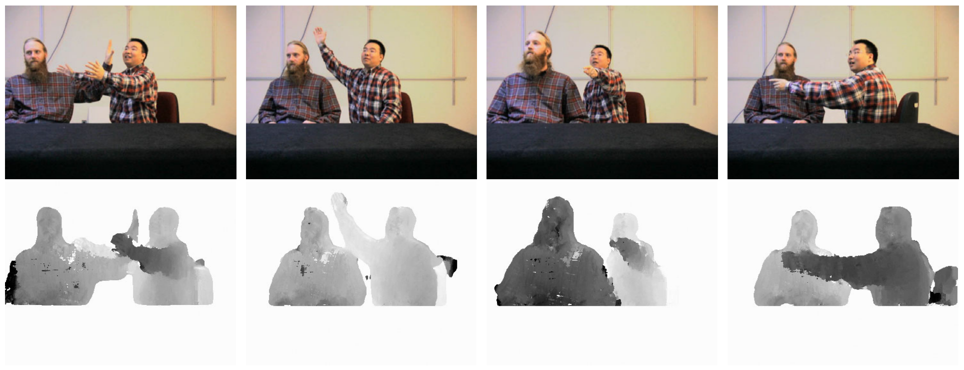

Figure 14: More results from our real-time online stereo system. The first row shows the input images; The second row shows the corresponding disparity map.

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.

In the meantime, we are planning on optimizing our real-time system by also carrying out the radial distortion correction and background segmentation on the graphics hardware. We are also looking at ways to efficiently implement more advanced reconstruction algorithms on graphics hardware. This work will be eased with newer generations of graphics hardware providing more and more programmability.

Like other recent work that makes use of increasingly powerful graphics hardware for non-traditional purposes [13, 32, 18, 21, 10, 9], we hope that our method inspires further thinking and additional new methods to explore the full potentials of modern graphics hardware.

Acknowledgement

This work was supported in part by the United States National Science Foundation grant "Electronic Books for the Tele-Immersion Age" (Grant No. 0121657), grant ISS-0237533, and by a generous 2002-2003 Link Foundation fellowship. We made use of equipment purchased under the United States Department of Energy contract "Front Projection Display Wall Group Teleimmersion Tracking Technologies" (Contract No. 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 Gary Bishop and Herman Towles for their contributions to an early version of this paper. We thank John Thomas and Jim Mahaney for their engineering assistance, and Scott Larsen and Deepak Bandyopadhyay for their participation in the experiments.

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; }
```

References

[1] OpenGL Specification 1.3, August 2001. http://www.opengl.org/developers/documentation/ version13/glspec13.pdf.

[2] R. D. Arnold. Automated Stereo Perception. Technical Report AIM-351, Artificial Intelligence Laboratory, Stanford University, 1983.

[3] A. F. Bobick and S. S. Intille. Large occlusion stereo. *International Journal of Computer Vision (IJCV)*, 33(3):181–200, 1999.

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

[6] O. Faugeras, B. Hotz, H. Mathieu, T. Viville, Z. Zhang, P. Fua, E. Thron, L. Moll, G. Berry, J. Vuillemin, P. Bertin, and C. Proy. Real time sorrelation-based stereo: Algorithm, amplementations and application. Technical Report 2013, INRIA, August 1993.

[7] O. Faugeras, B. Hotz, H. Mathieu, T. Viville, Z. Zhang, P. Fua, E. Thron, L. Moll, G. Berry, J. Vuillemin, P. Bertin, and C. Proy. Real time sorrelation-based stereo: Algorithm, amplementations and application. Technical Report 2013, INRIA, August 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] S. Guha, K. Munagala, S. Krishnan, and S. Venkatasubramanian. The Power of a Two-sided Depth Test and its Application to CSG Rendering and Depth Extraction. In *ACM SIGGRAPH Symposium on Interactive 3D graphics*, 2003. To appear.

[10] Mark J. Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. Physically-Based Visual Simulation on Graphics Hardware. In *Eurographics Workshop on Graphics Hardware 2002*, 2002.

[11] Heiko Hirschmuler. Improvements in Real-Time Correlation-Based Stereo Vision. In *Proceedings of IEEE Workshop on Stereo and Multi-Baseline Vision*, pages 141–148, Kauai, Hawaii, December 2001.

[12] H. Hirschmuller, P. Innocent, and J. Garibaldi. Real-Time Correlation-Based Stereo Vision with Reduced Border Errors. *International Journal of Computer Vision*, 47(1-3), April-June 2002.

[13] 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*. ACM Press, March 2001.

[14] Point Grey Research Inc. http://www.ptgrey.com.

[15] 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.

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

[17] K. Kutulakos and S. M. Seitz. A Theory of Shape by Space Carving. *International Journal of Computer Vision (IJCV),*, 38(3):199–218, 2000.

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

[19] 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.

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

[21] Wei Li, Xiaoming Wei, and Arie Kaufman. Implementing Lattice Boltzmann Computation on Graphics Hardware. Technical Report 010416, Computer Science Department, SUNY at Stony Brook, 2001.

[22] 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.

[23] 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.

[24] J. Mulligan, V. Isler, and K. Daniilidis. Trinocular Stereo: A New Algorithm and its Evaluation. *International Journal of Computer Vision (IJCV), Special Issue on Stereo and Multi-baseline Vision*, 47:51–61, 2002.

[25] H. Nishihara. PRISM, a Pratical Real-Time Imaging Stereo Matcher. Technical Report A.I. Memo 780, MIT, 1984.

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

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

[28] D. Scharstein and R. Szeliski. A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms. *International Journal of Computer Vision*, 47(1):7–42, May 2002.

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

[30] S. M. Seitz and C. R. Dyer. Photorealistic Scene Reconstruction by Voxel Coloring. *International Journal of Computer Vision (IJCV),*, 35(2):151–173, 1999.

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

[32] Chris Trendall and A. James Stewart. General Calculations using Graphics Hardware, with Application to Interactive Caustics. In *Eurographics Workshop on Rendering*, pages 287–298. Springer, June 2000.

[33] Lance Williams. Pyramidal Parametrics. In *Computer Graphics (SIGGRAPH 1983 Proceedings)*, volume 17, pages 1–11, July 1983.

[34] Mason Woo, Jackie Neider, and Tom Davic. *"OpenGL Programming Guide"*. Addison-Wesley, second edition, 1996.

[35] 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.

[36] R. Yang, G. Welch, and G. Bisop. Real-Time Consensus-Based Scene Reconstruction Using Commodity Graphics Hardware. In *Proceedings of Pacific Graphics 2002*, pages 225–234, Beijing, China, October 2002.

[37] Ruigang Yang and Marc Pollefeys. Multi-Resolution Real-Time Stereo on Commodity Graphics Hardware. In *CVPR 2003*, 2003.

Photo and Bibliography

Ruigang Yang is an Assistant Professor in the Computer Science Department at the University of Kentucky. He received his Ph.D. degree in Computer Science from University of North Carolina at Chapel Hill in 2003. Prior to coming to UNC-Chapel Hill, computer he earned a M.S. degree in Computer Science from Columbia University in 1998.

Dr. Yang's research interests include computer graphics, vision, and multimedia. He is a member of the IEEE Computer Society.

Marc Pollefeys is an assistant professor of computer vision in the Department of Computer Science at the University of North Carolina at Chapel Hill. Previously he was a postdoctoral researcher at the Katholieke Universiteit Leuven in Belgium, objects, where he also received his M.S. and Ph.D. degrees in 1994 and 1999, respectively. Dr. Pollefeys joined the faculty at UNC-Chapel Hill in July 2002. One of his main research goals is to develop flexible approaches to capture visual representations of real world scenes and events. Dr. Pollefeys has received several prizes for his research, including the prestigious Marr prize at ICCV '98. He is the author or co-author of more than 70 technical papers. He has organized workshops and courses at major vision and graphics conferences and has served on the program committees of many conferences. He is a regular reviewer for most of the major vision, graphics and photogrammetry journals.

Hua Yang received his Master degree of Pattern Recognition and Artificial Intelligence from Chinese Academy of Science in 2001. From August 2001, he has been a graduate student in Department of Computer Science, Univ. of North Carolina at Chapel Hill.

Greg Welch is a Research Associate Professor in the Computer Science Department at the University of North Carolina at Chapel Hill. His research interests include virtual/augmented environment tracking systems and telecollaboration. Welch graduated with *highest distinction* from Purdue University with a degree in Electrical Engineering Technology in 1986, and received a Ph.D. in computer science from UNC-Chapel Hill in 1996. Prior to coming to UNC, he worked at NASA's Jet Propulsion Laboratory and Northrop-Grumman's Electronic Systems Division. He is a member of the IEEE Computer Society and the ACM.