Fast and efficient dense variational stereo on GPU

Julien Mairal, Renaud Keriven and Alexandre Chariot
CERTIS
ENPC
77455 Marne-la-Vallee cedex 2 France
julien.mairal@m4x.org, keriven@certis.enpc.fr, chariot@certis.enpc.fr

Abstract

Thanks to their high performance and programmability, the latest graphics cards can now be used for scientific purpose. They are indeed very efficient parallel Single Instruction Multiple Data (SIMD) machines. This new trend is called General Purpose computation on Graphics Processing Unit (GPGPU [4]). Regarding the stereo problem, variational methods based on deformable models provide dense, smooth and accurate results. Nevertheless, they prove to be slower than usual disparity-based approaches. In this paper, we present a dense stereo algorithm, handling occlusions, using three cameras as inputs and entirely implemented on a Graphics Processing Unit (GPU). Experimental speedups prove that our approach is efficient and perfectly adapted to the GPU, leading to nearly video frame rate reconstruction.

1. Introduction

For the last ten years graphics cards have grown significantly in terms of performance, functionalities and relative importance in a computer.

Compared to Central Processing Units (CPUs), the growth rate has been far more higher and the GPUs do not obviously follow Moore's law. The newest NVIDIA/ATI cards confirm this trend in terms of pipelines numbers (24/48), memory amounts (512MB), clock frequencies (650MHz), memory bandwidth (50GB/sec), etc.

The idea of using them for something else than three-dimensional rendering emerged when GPU designers made them programmable, in order to provide more expressiveness to software developers who wanted to design their own effects.

Nevertheless, being provided with a "programmable" GPU does not mean having any generic parallel machine. A graphics card is mainly designed to render a scene and display it. Therefore, very tight constraints remain. In a simplified view, data structures in video memory are either a set of vertices or a texture. A vertex is a set of coordinates in the three-dimensional space, augmented with attributes, such as texture coordinates or colors. A texture is an array of pixel color values, each usually composed of four channels when using the RGBA mode. Moreover, because of the programming paradigm, namely the Concurrent Read Exclusive Write Single Instruction Multiple Data (CREW SIMD) programming model, only specific algorithms can be adapted and efficiently implemented on a GPU.

In this paper, we present an implementation of a dense stereo algorithm which runs entirely on a GPU. It is based on the variational framework of deformable models proposed by the pioneering work of [5]. In this work, the authors consider the n-view stereovision problem and recover the entire surface of an object, minimizing some energy that incorporates both photometric consistency and regularizing constraints. Here, for speed reasons, we restrict ourselves to the case of two or three cameras, and model the surface as a depth map from one reference camera. Although providing smooth and accurate results for three-dimensional shape reconstruction, such approaches are usually neglected in everyday stereo-vision applications because of their relative lack of efficiency. This is what motivated our work. Indeed, most of the efforts that have been made toward efficient stereo, are disparity-based. Instead of recovering three-dimensional information directly, the process is a two stages one: first, estimating the correspondences (disparity) between two rectified images; second reconstructing the three-dimensional object. In this context, the authors of [6, 12, 7, 15, 11, 14, 13] have developed real-time or near real-time stereo algorithms on GPUs. For example, the algorithm designed by Yang and Pollefeys in [13] can reach up 289 million disparity evaluations per second on a (now old!) ATI Radeon 9800.

Exactly like the state of the art methods classified on the reference "Middlebury Stereo Vision Page" [10], this impressive result is not focused toward accurate three-dimensional shape reconstruction. Our goal here is differ-

ent: it consists in obtaining as fast as possible a coherent and accurate surface, from two or three images or video streams. A first attempt to use GPUs for the same problem has been made by the authors of [16]. Yet, although they present visibly good results, they use a non-robust difference of gray levels as a photometric consistency criterion where we use instead normalized cross-correlation. Above all, the main difference between our work and theirs is that we use a mathematically founded gradient descent where they let their surface move at constant speed, just looking for a decrease of energy.

Finally, note that, although correlation based, we do neither require any image rectification nor need the images be oriented the same way: the images are back-projected on the surface and correlated directly on it. To improve convergence and cope with local minima, our algorithm is multi-scale. Camera selection and occlusion are taken into account through considerations based on the normal to the surface.

2. GPU programming

2.1. Overview of GPU architecture

We describe here in a simplified manner the graphics "pipeline". For more details, we refer the reader to [8, 1, 4, 2]. Figure 1 presents a view of a graphics card:

1. Vertices are sent to the card with directives indicating their topology (full or wireframe polygons, triangles or quads, etc.).

2. The vertex units, which are Multiple Instructions Multiple Data (MIMD) processors on the latest cards, compute, for each vertex, a position on the screen (or on an off-screen buffer), as well as some new attributes (see [8]). This first step is now programmable. Vertex programs are also called vertex shaders.

3. Some operations are then performed, such as culling (discarding vertices which are out of a previously defined bounding box) or stencil testing (discarding vertices which are on a previously defined area).

4. The so-called rasterizer converts then the vertices and their topology into sets of pixels on the screen. It links the output registers of the vertex units to the input registers of the pixel units so that each pixel receives interpolated values from the three vertices of the triangle it belongs to.

5. The pixel units, which are Single Instruction Multiple Data (SIMD) processors compute the color of each pixel. This step is programmable, pixels programs being also called fragment shaders, and is usually the one considered for GPGPU, although the increasing programmability of vertex shaders now turns then into good GPGPU candidates in some cases.

6. The graphics card memory is mainly organized into textures that can be seen as two-dimensional arrays. Textures are used to communicate between the CPU and the GPU or between two different successive computations (called rendering for obvious reasons). Pixels programs have a full random read access to textures and exclusive write access to one location in one[1] texture (the previously mentioned CREW model). Vertex programs can access textures too, but this access is very slow.

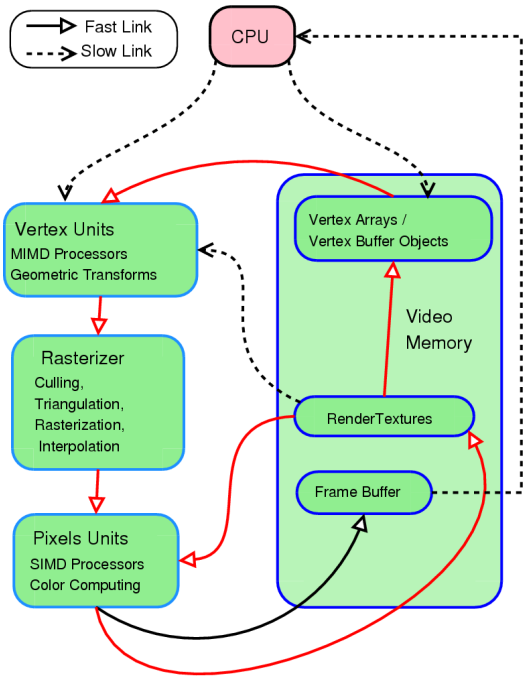

Figure 1. A simplified view of a recent graphics pipeline

2.2. Iterative mesh deformation

To take advantage of the efficiency of GPUs, it is essential to avoid as much as possible data transfers between the GPU and the CPU: they are very slow compared to the internal GPU video memory access. In the context of an iterative process like the one we will have to deal with, there are two GPU/CPU transfer issues:

[1] Actually, up to four textures using so-called Multiple Rendering Target (MRT)

1. An iteration should be able to process some input data into output data available for the next iteration without transmitting output back to the CPU. When using the pixels units only, data inputs and output are stored into so-called *Render Textures* (or more recently *Frame Buffer Objects* (FBO)), allowing the card to render into an off-screen buffer, that can be linked afterward to an input texture register. This is the usually adopted solution. When using also the vertex units, data inputs must be stored into *Vertex Buffer Objects* (VBO), that are actually a set of vertices data into the video memory. It is possible to transfer data from pixel textures to a VBO. This is the solution we adopted: our surface is handled through a so-called *Vertex Array*, vertices attributes being stored in a VBO.

2. One needs a way to test some stopping criterion, despite the fact that the pixels do not communicate, without transmitting data back to the CPU. This is hopefully possible thanks to a mechanism called *occlusion query*, which counts the number of rendered pixels. It is then easy to have every single pixel artificially not rendered is some test is true, and being instantaneously warned if this test is true for all the pixels.

3. Two cameras

3.1. Model

Let us consider the case where two cameras are used. Dealing with a third camera is similar and will be detailed further. Considering two fully-calibrated cameras and choosing camera 1 as a reference, we model the object as a regular triangulated deformable surface S (see figure 2) where each vertex M lie on a fixed ray issued from the optical center O of camera 1. Although asymmetrical, this representation spares a lot of computation. Let d_M be the distance between point O and point M. We will design some energy $E(S)$ that is actually a function of the d_Ms and minimize it by means of a gradient descent.

3.2. Energy

The simplest energy could be the sum of the differences between the gray levels:

$$E(S) = \int_S (I_2 \circ \Pi_2(m) - I_1 \circ \Pi_1(m))^2 dS(m)$$

where the I_i are the images, Π_i the projections associated to the cameras and $\circ$ denotes the function composition. Because the sum is done all over the surface, one could think this measure is robust enough, like mentioned in [16]. We have tried it without success on real images. Following [5],

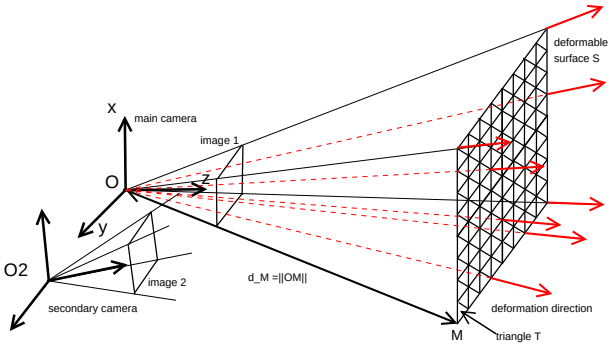

Figure 2. Framework of the cameras.

we use a more robust energy, based on normalized cross-correlation, where the images are back-projected onto the surface (or the plane tangent to S around m) and correlated on some neighborhood of m:

$$E(S) = \int_S (1 - \rho(I_1 \circ \Pi_1, I_2 \circ \Pi_2, m)) dS(m) \qquad (1)$$

We choose a discrete version of $E(S)$, computing the correlations on each triangle only. Denoting by T the triangles of S, we thus take:

$$E(S) = \sum_T E_T = \sum_T (1 - \rho_T(I_1 \circ \Pi_1, I_2 \circ \Pi_2)) \qquad (2)$$

with (omitting dependencies in T in the notations):

$$\begin{aligned} \rho_T &= \frac{< I_1, I_2 >}{|I_1|.|I_2|} \qquad (3) \\ < I_1, I_2 > &= \frac{1}{A_T} \int_T (I_1 \circ \Pi_1(m) - \overline{I_1 \circ \Pi_1}) \\ &\qquad (I_2 \circ \Pi_2(m) - \overline{I_2 \circ \Pi_2}) dm \\ \overline{I_i \circ \Pi_i} &= \frac{1}{A_T} \int_T I_i \circ \Pi_i(m) dm \\ |I_i| &= \sqrt{< I_i, I_i >} \end{aligned}$$

A_T denoting the surface of triangle T. Note that we do not use any surface metric anymore. Multiplying the correlation by the area of the triangle would lead to the smallest possible surface (i.e. converging toward point O). This behavior is common to all active contour based methods and is usually solved with some balloon force. We could have done this here but we got better results with above formulation added with some regularization term (see further).

3.3. Gradient

We minimize the energy by means of a gradient descent, using a multi-resolution scheme, in terms of both mesh and

image sizes, in order to cope with local minima as much as possible. $E(S)$ being actually a function of the distances d_Ms, we have to compute its gradient with respect to these distances. Let $V(M)$ be the set of triangles to which a vertex M belongs, we write:

$$\frac{\partial E(S)}{\partial d_M} = \sum_{T \in V(M)} \frac{\partial E_T}{\partial d_M} \tag{4}$$

Computing these quantities from equations (2) and (3) when M is one of the vertices of T is straightforward but gives rather long expressions we will not detail here. We refer the reader to [9] for their complete writing. To discuss their GPU implementation, the important fact is that they boil down to double sums (the mean values being encapsulated inside the main sum) depending on the following quantities:

$$\frac{\partial I_i \circ \Pi_i(m)}{\partial d_M} \tag{5}$$

where point m is a point of T and M one of its vertices.

3.4. GPU discretization

Implementing the above quantities on a GPU, the continuous sums $\int_T F(m)dm$ involved in equation (2) for some F will indeed be replaced by discrete sums $\sum_p F(p)dm(p)$ where p is determined by the rasterizer and each pixel shader will compute $f(p)$. Let T be the triangle (M_1, M_2, M_3), p will be some barycenter $p = \alpha_1 M_1 + \alpha_2 M_2 + \alpha_3 M_3$ with $\alpha_i \geq 0$ and $\alpha_1 + \alpha_2 + \alpha_3 = 1$. We then need to compute the quantities of equation (5), i.e.

$$D_{i,k}(\alpha_1, \alpha_2, \alpha_3) = \frac{\partial I_i \circ \Pi_i(p)}{\partial d_{M_k}}$$

for $k = 1, 2, 3$ and every $(\alpha_1, \alpha_2, \alpha_3)$ chosen by the rasterizer. A direct computation involving camera projections gives:

$$D_{i,k}(\alpha_1, \alpha_2, \alpha_3) = g_i(\alpha_k f_i(M_k), \Pi_i(p), (\nabla I_i) \circ \Pi_i(p))$$

where f_i and g_i are simple geometric function given in [9]. Note that, when a pixel shader is called for a point p, the value α_k is not known. Yet, it is possible to get the required $\alpha_k f_i(M_k)$: adding a virtual attribute to the vertices and setting it to $f_i(M_k)$ for M_k and to 0 for $M_j (j \neq k)$, automatically provides the interpolated quantity $\alpha_k f_i(M_k)$ when processing p. Moreover, the rasterizer can compute $\Pi_i(p)$ because $\Pi_i(p) = \alpha_1 \Pi_i(M_1) + \alpha_2 \Pi_i(M_2) + \alpha_3 \Pi_i(M_3)$. An advantage of this is that the quantities depending on the vertices M_k are computed once only by the vertex units and not for each point p.

Note also that, when rendering a given triangle $T = (M_1, M_2, M_3)$, the three values $D_{i,1}$, $D_{i,2}$ and $D_{i,3}$ can be computed simultaneously for each p thanks to vectorial capacities of the GPU. Now, choosing to render the surface from camera 1 point of view, we get that, for a given $p \in T$, the value of $I_1 \circ \Pi_1(p)$ does not depend on the positions of the vertices of T, i.e:

$$D_{1,k}(\alpha_1, \alpha_2, \alpha_3) = 0$$

3.5. Summations

Each $F(p)$ being computed for some F by pixel shaders for each $p \in T$, we still need to recover $\sum_{p \in T} F(p)dm(p)$. Such a *reduction* is a classical problem on SIMD machines and logarithmic complexity algorithms are usually designed. Here, our triangles have a small number of pixels thanks to our multi-scale approach adapting mesh size to image dimensions. Thus a simple pass where each pixel shader deals with one triangle and performs a loop[2] over it points is much more efficient.

We also have to sum, for each vertex, quantities over its related triangles (equation (2)). Again, vertices are now assigned to pixel shaders that perform a loop over their respective triangles.

3.6. Regularization

As we mentioned previously, we left any surface metric apart and should add manually a regularization term to the energy. Mean curvature motion could be used here (or in an equivalent manner adding the area of the surface to the energy). Actually, we got better results directly smoothing d_M, adding the following term to the gradient:

$$K\Big((\frac{1}{|N(M)|} \sum_{M' \in N(M)} d_{M'}) - d_M\Big)$$

where $N(M)$ is the set of the neighbors of M and K some constant adapted to the considered level of detail.

3.7. Stopping criterion

An obvious stopping term based on the maximum value of the gradient gives good results. In fact, a fixed number of iterations for each level of detail gives faster convergence, yet keeping similar results.

3.8. Complete scheme

As depicted figure 3, one complete iteration is finally:

1. Assign the surface vertices to the GPU vertices and render, computing $D_{2,k}(p), (k = 1, 2, 3)$ for every point p.

[2] available on recent GPUs

2. Assign the surface triangles to pixel shaders and perform the double summation yielding $\frac{\partial E_T}{\partial d_M}$.

3. Assign the surface vertices to pixel shaders and perform the neighborhood summation leading to the gradient $\frac{\partial E(S)}{\partial d_M}$. Update the vertices position d_M according to this gradient.

4. Update the VBO containing the vertices positions and optionnaly compute a stopping criterion.

3.9. Results

Our experiments were done on a standard 3GHz PC with a (now outdated!) *NVIDIA Geforce 7800 GTX 256* graphics card. The OpenGL library and the Cg language were used. I_2, I_1 and ∇I_2 at the different resolutions were preprocessed in a first step. The textures were in a 16 bits mode (yielding 40% faster programs). We obtained a fast convergence with an average 10 iterations per level of detail. Our multi-resolution approach proved to prevent from converging toward a local minimum.

As a reference, we developed a CPU version minimizing the same energy than the GPU version. This version was cautiously written and compiled with the latest compilers with all of the optimizations turned on. Table 1 shows the speedups between CPU and GPU versions. For each level of detail, the image and mesh resolutions are given. We observed a mean speedup of 10 times to 15 times. This shows that our implementation makes the most of the graphics pipeline.

Some results and total running times are given on figures 7, 8, 9 and 10.

For each presented result, we ran about 8 iterations for each level of detail mentioned table 1 except for the two last level of detail that are run respectively 4 and 2 times. A total running time of about 250ms is observed for global convergence. As expected, our algorithm is not as fast as those dealing with disparity maps. Again, we insist that our targeted applications are different.

A drawback so far is that we do not handle occlusions. We could easily prevent the algorithm taking occluded triangles into account. We will do it show in the next section when using a third camera.

4. Three cameras - Occlusions

Our model can easily be extended to three cameras. Let us denote them C, R and L, respectively a "center", a "right" and a "left" cameras. C will be the reference camera (the role played previously by camera 1) and the correlation will be computed between cameras C and R or between cameras C and L. Note that the cameras do not have to be

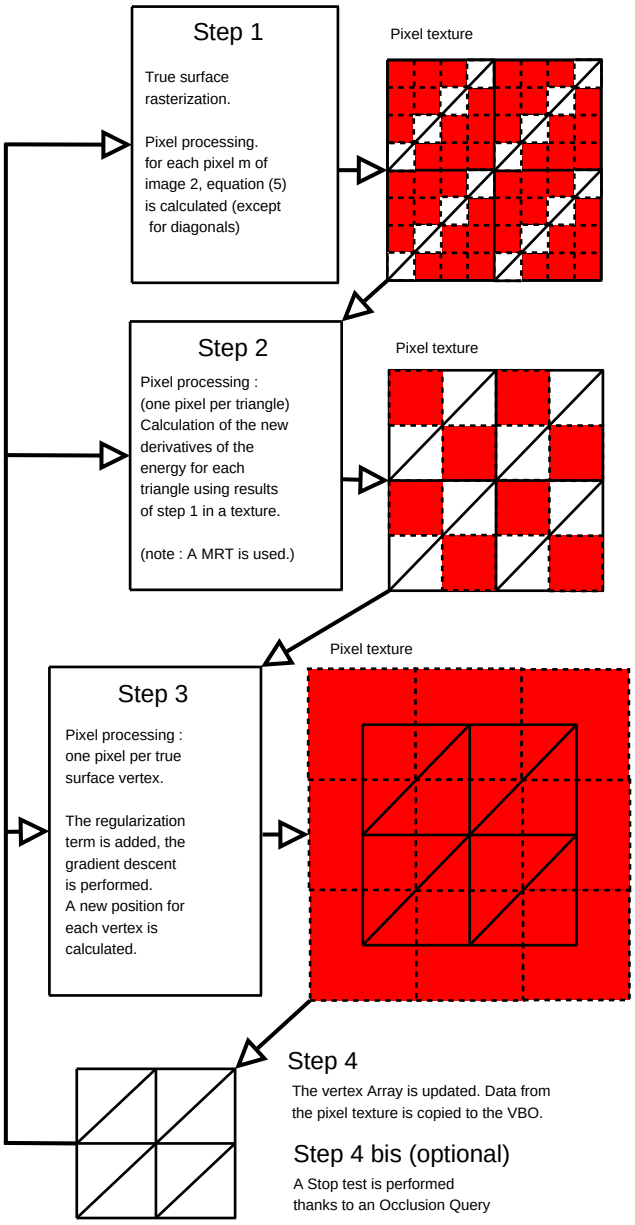

Figure 3. Example of one iteration for a mesh of 3x3 vertices and an image of 8x8 pixels. Actually, our implementation begins with a 3x3 mesh and a 32x32 image. On this figure, a red quad in dotted line represents one rendered pixel. The plain lines represent the mesh.

Image	Mesh	GPU	CPU	Speedup
64^2	5^2	1.60 kHz	555 Hz	2.9
128^2	9^2	1.33 kHz	116 Hz	11.5
256^2	17^2	464 Hz	28.6 Hz	16.2
512^2	33^2	102 Hz	7.5 Hz	14.1
512^2	65^2	89.4 Hz	7.3 Hz	12.2
512^2	129^2	67.9 Hz	7.2 Hz	9.0

Table 1. Iterations per seconds for each level of detail.

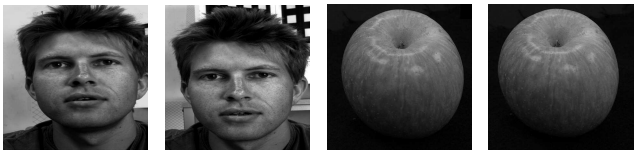

Figure 4. First and second data sets.

Figure 5. Third and fourth data sets (courtesy of Pr. Kyros Kutulakos (University of Toronto)).

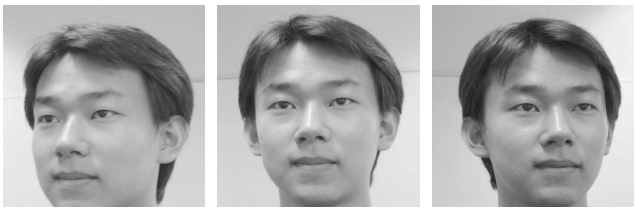

Figure 6. A data set for three cameras from [3].

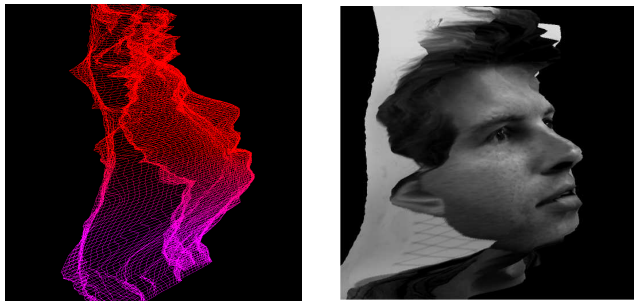

Figure 7. 240ms, two cameras.

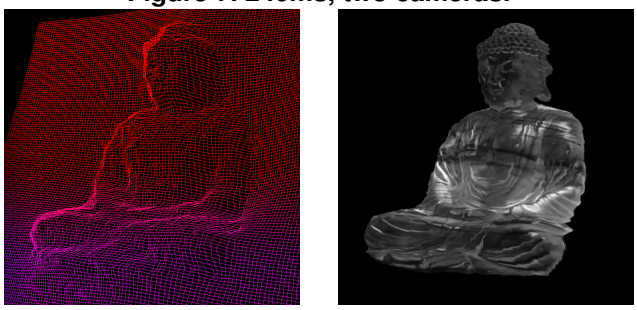

Figure 8. 250ms, two cameras.

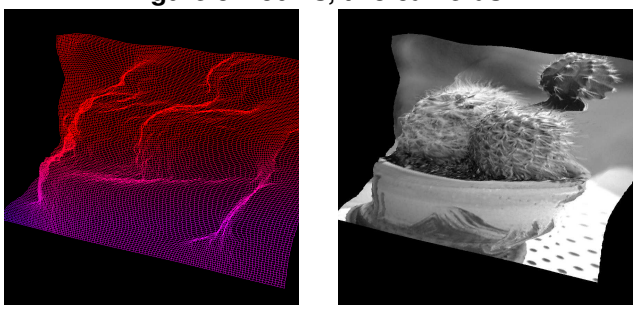

Figure 9. 230ms, two cameras.

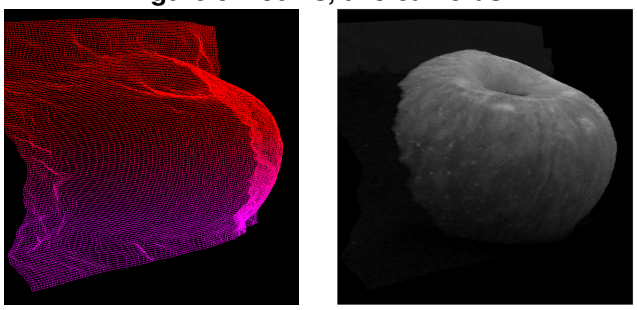

Figure 10. 220ms, two cameras.

disposed in a (left, center, right) way. The only criteria to assign the cameras are that: (i) the surface is modeled from the optical center of C, and (ii) correlation between L and R is not taken into account.

This choice being done, we divide at each iteration the triangles into two sets S_R and S_L, S_R (respectively S_L) being the set of the triangles that will be correlated between camera C and camera R (respectively camera L). The associated energy is indeed similar to the one given equation (2):

$$E(S) = \sum_{T \in S_R} (1 - \rho_T(I_C \circ \Pi_C, I_R \circ \Pi_R)) + \sum_{T \in S_L} (1 - \rho_T(I_C \circ \Pi_C, I_L \circ \Pi_L))$$

In our tests, we simply use the normal to the triangles to choose the best secondary camera, R or L. Moreover, occlusions can easily be taken into account. Rendering the surface from secondary cameras viewpoints, we determine whether a triangle is seen or not from them. When a triangle is not seen by any secondary camera, we just assign it neither to S_R nor to S_L. Then, keeping the energy given by equation (6), a triangle that is not seen by at least C and another camera does not contribute to the energy. Actually, we ignore also triangles that are not enough "in front of" the cameras in a certain sense. This yields a first way to handle discontinuities, an important point that our model does not take into account so far.

We have implemented this algorithm on both CPU and GPU. The CPU version does not take occlusions into account (a CPU visibility test would be too slow) and runs only 9% slower than its two cameras version.

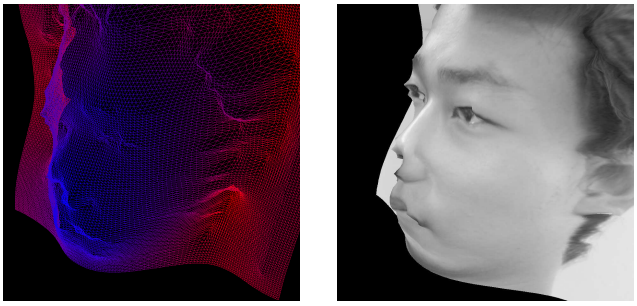

Figure 11. Two cameras with occluded areas

To implement the GPU version, we use a so-called *Stencil Buffer* to classify the triangles into two groups, being then able to work on one group only at a time. This functionality consist in discarding the rendering of a pixel p when the value of the stencil buffer at p is below, equal to or greater than some reference value. The solution is thus to memorize in the stencil buffer whether a pixel (or a triangle) belongs to S_R, to S_L or to none of them. This test is hardware implemented and is therefore very fast. An overview of the algorithm for the GPU version is given figure 13. Note that rendering pixels in two passes (first S_R and then S_L) could lead to memory cache inefficiency if the pixels were randomly distributed between S_R and S_L. Hopefully, this is not the case here! Our first results show a 30% overhead between the two and three cameras GPU versions. We are still in the process of designing a more optimized three cameras GPU version.

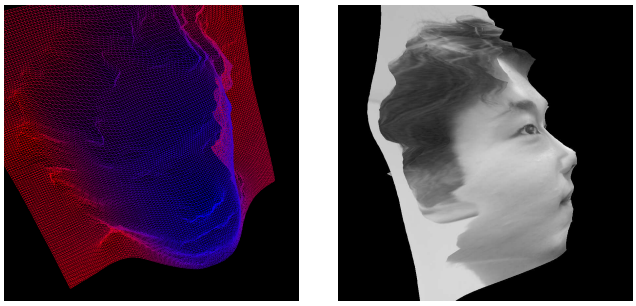

Figure 12. Three cameras.

A first result is presented figures 11 and 12. Thanks to occlusion handling, but also thanks to more precision where the three cameras are available, the three cameras reconstruction (fig. 12) is more correct and more accurate than the two cameras one (fig. 11).

5. Videos

We have tested our algorithm on video sequences. Convergence on the first frame takes about $250ms$. Yet, taking advantage of temporal continuity, we just take the surface recovered at time t as an initial value for convergence at time $t + 1$. Some preliminary tests, essentially consisting in adjusting choices of levels of detail and number of iterations, showed promising results at a rate of 8 frames per second for the two cameras algorithm. With the recently available graphics cards a 12 frames/sec rate should already be possible. Moreover, it should be noted that our approach could easily be turned into a parallel version using two graphics cards with a minimum of communication at each iteration, yielding full video rate for a very affordable price.

6. Conclusion

In this paper, we have presented a fast dense stereo algorithm based on variational principles. Handling occlusions, it takes two or three cameras as inputs and is entirely implemented on Graphics Processing Units. Experiments show that it is efficient and well adapted to the GPU: speedup of about 10 to 15 times are coherent with the graphics card we

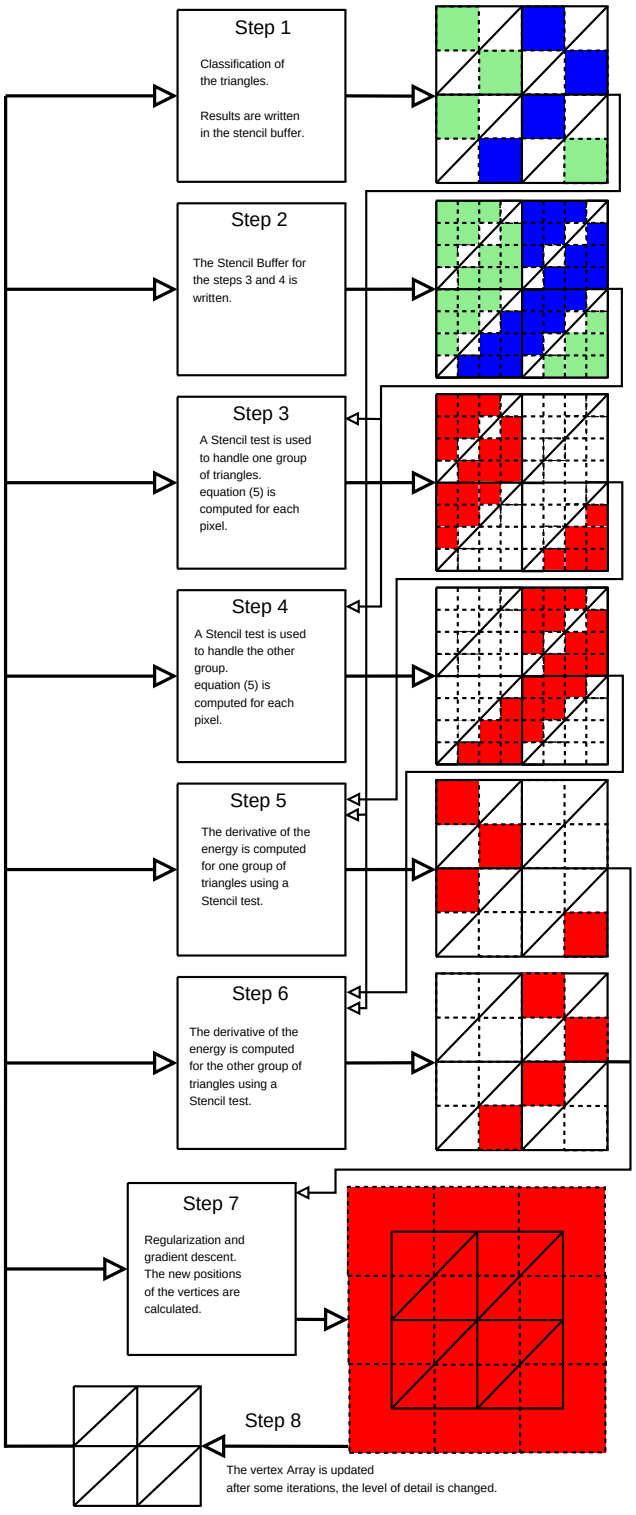

Figure 13. The GPU implementation with three cameras.

used. The reconstructed surface is accurate. This fully justifies considering this approach instead of usual disparity-based algorithms for certain applications. Taking advantage of temporal continuity, we achieved reconstruction at a video rate of about 8 frames/sec. Future work includes taking discontinuities into account, dealing with more than one GPU, and investigating the reconstruction of a complete object several such systems.

References

[1] Opengl specifications.

[2] `http://developer.nvidia.com`.

[3] `http://www.cs.ust.hk/~quan/WebPami/pami.html`.

[4] `http://www.gpgpu.org`.

[5] O. D. Faugeras and R. Keriven. Variational principles, surface evolution, pdes, level set methods, and the stereo problem. *IEEE Transactions on Image Processing*, 7(3):336–344, 1998.

[6] I. Geys, T. P. Koninckx, and L. V. Gool. Fast interpolated cameras by combining a gpu based plane sweep with a max-flow regularisation algorithm. In *3DPVT '04: Proceedings of the 3D Data Processing, Visualization, and Transmission, 2nd International Symposium on (3DPVT'04)*, pages 534–541, Washington, DC, USA, 2004. IEEE Computer Society.

[7] M. Gong and Y.-H. Yang. Near real-time reliable stereo matching using programmable graphics hardware. In *CVPR '05: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1*, pages 924–931, Washington, DC, USA, 2005. IEEE Computer Society.

[8] E. Kilgariff and R. Fernando. *The GeForce 6 Series GPU Architecture*, volume GPU Gems 2, chapter Chap. 30. 2005.

[9] J. Mairal and R. Keriven. A gpu implementation of variational stereo. Technical Report O5-13, CERTIS, November 2005.

[10] D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *Int. J. Comput. Vision*, 47(1-3):7–42, 2002.

[11] J. Woetzel and R. Koch. Multi-camera real-time depth estimation with discontinuity handling on pc graphics hardware, August 2004.

[12] J. Woetzel and R. Koch. Real-time multi-stereo depth estimation on gpu with approximative discontinuity handling, March 2004.

[13] R. Yang and M. Pollefeys. A versatile stereo implementation on commodity graphics hardware. *Real-Time Imaging*, 11(1):7–18, 2005.

[14] R. Yang, M. Pollefeys, H. Yang, and G. Welch. A unified approach to real-time, multi-resolution, multi-baseline 2d view synthesis and 3d depth estimation using commodity graphics hardware.

[15] C. Zach, K. Karner, and H. Bischof. Hierarchical disparity estimation with programmable 3d hardware, 2004.

[16] C. Zach, A. Klaus, M. Hadwiger, and K. Karner. Accurate dense stereo reconstruction using graphics hardware, 2003.