GLStereo: Stereo Vision Implemented in Graphics Hardware

Dustin Lang and James J. Little

Department of Computer Science
University of British Columbia, Canada
{dalang,little}@cs.ubc.ca

Abstract. We present an implementation of the standard sum of absolute differences (SAD) stereo disparity algorithm, performing all computation in graphics hardware. To our knowledge, this is the fastest published stereo disparity implementation on commodity hardware. With an inexpensive graphics card, we achieve 'raw' SAD performance above 170 MPDS (mega-pixel disparities per second), corresponding to 5×5 neighbourhoods, 640×480 pixel images, 54 disparities, 10 frames per second (fps) (or 320×240 pixels, 96 disparities, 25 fps). The CPU is approximately 90% idle while this computation is being performed. Other authors have presented stereo disparity implementations for graphics hardware. However, we focus on filtering the raw results in order to eliminate unreliable pixels, thereby decreasing the error in the final disparity maps. Since the standard SAD algorithm produces disparity maps with relatively high error rates, such filtering is essential for many applications. We implement shiftable windows, left-right consistency, texture, and disparity smoothness filters, all using graphics hardware. We investigate the accuracy/density tradeoff of the latter three filters using a novel analysis. We find that the left-right consistency and smoothness filters are particularly effective, and using these filters we achieve performance above 110 MPDS: 640×480 pixel images, 36 disparities, 10 frames per second (or 320×240 pixels, 66 disparities, 25 fps). This level of performance demonstrates that graphics cards are powerful co-processors for low-level computer vision tasks.

1 Introduction

Current commodity graphics cards incorporate significant computational resources. Indeed, they are now called Graphics Processing Units (GPUs), and form a complete coprocessing subsystem, with fast parallel hardware and large local memories. As GPUs have become more powerful, they have also become more programmable. This paper demonstrates that recent generations of graphics cards provide sufficient programmability to be used for low-level computer vision tasks such as stereo disparity computation.

Using the GPU as a coprocessor for computer vision applications has several advantages. Since the computation is performed in graphics hardware, the CPU can perform other computation in parallel. In robotics applications, for example, the CPU could perform path planning, mapping, or manipulator control while the GPU computes stereo disparity. There are advantages to using a GPU rather than a multi-processor machine or several networked machines. The GPU is connected to the CPU by a fast bus so the communication overhead is small, latency is low, and reliability is high compared to the networked case. The GPU has its own local memory, so no access to main memory is required during computation, unlike the multi-processor case, in which the processor performing stereo disparity would require many memory accesses. Finally, graphics cards are readily available and inexpensive devices.

2 Related Work

Scharstein and Szeliski [1] provide a survey and taxonomy of dense stereo algorithms. The algorithm presented here is essentially the "traditional" sum-of-absolute-differences (SAD) algorithm, plus filtering steps which are also well known. We evaluate our algorithm using their framework, which was designed for stereo algorithms that produce disparity estimates for every image pixel. However, we have focused on filtering our results in order to remove erroneous disparity estimates, at the expense of producing non-dense results. We therefore present an alternate analysis of our algorithm.

Many stereo disparity algorithms are simple in structure and require only basic operations. This has allowed researchers to develop many stereo disparity systems in specialized hardware. For example, Kanade *et. al* present a DSP-based system that performs sum-of-squared-differences (SSD) and achieves 30 MPDS (mega pixel-disparities per second) [2, 3]. Woodfill *et. al* present an FPGA-based implementation that performs the census transform and achieves 113 MPDS [4, 5]; the commercial *DeepSea* ASIC implementation claims performance above 1700 MPDS[1]. Many other examples exist.

Yang and Pollefeys [6] present a graphics-hardware implementation that uses centre-weighted sum-of-squared-differences (SSD). They construct an image pyramid of SSD values and aggregate error across several levels of the pyramid. They achieve speeds comparable to our implementation. However, they do not

[1] `http://www.tyzx.com/Systems.shtml`, accessed Oct 3, 2003.

attempt to filter their results, so their disparity images have the relatively high error rate typical of the SAD and SSD methods.

Zach *et. al* [7] present an implementation that uses the graphics hardware to perform the inner loop of an iterative mesh-refinement method. Since they use the projective texturing capabilities of the graphics hardware and a coarse-to-fine mesh strategy, they can handle a large range of disparities. They generate a disparity estimate at each mesh point, with one mesh point for each four pixels.

In contrast to these two implementations, we perform real-time pixel-based SAD disparity computation followed by a filtering stage which decreases the error significantly.

3 Implementation

We first describe the structure and capabilities of the graphics subsystem as exposed by the OpenGL[2] interface. We then describe our algorithm and its implementation in graphics hardware.

3.1 Graphics Subsystem

OpenGL provides a cross-platform and cross-hardware interface to the graphics subsystem. We treat the OpenGL graphics subsystem as a single-instruction multiple-data (SIMD) coprocessor.

Our view of the OpenGL graphics subsystem is shown in Figure 1. The CPU and main memory are connected to the graphics system through the Accelerated Graphics Port (AGP). The graphics subsystem is composed of texture memory, texturing units, display lists, the rendering engine, and the framebuffer.

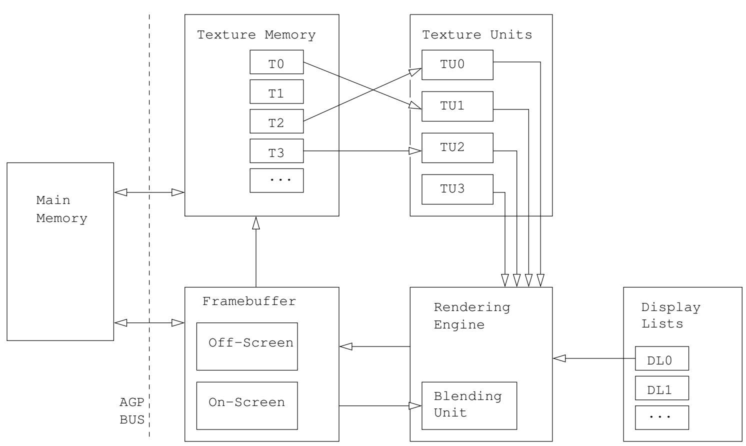

Fig. 1. Our view of the OpenGL graphics subsystem.

[2] See `http://www.opengl.org`.

For our purposes, texture memory holds the input images and intermediate results. The basic data element is a *texture*[3], which is an array of pixels (called *texels*) organized into a (small) number of channels (Red, Green, Blue, and Alpha, typically). The number of channels and the size of the texture are set when the texture is allocated. The number of textures that can be handled efficiently is limited only by the memory on the graphics card.

Texturing is a fundamental operation in graphics processing. Consider a texture to be a rubber sheet on which texels are drawn. When rendering a polygon to the framebuffer, we can specify that each corner of the polygon is attached to some point on the texture. Texturing is the process of 'stretching' the rubber sheet to fit the polygon. For our purposes, texturing allows us to retrieve data stored in texture objects. For example, if we render a rectangle to the framebuffer and specify the texture coordinates such that the corners of the texture are attached to the corners of the rectangle (the rubber sheet is unstretched), then for each rendered pixel, one texel will be retrieved from the texture.

Texturing is performed by *texture units*. Each texture unit is *bound* to one texture. The number of texture units is fixed by the hardware, and current GPUs provide four units which can be dynamically bound to textures. By using multiple texture units, we can retrieve data from several texture objects and perform useful computations.

Display lists are compiled instructions for the GPU which are stored in graphics memory. To execute a display list, the CPU merely refers to its unique identifier, which requires very little data transfer across the AGP bus.

The *rendering engine* is the computational unit of the graphics subsystem. It receives data from the texture units and instructions from the display lists, and places its output in the framebuffer. One part of the rendering engine, the *blending* section, allows the output from the rendering engine to interact with data already in the framebuffer.

The *framebuffer* receives the output from the rendering engine. A portion of the framebuffer can be displayed on the screen; there can also be several offscreen framebuffer regions.

OpenGL sub-programs are generally composed of two phases: first, the state of the texturing units and rendering engine is specified. Then, a set of polygons is rendered to the framebuffer. Each polygon specifies a set of pixel locations to be rendered. We typically render a single polygon - a rectangle the same size as the input image - which passes the entire image through the pixel pipeline.

Our implementation relies on several extensions to the OpenGL specification by NVIDIA Corporation[4] which provide flexible computational capabilities in the rendering engine. One of these, *register combiners*, is particularly important. The register combiners extension presents the rendering engine as a chain of register sets, separated by processing stages called *combiners*. See Figure 2. Each

---

[3] The term "texture" is used by both the vision and graphics communities, with different meanings. For clarity, we will hereafter use the phrase "feature density" rather than "texture" in the vision sense.

[4] See `http://www.nvidia.com`.

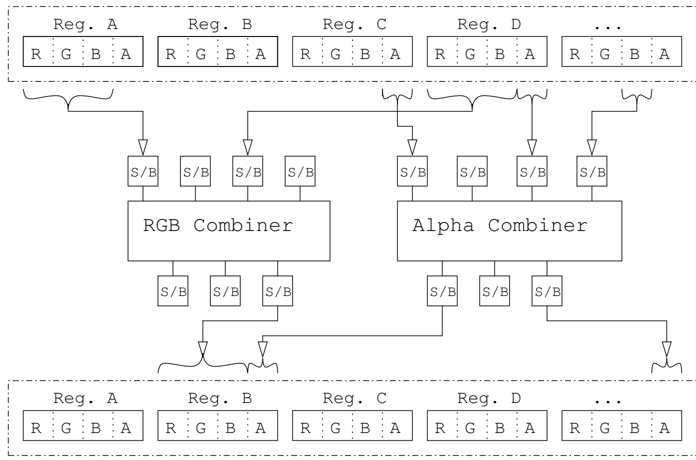

Fig. 2. One stage of register combiners. The top dotted box contains one set of registers; the bottom dotted box contains the next register set. The RGB and Alpha combiners are shown in the middle. The arrows show data flowing from selected input registers, through scale/bias (S/B) units to the combiners, then through S/B units to the next stage of registers.

register set contains eight read-write registers and three read-only registers. The registers in the first stage are initialized with data values fetched by the texture units. Each pixel to be rendered is passed through the chain of register combiners. The order in which pixels are processed is not specified, and each pixel is processed independently: no inter-pixel state can be kept in the registers.

Each stage includes RGB and Alpha combiners. Combiners can accept four inputs, perform one of several operations, and produce up to three outputs. The source and destination registers are selectable, and scaling, bias, and clamping (range restriction) can be performed on both input and output values. The RGB combiners can read and write the RGB colour channels, while the Alpha combiners can read from the Alpha and Blue channels and write to the Alpha channel.

The operations that can be performed in individual register combiners are rather limited, but with several stages it is possible to compose more complex operations. Current GPUs provide eight register combiner stages. The register operations include multiplication, addition, dot product, and multiplexing. The multiplex operation is similar to the `C` construct $Output = (Value \geq \frac{1}{2})\ ?\ A : B$. This operation is quite versatile and allows such operations as thresholding and simple decisions.

3.2 Algorithm

Our algorithm is:

1. **Load input images** from main memory into texture memory.

2. **Pre-process images** to take advantage of hardware parallelism.
3. **Compute disparities**: for each disparity,
   (a) **Compute absolute difference** between stationary left image and shifted right image.
   (b) **Neighbourhood average** the difference image.
   (c) **Compute new winner (left)**: check if this disparity is the new winner in the left reference frame. If so, store the new lowest error and best disparity.
   (d) **Optionally, compute new winner (right)**: if left-right consistency filtering is enabled, store the new winning error and disparity.
4. **Optionally, perform min-filtering**: select the disparity with the lowest error within a neighbourhood around each pixel.
5. **Optionally, check consistency**: perform left-right consistency check, marking invalid pixels.
6. **Optionally, compute feature density measure** and filter.
7. **Optionally, perform disparity smoothness filtering**: detect regions of large disparity change and mark them as invalid. Also, mark the neighbours of invalid pixels as invalid.
8. **Read back** disparity image to main memory.

Load input images. We first move the left and right input images into texture memory.

Pre-process images. The graphics hardware provides parallel processing of Red, Green, and Blue colour channels, so we can check three disparities at once. For the left input image, we duplicate the grayscale image in all three channels. For the right image, we place shifted copies of the image in the three colour channels.

Compute absolute difference. We bind the left and right textures to two texture units, and find the absolute difference using a simple register combiner program,

Neighbourhood average. Next, we perform convolution of the absolute difference (error) image. Although OpenGL provides a convolution operation, we have found that it is not implemented in an optimal manner, so we do our own convolution using a register combiner program and blending. In CPU code, the 'sliding average' optimization is used, which allows the $N \times N$ neighbourhood average to be computed in $O(1)$ time per pixel. Expressing this optimization in OpenGL code is problematic, since much less parallelism is possible: instead of processing an image at a time, we must process the image one column or row at a time. The resulting "optimization" has larger error and, for filters of size less than about 30, is slower than simply performing a general convolution. Therefore, this step of our algorithm requires $O(N)$ time per pixel. This also means that we could convolve with a non-constant (for example, Gaussian) kernel, if so desired, at no extra cost.

Compute new winners. After the error image has been calculated, we perform the winner-takes-all step. Of the three disparities currently being checked, if the smallest error is smaller than the current winner, then that disparity becomes the new winner and its error and disparity values are rendered to the framebuffer. This step is executed by a fairly complicated register combiner program.

Perform min-filtering. At depth boundaries, square neighbourhood windows aggregate error from disparate regions of the scene. Part of the window will have small error at one disparity, while other parts of the window will have large error at this disparity but have small error at a different disparity. The total error large, and the error minimum is unlikely to lie at the correct disparity. One way of alleviating this problem is to allow the neighbourhood window to shift from its centre, so that instead of straddling a depth boundary, it can move into one region or the other. Shiftable windows can be implemented with a min-filter [8, 9].

Consistency check. The disparities in the left and right reference frames are checked for consistency. Fua describes this filter [10], which is similar to the uniqueness constraint described by Marr and Poggio [11].

Compute feature density measure. In large featureless regions of the image, small neighbourhoods appear very similar so winner-takes-all matching produces unreliable results. We attempt to detect and invalidate these regions.

To measure feature density, we average the image horizontally, take the absolute difference from the original image, then average the difference image. This measure assigns high values to regions that have sharp horizontal variations in intensity, such as lines and edges, and low values to regions that are nearly uniform.

Perform smoothness filtering. Marr and Poggio note that in most regions of the image, disparity should vary smoothly [11]. We make an additional observation: often, pixels neighbouring incorrect pixels are themselves incorrect. If the sum of absolute differences between a disparity pixel and its neighbours is larger than the filtering threshold, or if any of the neighbours are marked as invalid, we mark the pixel invalid. Due to hardware limitations, we perform separate horizontal and vertical filtering passes, examining only two neighbours in each pass. In our experiments, this produces results not appreciably different than examining four neighbours at once.

Read back. We read the disparity image from the framebuffer back to main memory.

4 Performance

Scharstein and Szeliski [1] provide a framework to evaluate stereo disparity algorithms. They focus on applications in which dense results are required, such as image-based rendering. For applications such as robot navigation, accuracy is often more important than density - we prefer to have fewer disparity values and be more confident that they are correct [12].

Our implementation was developed for such applications, so has several features that help to detect and reject incorrect results, producing non-dense but more accurate disparity maps. In order to use the Scharstein and Szeliski framework, however, we are forced to choose some disparity value for the pixels we have marked as incorrect. We set these pixels to the average of their valid neighbours, iterating until all invalid pixels have been filled. This section will first present our implementation's results in this framework, then present further analysis and results when sparseness is allowed.

4.1 Dense Results

Scharstein and Szeliski specify three primary error metrics in their evaluation framework. $B_{\overline{\mathcal{O}}}$ is defined as the proportion of "bad" pixels in non-occluded regions of the image, where "bad" means that the absolute difference between the computed and ground-truth disparities is greater than one pixel. $B_{\overline{\mathcal{T}}}$ is defined as the proportion of bad pixels in "textureless" ("regions of low feature density", in our terminology), non-occluded image regions. $B_{\mathcal{D}}$ is defined as the proportion of bad pixels near disparity boundaries, in non-occluded regions. $B_{\overline{\mathcal{O}}}$ is presented as an overall performance measure, and $B_{\overline{\mathcal{T}}}$ and $B_{\mathcal{D}}$ describe performance in image regions that are prone to errors for local disparity algorithms.

Our implementation has several parameters that allow accuracy and density to be traded off. We searched the parameter space for the optimal values, based on the $B_{\overline{\mathcal{O}}}$ error metric. We present our results in Table 4.1 and Figure 3.

	Tsukuba			**Sawtooth**			**Venus**			**Map**	
	$B_{\overline{\mathcal{O}}}$	$B_{\overline{\mathcal{T}}}$	$B_{\mathcal{D}}$	$B_{\overline{\mathcal{O}}}$	$B_{\overline{\mathcal{T}}}$	$B_{\mathcal{D}}$	$B_{\overline{\mathcal{O}}}$	$B_{\overline{\mathcal{T}}}$	$B_{\mathcal{D}}$	$B_{\overline{\mathcal{O}}}$	$B_{\mathcal{D}}$
GLStereo (Joint)	7.01	9.11	18.32	1.76	0.26	12.41	2.74	4.52	12.92	1.10	15.17
GLStereo (Best)	5.88	7.83	21.06	1.49	0.11	10.91	1.54	1.84	13.01	0.47	5.85
SSD+MF	5.23	3.80	24.66	2.21	0.72	13.97	3.74	6.82	12.94	0.66	9.35

Table 1. Performance of *GLStereo* in the Scharstein and Szeliski [1] framework. The first row gives the best results with parameters optimized over all four image pairs; the second row gives the best results with parameters optimized for each image pair individually. The third row is reproduced from [1], Table 5, and gives results for "traditional" sum of square differences, with min-filter and 21-by-21 block size. Since our implementation is similar to this algorithm, the results are comparable.

4.2 Non-Dense Results

We developed our implementation with the intent of using it for autonomous robot navigation. For this task, it is preferable to mark a region of the image as uncertain, rather than produce erroneous disparity results. Our implementation includes several filters that allow unreliable results to be detected and marked as invalid. These filters are all implemented using the graphics hardware. This section presents these results.

In our analysis, we categorize the pixels in the density image into three types: invalid, good (correct), and bad (incorrect). We use the same definition of bad pixels as Scharstein and Szeliski. A perfect invalidation filter moves pixels from

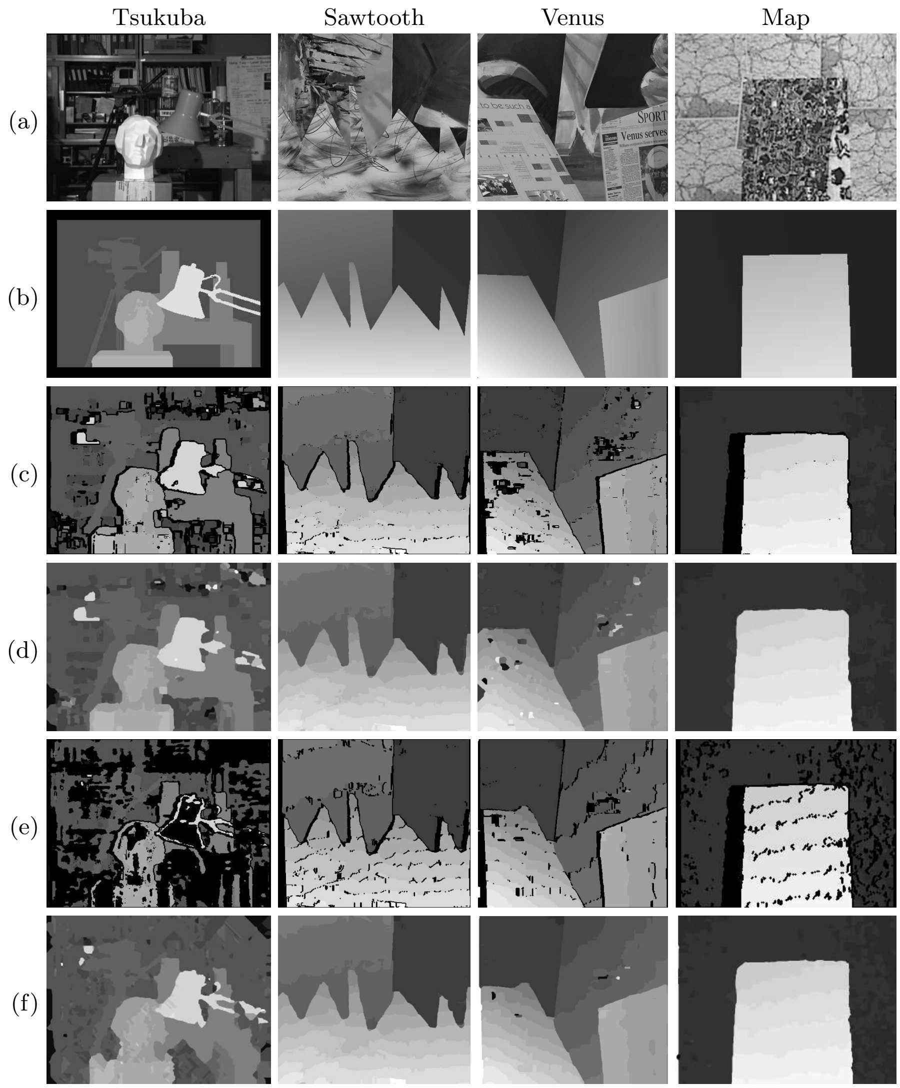

Fig. 3. *GLStereo* results: (a) left images, (b) ground-truth disparities. *GLStereo*, with parameters that minimize the sum of $B_{\overline{\mathcal{O}}}$ errors, (c) with invalidated regions marked in black and (d) with invalidated regions filled with the average of their valid neighbours. *GLStereo*, with parameters that minimize $B_{\overline{\mathcal{O}}}$ for each image individually, (e) with invalidated regions marked in black and (f) invalidated regions filled.

the bad class to the invalid class. Non-ideal filters will move some good pixels into the invalid class as well.

We describe three invalidation filters: left-right consistency, feature density, and smoothness. Min-filtering is not an invalidation filter (since it never marks pixels as invalid but rather adjusts the values of pixels), so is not analysed here. These three filters have thresholds that determine how many pixels will be filtered. To observe the effectiveness of our filters, we can plot error rate against density. This plot is similar to the *Precision-Recall* plot commonly used in Information Retrieval.

We define density D and error rate E as

$$D = \frac{G+B}{G+B+I} \qquad \text{and} \qquad E = \frac{B}{G+B+I} \quad ,$$

where G, B, and I are the number of pixels in the good, bad, and invalid classes, respectively. $G + B + I$ is a constant. Note that the definition of E is somewhat non-intuitive: it is the number bad pixels as a fraction of the total number of pixels, rather than of the number of valid pixels $(G + B)$. We have chosen this definition because it allows us to express the performance of best-case, worst-case, and random-case filters with clean analytic expressions.

The output of the SAD algorithm becomes the input to the filtering stage. This high-density, high-error initial point appears in the upper right of the plot. A perfect filter will move pixels from the bad class to the invalid class, producing a line with slope one. The worst-case filter will move pixels from the good class to the invalid class, producing a line with slope zero. A filter that invalidates pixels randomly will produce a line from the initial point to the origin.

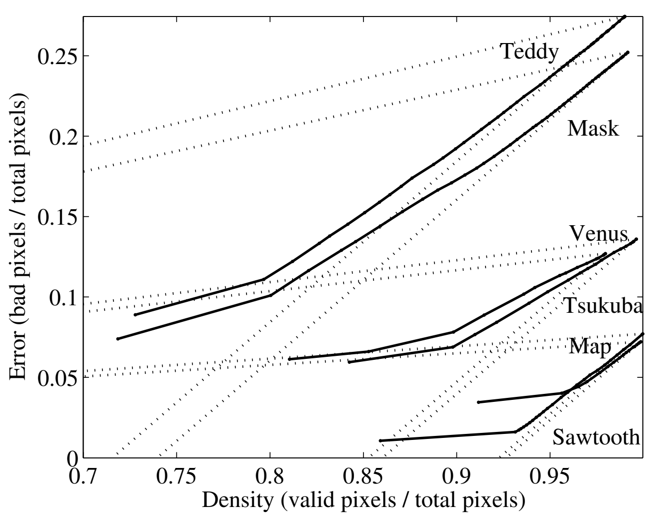

Fig. 4. Consistency filter performance for the six test images. The rightmost data point for each image is the raw disparity result. The upper and lower dotted lines show the performance of a random-case and best-case filter, respectively.

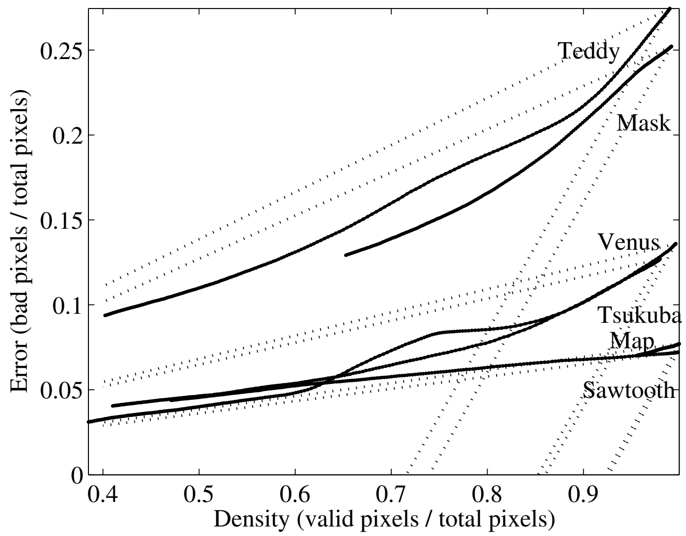

Fig. 5. Feature density filter performance for the six test images.

The performance of the consistency, feature density, and smoothness filters are shown in Figures 4, 5, and 6, respectively. The best-case (bottom) and random-case (top) performance lines are shown for each test image. The consistency and smoothness filters show similar performance: initially (at high density), they operate very effectively. As the filtering threshold allows more pixels to be filtered, the performance drops, eventually reaching random-case behaviour. It seems that these filters are very effective at eliminating the 'most egregious' (according to their filtering criteria) bad pixels, but as more pixels are filtered, they become less able to distinguish good pixels from bad.

The feature density filter does not show the same highly-effective performance of the other filters, but rather shows mediocre performance, followed by performance no better than random. In the case of the 'map' and 'sawtooth' images, which have few areas with low feature density, the filter is almost completely ineffective.

4.3 Speed

In the stereo literature it is common to cite performance figures in units of mega-pixel disparities per second (MPDS). Our algorithm, however, depends in significant part upon several other parameters, so the MPDS figure can vary across nearly an order of magnitude. It is therefore not a particularly good performance metric, except insofar as it allows comparison to other published results.

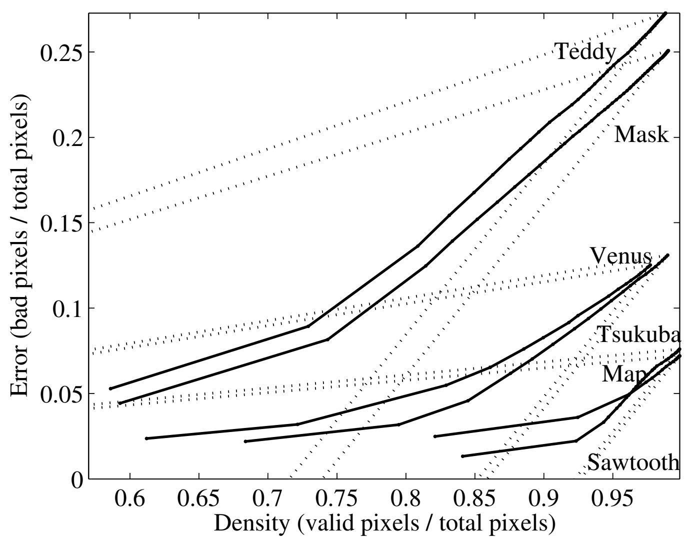

Fig. 6. Smoothness filter performance for the six test images.

Image Size	D	N	Raw results Time (ms)	Raw results Speed (MPDS)	Filtered results Time (ms)	Filtered results Speed (MPDS)
320 × 240	30	3	12.4	185	19.3	119
320 × 240	30	7	16.8	137	23.6	97.5
320 × 240	30	11	21.1	109	28.0	82.3
640 × 480	30	3	49.0	188	76.3	121
640 × 480	30	7	67.0	138	94.3	97.8
640 × 480	30	11	85.4	108	112	82.0
640 × 480	60	3	81.2	227	123	150
640 × 480	60	7	118	157	160	116
640 × 480	60	11	154	119	196	94.1

Table 2. Performance with typical image sizes, for raw and consistency- and smoothness-filtered results. D is the number of disparities. N is the neighbourhood size.

The hardware setup is a Pentium 4, 3.1 GHz machine with an NVIDIA GeForce FX 5900 graphics card, at AGP4X speed, running Linux. The performance is shown in Table 2.

4.4 CPU Usage

Our implementation is written such that all computation can be performed on the graphics card. This should mean that the CPU is free to perform other processing while the GPU is computing. We find that this is not entirely the case with the NVIDIA driver. Rather, there appear to be several synchronization points in our algorithm, at which the driver polls the GPU in a tight loop. This results in high CPU usage. However, if we break our algorithm into several pieces, each ending in synchronization points, the CPU is free to perform other work while waiting for the GPU. Doing this, we find that the CPU usage is approximately 10 percent. Since each piece of the algorithm is fast, we need to use the CPU frequently but in short bursts, which is problematic when using user-level threading.

5 Conclusions

We present a fast implementation of a well-known stereo disparity algorithm on graphics hardware. We focus on filtering our results to decrease the error, while maintaining performance. We explore the performance of three filters using a density-error analysis, which allows us to easily compare the performance of the filters against each other and against best-, worst-, and random-case filters.

Our implementation demonstrates that graphics co-processors are powerful and flexible enough to implement significant portions of early vision computation. The programmability of graphics hardware coupled with the expected rapid increase in its power suggests that graphics hardware will play an increasing role in early vision computation.

References

1. Scharstein, D., Szeliski, R.: A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. In: Proc. IEEE Workshop on Stereo and Multi-Baseline Vision. (2001) 1–35
2. Kanade, T., Kano, H., Kimura, S., Yoshida, A., Oda, K.: Development of a video-rate stereo machine. In: Proc. International Robotics and Systems Conference. (1995) 95–100
3. Kanade, T., Yoshida, A., Oda, K., Kano, H., Tanaka, M.: A stereo machine for video-rate dense depth mapping and its new applications. In: IEEE Conf. on Computer Vision and Pattern Recognition. (1996) 196–202
4. Woodfill, J., Herzen, B.V.: Real-time stereo vision on the PARTS reconfigurable computer. In Pocek, K.L., Arnold, J., eds.: IEEE Symposium on FPGAs for Custom Computing Machines, Los Alamitos, CA, IEEE Computer Society Press (1997) 201–210
5. Woodfill, J., von Herzen, B., Zabih, R.: Frame-rate robust stereo on a PCI board (1998)

6. Yang, R., Pollefeys, M.: Multi-resolution real-time stereo on commodity graphics hardware. In: IEEE Conference on Computer Vision and Pattern Recognition. (2003) 211–217
7. Zach, C., Klaus, A., Reitinger, B., Karner, K.: Optimized stereo reconstruction using 3d graphics hardware. Technical Report TR/2003/024, VRVis: Zentrum für Virtual Reality und Visualisierung Forschungs (2003)
8. Okutomi, M., Kanade, T.: A locally adaptive window for signal matching. International Journal of Computer Vision **3** (1989) 209–236
9. Arnold, R.D.: Automated stereo perception. Technical Report AIM-351, Artificial Intelligence Lab, Stanford University (1983)
10. Fua, P.: A parallel stereo algorithm that produces dense depth maps and preserves image features. Machine Vision and Applications **6** (1993) 35–49
11. Marr, D., Poggio, T.: Cooperative computation of stereo disparity. Science **194** (1976) 283–287
12. Murray, D., Little, J.: Using real-time stereo vision for mobile robot navigation. Autonomous Robots **8** (2000) 161–171