Accurate Dense Stereo Reconstruction using Graphics Hardware

C. Zach, A. Klaus, M. Hadwiger and K. Karner
VRVis Research Center, Graz/Vienna, Austria

June 5, 2003

Abstract

Vertex programs and pixel shaders found in modern graphics hardware are commonly used to enhance the realism of rendered scenes. Recently these hardware facilities were exploited to obtain interactive non-photorealistic effects and to perform low-level image processing tasks like texture filtering for volume visualization. We exploit modern graphics hardware to accomplish the higher level vision task of dense stereo reconstruction. In our system almost every stage of the matching procedure is executed on 3D graphics hardware, therefore utilizing the parallel vertex and pixel pipelines. Our implementation performs accurate calculations and does not suffer from the limited precision of color channels. On state of the art PC hardware our algorithm requires less than one second to reconstruct a dense mesh with subpixel accuracy for input images with one megapixel resolution.

1 Introduction

The automatic generation of 3D models from digital images is a very active research area. The obtained models can be used for planning tasks, measurement problems and in particular for visualization of virtual environments. Typically the first step to acquire a reconstructed 3D model is the generation of a dense point cloud or detailed mesh suitable for higher-level processing. In this work we describe a dense reconstruction procedure that can be effectively accelerated by graphics hardware. Our approach generates a 3D mesh for objects visible in a pair of stereo images by finding a dense set of corresponding points between the images.

The basic element of our reconstruction software is image warping, i.e. deforming an image in accordance to a (piecewise linear) 2D mapping. The texturing capability of 3D graphics hardware is perfectly suited to execute this step very fast, therefore it is reasonable to utilize this feature of graphics hardware to obtain faster stereo matching methods. Before versatile vertex and fragment programs were available, every other step of dense stereo matching had still to be performed on the main CPU. Such an approach is typically not much faster than a pure software based implementation, since the warped image has to be read back from video memory into main memory. This operation proves to be rather slow. There are additional motivations for moving as many steps as feasible to graphics hardware:

- Most stages of dense stereo matching can be effectively executed in parallel. Performing these steps with a general purpose CPU does not exploit the parallelism.
- For some elementary operations required during the stereo matching procedure (like bilinear pixel access) modern 3D graphics hardware has very fast special purpose circuits.

For these reasons we moved as much of the stereo matching procedure as possible to modern graphics hardware. With the general availability of programmable vertex and pixel shaders we are able to execute almost all stages of our stereo matching method on 3D hardware without the need of transfering a large amount of data between main and video memory.

2 Related Work

2.1 Reconstruction from Stereo Images

Stereo matching is the process of finding corresponding or homologous points in two or more images and is an essential task in computer vision.

We distinguish between feature and area based methods. Feature based methods incline to sparse but accurate results, whereas area based methods permit very dense homologous points. The former methods use local features that are extracted from intensity or color distribution. These extracted feature vectors are utilized to solve the correspondence problem. Area based methods employ a similarity constraint, where corresponding points are assumed to have similar intensity or color within a local window. Frequently a continuity constraint is used as well that results in rather smooth reconstruction. This additional constraint is unavoidable in homogeneous textured regions. If the orientation of the images is known it is also advisable to use the so called epipolar constraint[20]. Each corresponding point in one image has to lie on the projected line of sight of the other images. Therefore only one degree of freedom remains for each corresponding point. This leads to a significant improvement in accuracy as well in performance. A good collection and comparison of different matching methods can be found in[2]. In order to obtain faster convergence and to avoid local minima we employ a hierarchical approach for matching[9, 11], which is in particular inspired by the work of Redert et al.[17]

The Triclops vision system[7] consists of a hardware setup with three cameras and appropriate software for realtime stereo matching. The system is able to generate depth images at a rate of about 20Hz for images up to 320x240 pixels on current PC hardware. The software exploits the particular orientation of the cameras and MMX/SSE instructions available on current CPUs. In contrast to the Triclops system our approach can handle images from cameras with arbitrary relative orientation.

Yang et al.[22, 21] developed a fast stereo reconstruction method performed in 3D hardware by utilizing a plane sweep approach to find correct depth values. The number of iterations is linear in the requested resolution of depth values. Therefore their method is very effective for coarse depth estimation in real-time, but only partially suitable for high quality reconstructions.

2.2 Accelerated Computations in Graphics Hardware

Even before programmable graphics hardware was available, the fixed function pipeline of 3D graphics processors was utilized to accelerate numerical calculations[5, 6] and even to emulate programmable shading[15]. The introduction of a quite general programming model for vertex and pixel processing[10, 16] opened a very active research area. The primary application for programmable vertex and fragment processing is the enhancement of photorealism in interactive visualization systems (e.g. [1, 3]) and entertainment applications ([12, 13]).

Recently several authors identified current graphics hardware as a kind of auxiliary processing unit to perform SIMD (single instruction, multiple data) operations efficiently. Thompson et al.[19] implemented several non-graphical algorithms to run on programmable graphics hardware and profiled the execution times against CPU based implementations. They concluded that an efficient memory interface (especially when transfering data from graphics memory into main memory) is still an unsolved issue. For the same reason our implementation is designed to minimize memory traffic between graphics hardware and main memory. Strzodka[18] discusses the emulation of higher precision numbers with several 8 bit color channels. We faced a similar problem of manipulating large integer values and storing them in the frame buffer and texture maps for later use.

3 Overview of Our Method

The input for our procedure are two relatively oriented gray-scale images suitable for stereo reconstruction and a coarse initial mesh to start with. This mesh can be a sparse reconstruction obtained by the relative orientation procedure (e.g. a mesh generated from a sparse set of corresponding points). In our experiments we use a planar mesh as the starting point for dense reconstruction. One image of the stereo pair is referred as the *key image*, whereas the other one is denoted as the *secondary image*. Consequently the cameras (resp. their positions) are designated as the *key camera* and the *secondary camera*.

The overall idea of the stereo matcher is that if the current mesh hypothesis corresponds to the true model, the appropriately warped secondary image resembles the key image. This similarity is quantified by some suitable error metric on images, which is the sum of absolute difference values in our current implementation. Modifying the current mesh results in different warped secondary images with potentially higher similarity to the key image (see Figure 1). The current mesh hypothesis is iteratively refined to generate and evaluate improved hypotheses. The huge space of possible mesh hypotheses can be explored efficiently, since local mesh refinements have only local impacts on the warped image, therefore many local modifications can be applied and evaluated in parallel.

The matching procedure consists of three nested loops:

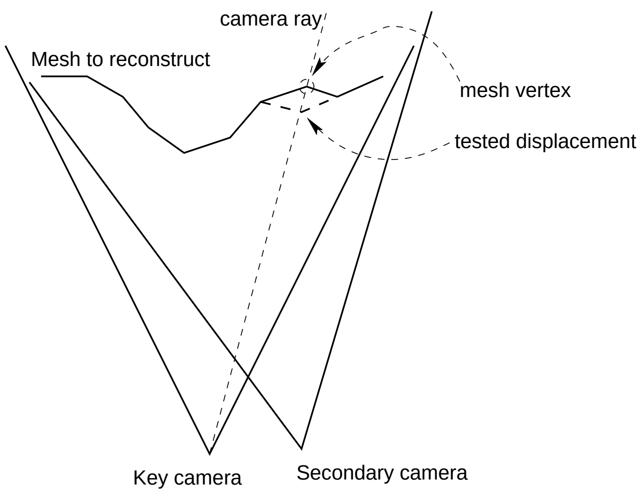


Figure 1: Mesh reconstruction from a pair of stereo images. Vertices of the current mesh hypothesis are translated along the eye ray of the key camera. The image obtained from the secondary camera is warped onto the mesh and the effect in the local neighborhood of the modified vertex is evaluated.

1. The outermost loop determines the mesh and image resolutions. In every iteration the mesh and image resolutions are doubled. The refined mesh is obtained by linear (and optionally median) filtering of the coarser one. This loop adds the hierarchical approach to our method.
2. The inner loop chooses the set of vertices to be modified and updates the depth values of these vertices after performing the innermost loop.
3. The innermost loop evaluates depth variations for candidate vertices selected in the enclosing loop. The best depth value is determined by repeated image warping and error calculation wrt. the tested depth hypothesis. The body of this loop runs entirely on 3D graphics hardware.

To perform image warping the current mesh hypothesis is rendered like a regular heightfield as illustrated in Figure 2. As it can be seen in Figure 3, a change of the depth value of one vertex has only influence on few adjacent triangles. Therefore one fourth of the vertices can be modified simultaneously without affecting each other. The optimization procedure to minimize the error between key image and warped image is a sequence of determining the best depth values for alternating fractions of the mesh vertices. Since vertices of the grid are numbered such that vertices, which are modified and evaluated in the same pass, comprise a connected block (Figure 4), we denote the fraction of vertices to change as a block.

In every step the depth values of one forth of the vertices is modified and the local error between the key image and the warped image in the affected neighbourhood of the vertex is evaluated. For every modified vertex the best depth value is determined and the mesh is updated accordingly. The procedure to calculate and update error values for modified vertices is outlined in Figure 5.

3.1 Image Warping and Difference Image Computation

Since the vertices of the mesh are moved along the eye rays of the key camera, the mesh as seen from the first camera is always a regular grid and mesh modifications do not distort the key image. The appearance of the secondary image as seen from the key camera depends on the mesh geometry.

From the 3D positions of the current mesh vertices and the known relative orientation between the cameras, it is easy to use automatic texture coordinate generation with appropriate coefficients to perform the image warping step. To minimize updates of mesh geometry we use our own vertex program to calculate texture coordinates for the secondary image. This vertex shader is described in more detail in Section 4.1.

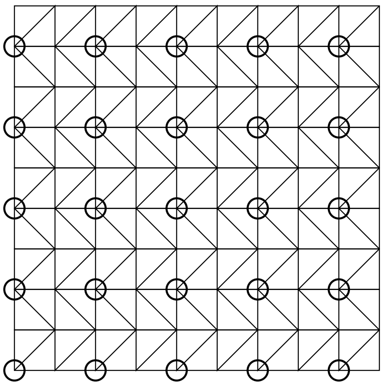


Figure 2: The regular grid as seen from the key camera. This grid structure allows fast rendering of the mesh using triangle strips with only one call. The marked vertices comprise one block. These vertices are shifted on the camera ray and evaluated simulateously in every iteration.

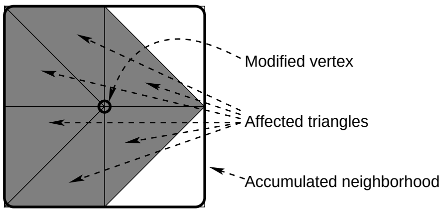


Figure 3: The neighborhood of a currently evaluated vertex. Moving this vertex on the camera ray will only effect the 6 shaded triangles. The actual error for this vertex is calculated for the enclosing rectangle, that is still disjoint with the neighborhoods of all other tested vertices.

3.2 Local Error Summation

After the difference between the key image and the warped image is computed and stored in a pixel buffer, we need to accumulate the error in the neighbourhoods of modified vertices. We perform a repeated downsampling procedure, which sums up four adjacent pixels into one resulting pixel. The target pixel buffer has half the resolution in every dimension of the source buffers. If one vertex is located every four pixels, the downsampling is performed three times to sum the error in an 8 by 8 pixel tile.

The downsampling procedure is rather easy: the input texture is bound to four texture units and a quadrilateral is rendered. The texture coordinates for the 4 texturing units are jittered slightly, such that the correct adjacent pixels are accessed for each final fragment. The filtering mode for the source textures is set to `GL_NEAREST`.

We need to mention that only $2^{n-1} \times 2^{n-1}$ error values are computed for a mesh with $(2^n+1) \times (2^n+1)$ vertices. Vertices at the right and lower edge of the grid do not have an associated error value. For these vertices we set the depth value equal to depth of left resp. upper neighbour.

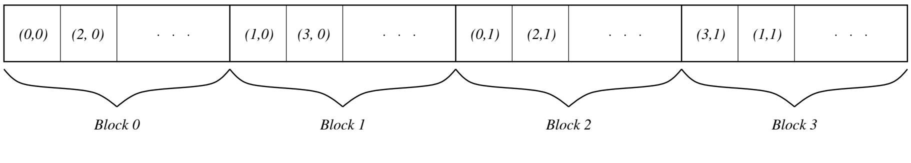


Figure 4: The correspondence between vertex indices and grid positions.

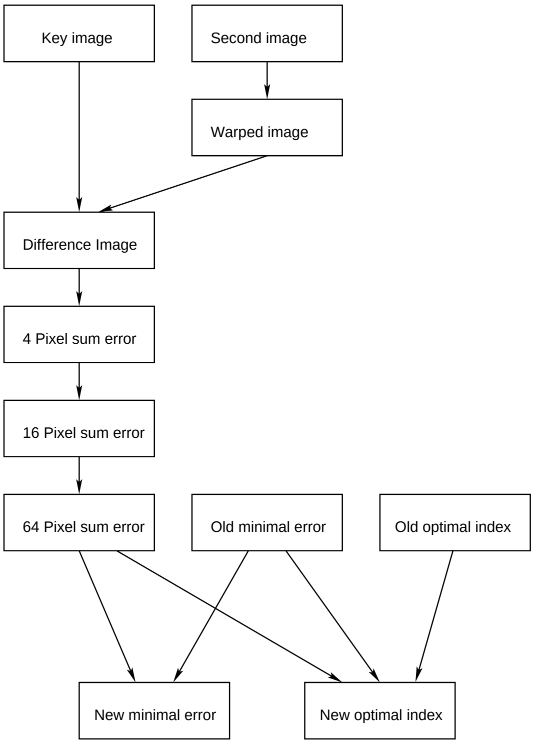


Figure 5: The basic workflow for one iteration. For the current mesh hypothesis a difference image between key image and warped secondary image is calculated in hardware. The error in the local neighborhood of the modified vertices are accumulated and compared with the previous minimal error value. The result of these calculations are minimal error values (stored in the red, green and blue channel) and the index of the best modification so far (stored in the alpha channel). All these steps are executed in graphics hardware and do not require transfers of large datasets between main memory and video memory.

3.3 Determining the Best Local Modification

If δ denotes the largest allowed depth change, then the tested depth variations are sampled regularly from the interval $[-\delta, \delta]$. To minimize the amount of data that needs to be copied from graphics memory to main memory, we do not directly read back the local errors to determine the best local modification in software. We store the currently best local error and the corresponding index in a texture and update these values within an additional pass. These values are read back after all depth variations for one block of vertices are evaluated.

3.4 Hierarchical Matching

In order to avoid local optima during dense matching we utilize a hierarchical approach. The coarsest level consists of a mesh with 9 by 9 vertices and an image resolution of 32 by 32 pixels. The initial model comprise a planar mesh with the approximate correct depth values known from the points of interest generated by our orientation procedure. After a fixed number of iterations the mesh calculated in the coarser level is upsampled (using a bilinear filter) and used as input to the next level. A median filter is optionally applied to the mesh to remove potential outliers especially found in homogeneous image regions.

The largest allowed displacement for mesh vertices is decreased for higher levels to enable higher precision. It is assumed that the model generated at the previous level is already a sufficiently accurate approximation of the true model and at the higher level only local refinements to the mesh are required. In the current implementation we halve the largest evaluated depth variation when entering the next hierarchy level. The corsest level starts with a maximum depth variation roughly equal to the distance of the object to the key camera.

4 Implementation

In this section we describe in more detail some aspects of our approach. Our implementation is based on OpenGL extensions available for the ATI Radeon 9700Pro, namely `VERTEX_OBJECT_ATI`, `ELEMENT_ARRAY_ATI`, `VERTEX_SHADER_EXT` and `FRAGMENT_SHADER_ATI`[4]. These extensions are available on the Radeon 8500 as well, therefore our method can be applied with these older (and cheaper) cards, too. For better reading we sketch the vertex program in *Cg* notation[14].

The major design criterion is to minimize the amount of data transfered between the CPU memory and GPU memory. Especially reading back data from the graphics card is very slow, therefore only absolutly necessary information is copied from video memory.

4.1 Mesh Rendering and Image Warping

For maximum performance we employ the `VERTEX_OBJECT_ATI` and `ELEMENT_ARRAY_ATI` OpenGL extension to store mesh vertices and connectivity information directly in graphics memory. In every iteration one fourth of the vertices needs to be updated to test mesh modifications. In order to reduce memory traffic we update the mesh only after all modifications are evaluated and the best modification is determined. The current tested offset is a parameter to a vertex program, that moves vertices along the camera ray as indicated by the given offset.

Additionally the mesh vertices are ordered such that vertices that are modified in the same pass comprise a single connected block, therefore only one fourth of the vertex array object stored in video memory needs to be updated.

We sketch the vertex program that calculates the appropriate texture coordinates for the secondary image in Figure 6. The vertex attributes consists of the position (`I.position`) and the primary color (`I.color`) encoding the block the vertex belongs to. Program parameters common for all vertices are

1. the currently tested depth `displacement` for the active block,
2. a matrix `M1` transforming pixel positions into eye rays of the first camera,
3. and a matrix `M2` representing the transformation from the first camera into image positions of the second camera.

If a vertex belongs to block i, then the i-th component of the primary color of this vertex is set to one. The other channels are set to zero. If all vertices of block j are currently evaluated, the displacement represented as a 4-component vector has the current offset value at position j and zeros otherwise. Therefore a four-component dot product between the primary color and the given displacement is either the displacement or zero, depending whether the block numbers match.

If A_1 and A_2 are the internal parameters of the key resp. the second camera (arranged in an affine matrix) and R is the relative orientation between the cameras, then `M1` and `M2` are calculated as follows:

$$\texttt{M1} = \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 0 & 1 \\ & & & 1 \end{pmatrix} \times A_1^{-1}$$

and

$$\texttt{M2} = \begin{pmatrix} 1/w & & & \\ & 1/h & & \\ & & 1 & \\ & & 1 & 0 \end{pmatrix} \times A_2 \times R,$$

where w and h represent the image width and height in pixels. If `M1` is applied to a vector $(x, y, \cdot, 1)$, the result is the direction $(\Delta x, \Delta y, 1, 1)$ of the camera ray going through the pixel at (x, y). This direction is scaled by the target depth value to obtain the vertex in the key camera space. Consequently, the vertex data for mesh points consists of vectors $(x, y, z, 1)$, where (x, y) are the pixel coordinates in the key image and z is the current depth value. The obtained texture coordinates (s, t, q, q) for the secondary image are subject to perspective division prior to texture lookup. On current hardware perspective texture lookup is performed for every texel, hence the correct perspective projection (and warping) is achieved.

Additionally we remark, that texture coordinate transformation from one image to another cannot be accomplished only by one transformation matrix: in this case the depth changes are applied in screen space, which maps world coordinates nonlinearly due to perspective division.

```
struct appdata {
    float4 position : POSITION;
    float4 color    : COLOR0;
};

struct v2f {
    float4 HPOS : POSITION;
    float4 TEX0 : TEXCOORD0;
};

v2f main(appdata I,
         uniform float4x4 M1,
         uniform float4x4 M2,
         uniform float4 displacement)
{
    v2f O;

    float4 winPos = I.position;
    winPos.z = 0; winPos.w = 1;

    float4 pos = mul(M1, I.position);

    float oldDepth = I.position.z;
    float delta = dot(displacement, I.color);
    float newDepth = oldDepth + delta;
    float4 newPos = newDepth * pos;

    O.TEX0 = mul(M2, newPos);
    O.HPOS = mul(glstate.matrix.mvp, winPos);

    return O;
} // main
```

Figure 6: The vertex program responsible for warping the secondary image. This vertex shader calculates appropriate texture coordinates for the second image based on the relative orientation of the cameras and the currently evaluated offset.

The described image warping transformation can result in texture coordinates lying outside the secondary image. It is possible to ignore mesh regions outside the secondary image explicitly, but according to our experience simple clamping of texture coordinates is sufficient.

4.2 Encoding of Integers in RGB Channels

Although the input images are grayscale images and one 8 bit gray channel is sufficient to represent the difference image, summation of local errors is likely to generate overflows. The newest generation of graphics cards supports float textures, but at the time of writing no pixel buffer format allowed color channels with floating point precision. Therefore we decided to employ a slightly more complex method to perform error summation with 8 bit RGB channels. In our current implementation no float textures are required.

Our integer encoding assigns the least significant 6 bits to the red channel, the middle 6 bits to the green channel and the remaining bits to the blue channel. The two most significant bits of the red and green channel are always zero. This encoding allows summation of four error values without loss of precision using a fragment program utilizing a dependent texture lookup. After (component-wise) summation of 4 input values the most significant bits of the red and green component of the register storing the sum are possibly set, hence this register requires an additional conversion to obtain the final error value with the desired encoding. This conversion is performed using a 256 by 256 texture map.

If more than four values are summed in one step, the number of spare bits needs to be adjusted, e.g. if 8 values are summed in one pass, the three most significant bits of the red and green channel must be reserved.

5 Results

We tested our hardware based matching procedure on artifical and on real datasets. In all test cases the source images are grayscale images with a resolution of 1024 by 1024 pixels. For the real datasets the relative orientations between stereo images are determined using the method described by Klaus et al.[8]

The artificial dataset comprise two images of a sphere mapped with an earth texture rendered by the Inventor scene viewer (Figure 7). The meshes obtained by our reconstruction method are displayed as point set for easier visual evaluation. Timing statistics for this dataset reconstructed at different resolutions are given in Table 1. The matching procedure performs 8 iterations with 7 tested depth variations for each hierarchy level. These values result in high quality reconstructions in reasonable time. Therefore the pipeline shown in Figure 5 is executed 56 times for each level. The number of levels varies from 4 to 6 depending on the given image resolution.

Hardware	Resolution	Matching time
Radeon 9700 Pro	256x256	0.187s
	512x512	0.312s
	1024x1024	0.71s
Radeon 9000 Mobility	256x256	0.231s
	512x512	0.66s
	1024x1024	2.23s

Table 1: Timing results for the sphere dataset on two different graphic cards.

The real datasets consist of resampled grayscale images of facades with given orientation. The reconstructed models are visualized in Figure 8–9. Note, that wrong matches occur in homogeneous regions showing the sky. Since the number of iterations is equal to the one chosen for the artificial dataset, the times required for dense reconstruction are similar.

6 Conclusion and Future Work

We presented a method to reconstruct dense meshes from stereo images that is almost completely performed in programmable graphics hardware. Pairs of oriented images with one megapixel resolution can be matched from scratch in less than one second on current consumer level hardware.

Future work includes the use of several images to obtain more accurate matching results. Additionaly our framework is applicable to estimate disparity maps instead of mesh geometry, therefore the relative orientation and dense reconstruction can be calculated simulatenously. In this case depth variations along the camera rays are replaced with displacements of mesh vertices of an initially regular 2D grid. This approach can be applied in settings where fast, but not very accurate reconstructions are required (e.g. for robot navigation).

Since our approach is originally targeted to speed up high quality reconstructions, there remains some work to be done to overcome hardware limits in texture and pixel buffer resolutions. Our goal is to obtain dense reconstruction for input images with a resolution of more than 10 Megapixels. The presented current matching procedure has difficulties with homogeneous image regions with little texture. Our CPU based implementation of a high-quality stereo matcher demonstrates, that modified error metrics incorporating a smoothness term give better results in such cases. We added preliminary support for smoother reconstructions (at the expense of slightly longer matching time), but this extension requires still more investigations. The obtained models have better quality in regions with homogeneous textures. These enhancements rely on higher accuracy of color channels found in newest generation graphics hardware.

References

[1] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '01*, pages 9–16, 2001.

[2] O. Faugeras, J. Malik, and K. Ikeuchi, editors. *Special Issue on Stereo and Multi-Baseline Vision. International Journal of Computer Vision*, 2002.

[3] M. Hadwiger, T. Theußl, H. Hauser, and M. E. Gröller. Hardware-accelerated high-quality filtering on PC hardware. In *Proc. of Vision, Modeling and Visualization 2001*, pages 105–112, 2001.

[4] E. Hart and J. L. Mitchell. Hardware shading with EXT_vertex_shader and ATI_fragment_shader. ATI Technologies, 2002.

[5] M. Hopf and T. Ertl. Accelerating 3d convolution using graphics hardware. In *Visualization 1999*, pages 471–474, 1999.

[6] M. Hopf and T. Ertl. Hardware-based wavelet transformations. In *Workshop of Vision, Modelling, and Visualization (VMV '99)*, pages 317–328, 1999.

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

[8] A. Klaus, J. Bauer, K. Karner, and K. Schindler. MetropoGIS: A semi-automatic city documentation system. In *Photogrammetric Computer Vision 2002 (PCV'02)*. ISPRS Commision III Symposium, 2002.

[9] H. S. Lim and T. O. Binford. Structural correspondence in stereo vision. In *Proc. Image Unterstanding Workshop*, volume 2, pages 794–808, 1988.

[10] E. Lindholm, M. J. Kilgard, and H. Moreton. A user-programmable vertex engine. In *Proceedings of SIGGRAPH 2001*, pages 149–158, 2001.

[11] C. Menard and M. Brändle. Hierarchical area-based stereo algorithm for 3D acquisition. In *Proceedings International Workshop on Stereoscopic and Three Dimensional Imaging*, pages 195–201, 1995.

[12] J. L. Mitchell. Hardware shading on the Radeon 9700. ATI Technologies, 2002.

[13] NVidia Corporation. http://developer.nvidia.com.

[14] NVidia Corporation. Cg language specification, 2002.

[15] M. S. Peercy, M. Olano, J. Airey, and P. J. Ungar. Interactive multi-pass programmable shading. In *Proceedings of SIGGRAPH 2000*, pages 425–432, 2000.

[16] K. Proudfoot, W. Mark, S. Tzvetkov, and P. Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of SIGGRAPH 2001*, pages 159–170, 2001.

[17] A. Redert, E. Hendriks, and J. Biemond. Correspondence estimation in image pairs. *IEEE Signal Processing Magazine*, pages 29–46, 1999.

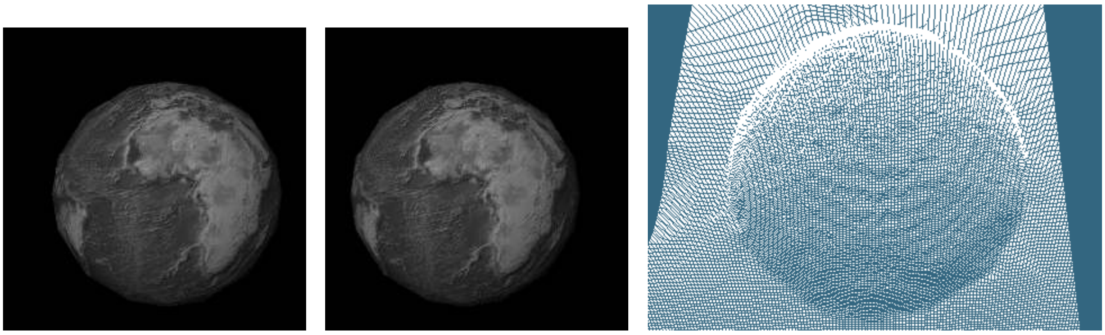
[18] R. Strzodka. Virtual 16 bit precise operations on rgba8 textures. In *Proc. of Vision, Modeling and Visualization 2002*, pages 171–178, 2002.

[19] C. J. Thompson, S. Hahn, and M. Oskin. Using modern graphics architectures for general-purpose computing: A framework and analysis. In *35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)*, 2002.

[20] J. Weng. Camera calibration with distortion models and accuracy evaluation. *IEEE Trans. Patt. Anal. Machine Intell.*, 14:965–980, 1992.

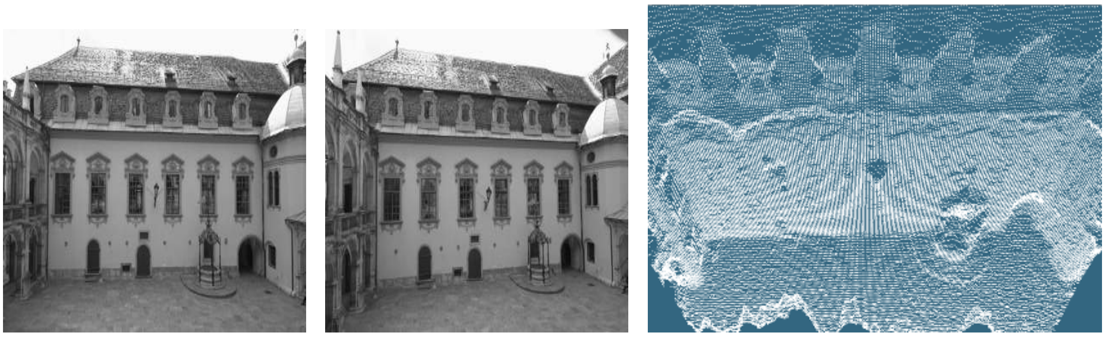
[21] R. Yang and M. Pollefeys. Multi-resolution real-time stereo on commodity graphics hardware. In *Conference on Computer Vision and Pattern Recognition*, 2003.

[22] R. Yang, G. Welch, and G. Bishop. Real-time consensus based scene reconstruction using commodity graphics hardware. In *Proceedings of Pacific Graphics*, 2002.



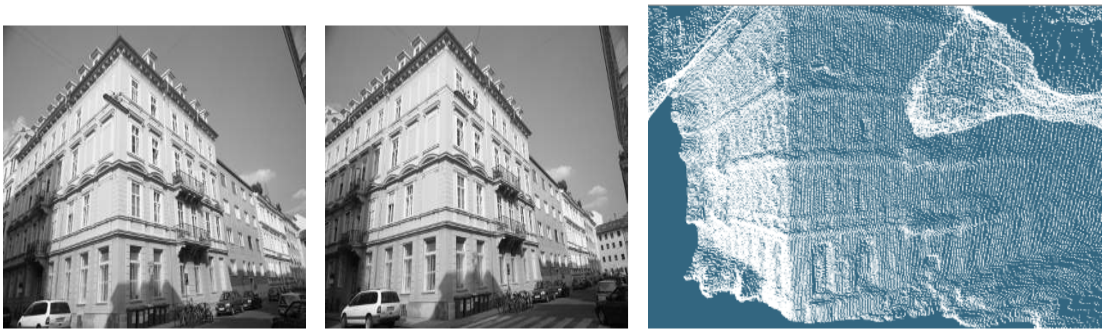
(a) The key image (b) The second image (c) The reconstructed model

Figure 7: Results for the artifical earth dataset.



(a) The key image (b) The second image (c) The reconstructed model

Figure 8: Results for a dataset showing a historic building.



(a) The key image (b) The second image (c) The reconstructed model

Figure 9: Results for a dataset showing an apartment house.