Fast Level Set Multi-View Stereo on Graphics Hardware

Patrick Labatut[1]
[1] Département d'Informatique
École normale supérieure
45, rue d'Ulm, 75005 Paris, France
patrick.labatut@ens.fr

Renaud Keriven[1,2]
Jean-Philippe Pons[2]
[2] CERTIS
École Nationale des Ponts et Chaussées
6-8 avenue Blaise Pascal, 77455 Marne-la-Vallée, France
{keriven,pons}@certis.enpc.fr

Abstract

In this paper, we show the importance and feasibility of much faster multi-view stereo reconstruction algorithms relying almost exclusively on graphics hardware. Reconstruction algorithms have been steadily improving in the last few years and several state-of-the-art methods are nowadays reaching a very impressive level of quality. However all these modern techniques share a very lengthy computational time that completely forbids their more widespread use in practical setups: the typical running time of such algorithms range from one to several hours. One possible solution to this problem seems to lie in the use of graphics hardware: more and more computer vision techniques are taking advantage of the availability of cheap computational horsepower and divert graphics hardware from its original purpose to accelerate the early stages of some algorithms. We present here an almost full implementation on graphics hardware of a multi-view stereo algorithm based on surface deformation by a PDE: this algorithm tries to minimize the error between input images and predicted ones by reprojection via the surface. As it mainly works on whole images, it is well suited for graphics hardware. We show how we succeeded to bring the whole reconstruction time within minutes. Results for synthetic and real data sets are presented with computational times and compared with those of other state-of-the-art algorithms.

1. Introduction

Three-dimensional shape reconstruction from a set of pictures is one of the oldest problems in computer vision and find its roots back in robotics. However, even if in a few years the quality of shape reconstruction has greatly improved, it is nowadays far from real-time: the current best algorithms for reconstruction from multiple views are typically very slow, taking one to several hours to run and thus hindering a more common use of such a tool.

A quite recent idea to improve the running time of many computer vision algorithms consists in using commodity graphics cards not for rendering fancy graphics but for general-purpose computations. Allowing portions of algorithms to run on GPU is not always possible and requires specific knowledge but when possible, the performance gain is often worth the efforts. We show how this approach was successful for us, allowing quality shape reconstruction within minutes.

The first section of this paper gives some background about GPUs and modern multi-view algorithms. The following section presents the shape reconstruction algorithm we used, the next section describes its implementation on graphics hardware and the last section shows some results with the corresponding running times on different data sets and comparison with other current methods on the exact same data sets.

2. Background and Previous Work

2.1. Multi-View Stereo Algorithms

Given n (≥ 2) images of the same scene (and the intrinsic/extrinsic parameters of the corresponding cameras), the goal is to build a 3D model of the scene as close as possible to the original. This goal is difficult to reach because occluded parts and lighting can substantially change the appearance of a scene from different viewpoints.

Currently state-of-the-art multi-view stereo algorithms can be very roughly divided into several classes: first, discrete methods à la *space carving* [16] which work on an initially whole discrete volume and compute a cost function to extract a surface from this volume [24, 25] (interestingly, the ancestor of *space carving*, *voxel coloring* [23] has a very low running time but can not be considered as a general and modern shape reconstruction algorithm). Then we find several variational methods [13, 5, 14, 20] based on the deformation of a surface under a PDE: these method are inspired from the initial work of [7]. And finally, there exists

methods computing and fusing together depth maps [15], and other methods that enforce visual hull hard constraints [24, 8] using graph-cuts to optimize the consistency of the recovered shape with the input images.

2.2. General-Purpose Computation on GPUs

In just a few years graphics cards have become heavily parallel processing machines with increased programming capabilities making their use possible for other purposes than standard real-time rendering [19]. A simplistic way to understand what a GPU actually does is to consider it as a stream processor [2] which executes a computational kernel over all the elements of an input stream (possibly accessing other streams) and puts the corresponding results into an output stream.

However we will need a more precise idea of the way a GPU accelerates the rendering process (see Fig. 1) in order to present our method: the graphical primitives (triangles) that constitute the objects to be rendered are downloaded to the graphic card as a list of vertices with several attributes (position, color, texture coordinates, ...). Then, these vertices and their attributes undergo (programmable) transformations before being projected to the screen (a 2D grid). The interior of the triangles defined by three consecutive projected vertices is then iterated over the screen and at every pixel location, a fragment is generated: it combines the coordinates of the current pixel and the attributes interpolated from the vertices attributes. This fragment undergoes a series of (programmable) transformations and if in particular its depth attribute allows it (i.e. is lower than the value stored in the depth buffer), the current pixel is colored with the fragment color (and the depth buffer is updated accordingly).

Using a graphics card as parallel computer is thus quite simple: a large quad it drawn to cover the whole screen, ensuring its depth is correct and a program that will execute on every generated fragment is loaded. Then the result is read back from video memory. If a loop is needed, e.g. for an iterative algorithm, lighter render-to-texture techniques can be used instead: the framebuffer is replaced by a texture that is first rendered to and which can later be read as an input texture by fragment programs.

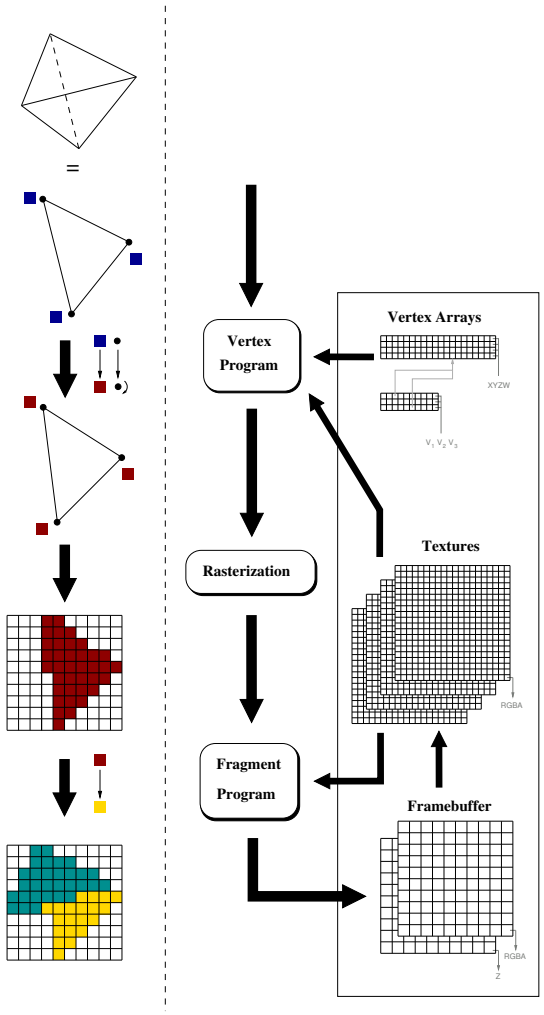


Figure 1. The graphics pipeline: how modern graphics cards accelerate 3D rendering from object triangles to framebuffer pixels

2.3. Computer Vision Using Graphics Hardware

Computer vision algorithms are nowadays often GPU-accelerated, as they work on the same kind of data as rendering. Recovering the disparity map of two images has already been thoroughly studied: from simple block matching strategy with a multi-scale approach [26], to mixed CPU/GPU approach initializing a graph-cut optimization with crude depth maps computed on GPU [9], and parallel dynamic programming on GPU [11]. Numerical schemes for 2D level sets have also been implemented by brute force [21] and more recently, 3D level sets for segmentation [17] introduced a GPU to CPU message passing system.

In the case of multi-view algorithm, the GPU is becoming a tool of choice, enabling hardware-accelerated rasterization and re-projection, but often only as a powerful preprocessing tool for the very first stages of the algorithm: for instance, [16] is one of the early promoter of graphics hardware, using it to project the input images back into a voxel volume. More recent methods, such as [24], even when not using GPUs suggest improvements to reduce running

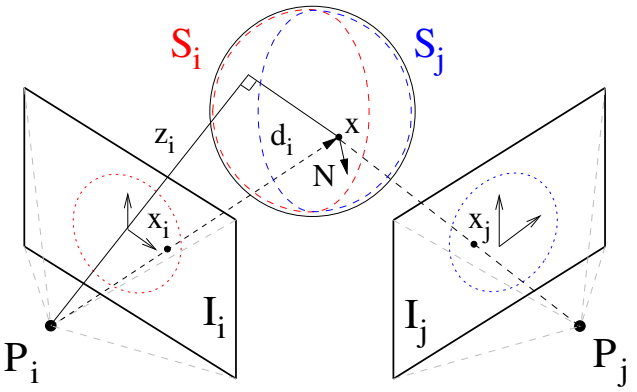


Figure 2. Notations

time using graphics hardware. However, to our knowledge, no multi-view stereo algorithm for full shape reconstruction has been thoroughly adapted to run on GPU.

What we present in this paper is an almost full GPU implementation of a modern reconstruction method which was actually designed with GPU in mind.

3. Shape Reconstruction Algorithm

The variational method of [20] borrows from [7] but reformulates the problem of shape reconstruction as an image registration problem. It is thus simpler, more robust than most other methods and more suitable for a GPU implementation. It is also worth noting that in opposition to most other algorithms, it does not rely on background-segmented input images (but can use these silhouettes if they are available).

3.1. Notations

A surface $S \subset \mathbb{R}^3$ models the shape of the scene being reconstructed. We note $P_i : \mathbb{P}^3 \to \mathbb{P}^2$ the perspective projection of the i^{th} camera and $I_i : \Omega_i \subset \mathbb{R}^2 \to \mathbb{R}^c$ the corresponding images with c channels (as usual, we assume that the camera can be described by the pinhole camera model and that distortions in the images have already been corrected). As the method we use take into account the visibility of the surface points, S_i will refer to the part of the surface S visible in the image I_i. Then $P_{i,S}^{-1} : P_i(S) \to S_i$ is the re-projection from the camera P_i onto the surface S. Finally $I_j \circ P_j \circ P_{i,S}^{-1} : P_i(S_j) \to \mathbb{R}^c$ is the re-projection of the image I_j in the camera P_i via the surface S.

3.2. Energy to Minimize

The energy we wish minimize is mainly a matching term i.e. a sum of dissimilarity terms between oriented pairs of images: each pair is composed of one of the input images and some predicted image obtained by re-projecting another input image into the camera corresponding to the former image. This leads to the following energy (M is the dissimilarity measure between two areas of image):

$$\begin{aligned}\mathcal{M}(S) &= \sum_i \sum_{j \neq i} \mathcal{M}(I_i, I_j)(S) \\ &= \sum_i \sum_{j \neq i} M|_{\Omega_i \cap P_i(S_j)} \left(I_i, I_j \circ P_j \circ P_{i,S}^{-1}\right)\end{aligned}$$

One must notice that this approach does not mix regularization and data fidelity together and is hence making the tuning of the regularization much easier. A user defined smoothing term is simply added. Computing the variation of the matching w.r.t. to an infinitesimal vector displacement of the surface δS, we get ($D\cdot$ is the Jacobian matrix of a function):

$$\left.\frac{\partial \mathcal{M}(S + \epsilon \delta S)}{\partial \epsilon}\right|_{\epsilon=0} = \int_{\Omega_i \cap P_i(S_j)} \underbrace{\partial_2 M(\mathbf{x}_i)}_{1\times c} \underbrace{DI_j(\mathbf{x}_j)}_{c\times 2} \underbrace{DP_j(\mathbf{x})}_{2\times 3} \underbrace{\left.\frac{\partial P_{i,S+\epsilon\delta S}^{-1}(\mathbf{x}_i)}{\partial \epsilon}\right|_{\epsilon=0}}_{3\times 1} d\mathbf{x}_i$$

As the surface moves the predicted images change accordingly. The variation of the matching term thus involves the derivative of the similarity measure w.r.t. its second argument which we will note $\partial_2 M$. After some additional steps, the previous expression leads to the equation giving the evolution of the surface S along its outward normal $\mathbf{N}$:

$$\frac{\partial S}{\partial t} = \left[-\lambda H + \sum_i \sum_{j \neq i} \delta_{S_i \cap S_j}\, \partial_2 M\, DI_j\, DP_j\, \frac{\mathbf{d}_i}{z_i^3}\right] \mathbf{N}$$

where H is the mean curvature of S (which corresponds to a smoothing term added to the energy), $\delta_\cdot$ is the Kronecker symbol, $\mathbf{d}_i$ the vector from the camera P_i to the considered point, z_i its depth and λ a smoothing coefficient ($\lambda > 0$).

3.3. Similarity Measure

The described method allows the use of whatever similarity measure we want: cross-correlation, correlation ratio, mutual information, etc... [12]. We limit ourselves to

the *local normalized cross-correlation* $\mathrm{cc}(I_i, I_j)$, which assumes a local affine relation between the intensities of the two image areas: however experiments have shown that it can even cope with non-lambertian surfaces provided the window size is small enough (albeit making the estimation less robust to noise and outliers). We also choose to use smooth Gaussian correlation windows instead of hard ones. We detail here the expression of M and $\partial_2 M$ as they are needed to later explain the GPU implementation:

$$\begin{aligned}
\mu(I_i) &= \frac{G_\sigma * I_i}{\omega} \\
\mathrm{v}(I_i) &= \frac{G_\sigma * I_i^2}{\omega} - \mu^2(I_i) + \tau^2 \\
\mathrm{v}(I_i, I_j) &= \frac{G_\sigma * I_i I_j}{\omega} - \mu(I_i)\,\mu(I_j) \\
\mathrm{cc}(I_i, I_j) &= \frac{\mathrm{v}(I_i, I_j)}{\sqrt{\mathrm{v}(I_i)\,\mathrm{v}(I_j)}}
\end{aligned}$$

where $\omega(\mathbf{x_0}) = \int_\Omega G_\sigma(\mathbf{x_0} - \mathbf{x})\, d\mathbf{x}$ is the spatial normalization to account for the shape of the correlation window, and G_σ is a Gaussian kernel.

The dissimilarity measure $M^{cc}(I_i, I_j)$ between images I_i and I_j is simply the sum of the normalized cross-correlation over the whole domain:

$$M^{cc}(I_i, I_j) = -\int_\Omega \mathrm{cc}(I_i, I_j)(\mathbf{x})\, d\mathbf{x}$$

Its partial derivative $\partial_2 M(I_i, I_j)$ required for the computation of the minimizing flow given above is defined as the function which for any image variation δI verifies:

$$\left.\frac{\partial M(I_i, I_j + \epsilon\delta I)}{\partial \epsilon}\right|_{\epsilon=0} = \int_\Omega \partial_2 M(I_i, I_j)(\mathbf{x})\delta I(\mathbf{x})\, d\mathbf{x}$$

In the case of the normalized cross-correlation, we get:

$$\partial_2 M^{\mathrm{cc}}(I_i, I_j) = \alpha(I_i, I_j)\, I_i + \beta(I_i, I_j)\, I_j + \gamma(I_i, I_j)$$

where:

$$\begin{aligned}
\alpha(I_i, I_j) &= G_\sigma * \frac{-1}{\omega\sqrt{\mathrm{v}(I_i)\,\mathrm{v}(I_j)}} \\
\beta(I_i, I_j) &= G_\sigma * \frac{\mathrm{cc}(I_i, I_j)}{\omega\,\mathrm{v}(I_j)} \\
\gamma(I_i, I_j) &= G_\sigma * \left(\frac{\mu(I_i)}{\omega\sqrt{\mathrm{v}(I_i)\,\mathrm{v}(I_j)}} - \frac{\mu(I_j)\,\mathrm{cc}(I_i, I_j)}{\omega\,\mathrm{v}(I_j)}\right)
\end{aligned} \quad (1)$$

3.4. Energy Minimization

The minimization of the energy by gradient descent is implemented within the *level set* framework (introduced by [4] and developed by [18]) which gives numerical stability and automatic handling of surface topological changes. However this comes at a cost and to reduce the computational burden, the *narrow band* algorithm [1] is used to evolve the level sets.

As the energy is optimized through a simple steepest gradient descent, it can easily get stuck in a local minimum. The algorithm therefore adopts a multi-scale approach by using the result of the optimization at a coarser scale to initialize the optimization at a finer level.

4. Graphics Hardware Implementation

Whereas other variational methods for multi-view stereo are CPU-only, [20] was designed with classical GPU acceleration in mind. We extend this approach further by using GPGPU techniques.

4.1. Analysis

We evaluated the bottlenecks of a the original partially GPU-accelerated implementation of [20] in order to convert it almost completely to GPU.

The main loop driving the evolution of the surface and executed at each time step can be decomposed as shown in Tab. 1. As all the surface points visible in image I_i should be points from the narrow band, the M^{cc} derivative is computed over the common domain of image I_i and image I_j re-projection, allowing for stream computation. Items 6 and 7 actually spend most of the time doing bilinear interpolations in either the input images pixels or the similarity measure derivative pixels. The depth computations and re-projections were already running on GPU. Finally the level sets computations cover only a fraction of the running time. We thus chose to concentrate our efforts on items 5.2, 6 and 7.

4.2. Re-projection and Visibility Masks

The depth computation is simply done by rendering the surface and updating the *depth buffer*. The visibility masks $\Omega_i \cap P_i(S_j)$ and the image re-projections are computed with the *shadow mapping* technique which consists in using the contents of the *depth buffer* we got from the P_j camera as a texture and rendering the surface in the camera P_i. Accessing texture elements in this special texture triggers a comparison between the depth of the current fragment and the depth stored in the texture and returns a boolean value. The

0 %	1 mesh update
10 %	2 distance function update
5 %	3 mesh download to the GPU
5 %	4 depth computation in every camera 5 similarity measure derivative update for each camera couple (i, j)
0 %	5.1 re-projection of the image I_j in the camera P_i
20 %	5.2 computation of the similarity measure derivative
20 %	6 computation of band points attributes for each band point / for each camera position / visibility / intensity computation
40 %	7 normal speed computation for each band point / for each camera pair if the point is visible in the two cameras 7.1 corresponding normal speed computation
0 %	8 CFL condition
0 %	9 level sets update

Table 1. Main loop with typical running time distribution

surface points are used as texture coordinates and the texture matrix (which is applied to the texture coordinates before accessing *texels*) is replaced by the P_j camera matrix. We can hence generate a depth mask using the P_j camera *depth buffer* as a texture. Then the I_j image re-projection is obtained by applying this image as a texture (see Fig. 3).

4.3. Similarity Measure Derivative

The smoothing Gaussian windows were at first implemented with a fast recursive filter [3]. IIR filters do not fit very well in the GPU computational model constraints so it was replaced by an equivalent simple separable convolution much more suited to the SIMD-CREW hardware of the fragment unit. In order to mask away some of the pixels in the texture, we take advantage of the efficient *Z-Culling* technique, well-known to help resolving static branching: a mask is loaded in the *depth buffer* that allows masked fragments to completely skip the execution of the fragment program thus saving time. Using this masking technique, the computation of α, β and γ from (1) easily maps to successive fragment programs:

- put 1, I_i, I_j, I_i^2, $I_i\, I_j$, I_j^2 into textures
- convolve the previous textures with G_σ
- compute ω, $\mu(I_i)$, $\mu(I_i)$, $\mathrm{v}(I_i)$, $\mathrm{v}(I_j)$, $\mathrm{v}(I_i, I_j)$, $\mathrm{cc}(I_i, I_j)$
- convolve the previous pass results with G_σ and combine the results to get α, β and γ.

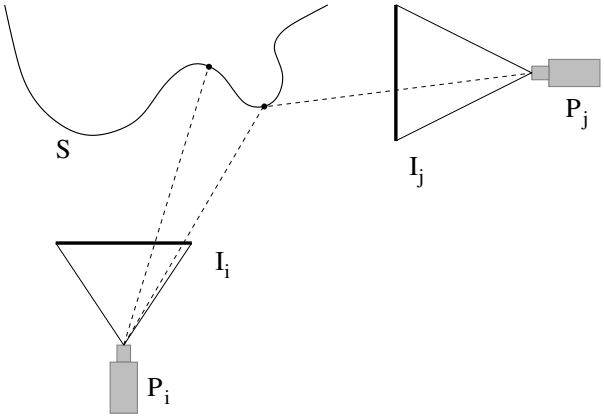


Figure 3. The shadow mapping technique enables fast computation of visibility masks using previously computed depth maps

4.4. Points Attributes

At the finest scale, the band typically contains many dozens of thousand of points. An input texture containing the coordinates of the band points is first created (as shown in Fig. 4). Fragment programs are iteratively executed over the cameras on all the band points, to compute the position, visibility and intensity attributes from this input texture, and output corresponding attribute textures. *Z-Culling* is once again used to mask away some parts of the input stream where no computation needs to be done.

For the normal speed we combine the visibility textures and the masks to generate a mask for *Z-Culling*. We then iterate over the camera pairs while accumulating the normal speeds computed for each points in the narrow band. The normal speed texture just computed is then read back into system RAM and used to finally update the level sets.

5. Results and Comparison with Other Methods

The OpenGL API and its extensions (mostly render-to-texture and fragment program extensions) were used to program the graphics hardware. Prototyping was done thanks to the Cg programming language. All the presented results (Fig. 5, Fig. 6 and Fig. 7) were obtained on a PC with an Intel Xeon 2.8 GHz CPU and 1 GB of system RAM equipped with an NVIDIA GeForce 7800 GTX graphics card with 256 MB of video RAM.

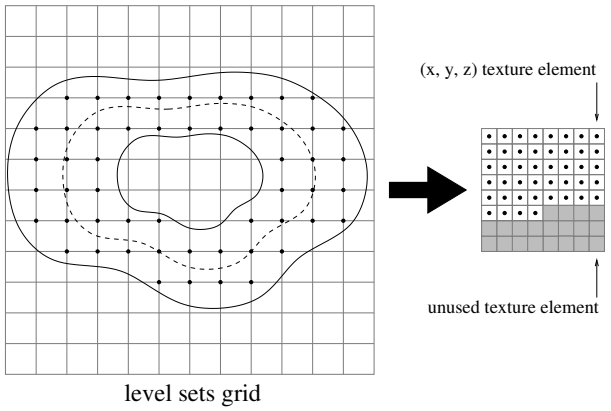


Figure 4. The narrow band is straightforwardly mapped to a 2D texture of 3D coordinates

#Images	25
Resolution	$256 \times 256 \times 3$
#Pairs	50
Level set volume	$128 \times 128 \times 128$
Running time (CPU/GPU)	$\sim 780\,\mathrm{s}$
Running time (GPU)	$\sim 210\,\mathrm{s}$

Figure 5. "buddha" data set (25 views): first row: some input images, second row: ground truth, third row: result

The "buddha" data set comes from the Intel OpenLF Mapping project and the "dino" and "temple" data sets come from a recent multi-view stereo algorithms evaluation [22].

These results are the same as those we got with the original implementation from [20].

The overall speed factor is almost about 4 when compared to the already partially GPU-accelerated implementation from [20]. . However the sections of the algorithms that were heavily using bilinear interpolations observe an eleven-fold improvement in general, and the computation of the measure derivative gets a ninefold decrease of its running time.

A recent survey on multi-view stereo algorithms [22] compared several state-of-the-art reconstruction techniques (see Tab. 2[1]) and mainly focused on quality of shape reconstruction using several metrics.

The respective running times of these methods were also reported. Given the diversity of hardware, all these running times are actually very difficult to compare (especially between CPUs and GPUs), however we will use exactly the same "normalization" as the authors: the times were corrected taking into account the frequency of the CPU (and for GPU-based algorithms, using the typical frequency of machine using such GPU). If clearly not flawless, this correction is supposed to give at least an idea of the relative performance of the different algorithms: even after this correction, our almost full GPU-accelerated method is accordingly the fastest method by far among those whose running times were reported. As we can see in Tab. 2, it is the only algorithm which took a few minutes to complete a full reconstruction. Most others are taking several hours long (of course, with the notable exception of [20] from which ours is derived).

[1] excerpt from the 2006-04-03 results.

6. Conclusion and Future Work

We based our work on the algorithm from [20]. We have evaluated the original mixed GPU/CPU-implementation to focus only on its bottlenecks. We reformulated the corresponding parts of the algorithms to run efficiently on graphics hardware. We get a significant decrease of the total running time which now allows shape reconstruction within minutes and makes our shape reconstruction the fastest among its class to our knowledge. If we were to compare this result with the original CPU-only method [7] from which [20] borrowed, we would get an even more im-

Algorithm	Hardware	Time	
		Reported	Corrected
Furukawa [8]	3.0 GHz Pentium 4	6:00	6:00
Goesele [10]	3.4 GHz Pentium 4	12:24	14:03
Hernandez [6]	3.0 GHz Pentium 4	1:46	1:46
Pons [20]	2.8 GHz Xeon 430 MHz GeForce 7800	0:11	0:11
Pons/Labatut	430 MHz GeForce 7800	0:03	0:03
Vogiatzis [25]	2.3 GHz Pentium 4	0:52	0:40

Table 2. Some of the results from a recent review of multi-view stereo reconstruction algorithms [22] ("dino" data set, 16 views): times were corrected in minutes at 3 GHz to allow easier comparison

pressive picture: the hypothetical overall performance gain would be about 200.

Currently, some parts of the algorithm are still not running on GPU: the level set related back and forth conversion to mesh and the update of the level sets within the narrow-band. The whole level set framework could actually be completely replaced by a simple dynamically refinable mesh deformation more suitable for GPU. Of course, this would come at the expense of easy topological changes handling but the whole algorithm would become even more appropriate for graphics hardware. Moreover one can argue that such a change is currently questionable given the current programming constraints in graphics hardware, but the steady increase in programmability of graphics hardware let us hope that the next generation of graphics hardware may be able to handle more smoothly such problems.

References

[1] D. Adalsteinsson and J. A. Sethian. A Fast Level Set Method for Propagating Interfaces. *Journal of Computational Physics*, 118(2):269–277, 1995.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics*, 23(3):777–786, Aug. 2004.

[3] R. Deriche. Fast Algorithms for Low-Level Vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1(12):78–88, Jan. 1990.

[4] A. Dervieux and F. Thomasset. A Finite Element Method for the Simulation of Rayleigh-Taylor Instability. *Lecture Notes in Mathematics*, 771:145–159, 1979.

[5] Y. Duan, L. Yang, H. Qin, and D. Samaras. Shape Reconstruction from 3D and 2D Data Using PDE-based Deformable Surfaces. In *European Conference on Computer Vision*, volume 3, pages 238–251, 2004.

[6] C. H. Esteban and F. Schmitt. Silhouette and Stereo Fusion for 3D Object Modeling. *Computer Vision and Image Understanding, special issue on 'Model-based and image-based 3D Scene Representation for Interactive Visualization'*, 96(3):367–392, Dec. 2004.

[7] O. Faugeras and R. Keriven. Complete Dense Stereovision using Level Set Methods. In *European Conference on Computer Vision*, volume 1406, pages 379–393, 1998.

[8] Y. Furukawa and J. Ponce. Carved Visual Hulls for Image-based Modeling. In *European Conference on Computer Vision*, May 2006.

[9] I. Geys, T. P. Koninckx, and L. J. V. Gool. Fast Interpolated Cameras by Combining a GPU-based Plane Sweep with a Max-Flow Regularisation Algorithm. In *International Symposium on 3D Data Processing, Visualization and Transmission*, pages 534–541, 2004.

[10] M. Goesele, S. M. Seitz, and B. Curless. Multi-View Stereo Revisited. In *Computer Vision and Pattern Recognition*, June 2006.

[11] M. Gong and Y.-H. Yang. Near Real-Time Reliable Stereo Matching Using Programmable Graphics Hardware. In *IEEE International Conference on Computer Vision and Pattern Recognition*, 2005.

[12] G. Hermosillo, C. Chefd'hotel, and O. Faugeras. Variational Methods for Multimodal Image Matching. *International Journal of Computer Vision*, 50(3):329–343, 2002.

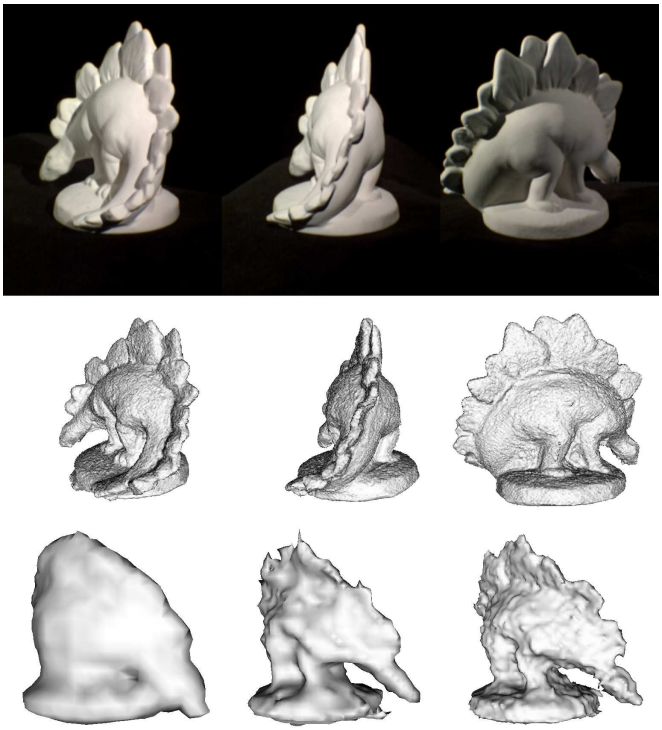
[13] H. Jin, S. Soatto, and A. J. Yezzi. Multi-View Stereo beyond Lambert. In *Computer Vision and Pattern Recognition*, volume 1, pages 171–?, 2003.

[14] H. Jin, S. Soatto, and A. J. Yezzi. Multi-View Stereo Reconstruction of Dense Shape and Complex Apperarance. *International Journal of Computer Vision*, 63(3):175–189, 2005.

[15] V. Kolmogorov and R. Zabih. Multi-Camera Scene Reconstruction via Graph Cuts. In *European Conference on Computer Vision*, volume 2352, May 2002.

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

[17] A. E. Lefohn, J. M. Kniss, C. D. Hansen, and R. T. Whitaker. A Streaming Narrow-Band Algorithm: Interactive Deformation and Visualization of Level Sets. *IEEE Transactions on Visualization and Computer Graphics*, 10(40):422–433, July 2004.



#Images	16
Resolution	$256 \times 256 \times 3$
#Pairs	32
Level set volume	$192 \times 192 \times 192$
Running time (CPU/GPU)	$\sim 860\,\mathrm{s}$
Running time (GPU)	$\sim 240\,\mathrm{s}$

Figure 6. "dino" data set (16 views): first row: some input images, second row: result, third row: evolution (first time step of each scale but the last)

#Images	24
Resolution	$256 \times 256 \times 3$
#Pairs	48
Level set volume	$128 \times 192 \times 96$
Running time (CPU/GPU)	$\sim 1130\,\mathrm{s}$
Running time (GPU)	$\sim 320\,\mathrm{s}$

Figure 7. "temple" data set (24 views): first row: some input images, second row: result

[18] S. Osher and J. A. Sethian. Fronts Propagating with Curvature-Dependent Speed: Algorithms Based on Hamilton-Jacobi Formulations. *Journal of Computational Physics*, 79(1):12–49, 1988.

[19] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, Aug. 2005.

[20] J.-P. Pons, R. Keriven, and O. Faugeras. Modelling Dynamic Scenes by Registering Multi-View Image Sequences. *International Conference on Computer Vision and Pattern Recognition*, 2:822–827, 2005.

[21] M. Rumpf and R. Strzodka. Level Set Segmentation in Graphics Hardware. In *IEEE International Conference on Image Processing*, volume 3, pages 1103–1106, 2001.

[22] S. Seitz, B. Curless, J. Diebel, D. Scharstein, and R. Szeliski. A Comparison and Evaluation of Multi-View Stereo Reconstruction Algorithms. In *Computer Vision and Pattern Recognition*, June 2006.

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

[24] S. Sinha and M. Pollefeys. Multi-View Reconstruction using Photo-consistency and Exact Silhouette Constraints: A Maximum-Flow Formulation. In *International Conference on Computer Vision*, Oct. 2005.

[25] G. Vogiatzis, P. H. S. Torr, and R. Cipolla. Multi-View Stereo via Volumetric Graph-Cuts. In *Computer Vision and Pattern Recognition*, pages 391–398, 2005.

[26] R. Yang and M. Pollefeys. Multi-Resolution Real-Time Stereo on Commodity Graphics Hardware. In *IEEE Conference on Computer Vision and Pattern Recognition*, volume 1, pages 211–218, 2003.