Accelerated Epipolar Geometry Computation For 3D Reconstruction Using Projective Texturing

Rui Rodrigues[*] António Ramires Fernandes

Universidade do Minho, Portugal

Abstract

The process of *3D reconstruction*, or depth estimation, is a complex one, and many methods often have several parameters that may require fine tunning to adapt to the scene and improve reconstruction results. Usability of these methods is directly related to their response time. Epipolar geometry, a fundamental tool used in 3D reconstruction, is commonly computed on the CPU. We propose to take advantage of the advances of graphic cards, to accelerate this process. Projective texturing will be used to transfer a significant part of the computational load from the CPU into the GPU. The new approach will be illustrated in the context of a previously published work for 3D point reconstruction from a set of static images. Test results show that gains of up to two orders of magnitude in terms of computation times can be achieved, when comparing current CPU's and GPU's. We conclude that this leads to an increase in usability of 3D reconstruction methods.

Keywords: Epipolar Geometry, Projective Texturing, Depth Estimation, 3D Reconstruction, OpenGL

1 Introduction

The reconstruction of view-independent 3D information from a set of static images has taken the attention of researchers for some years [Faugeras 1993; Hartley and Zisserman 2000]. The approaches are multiple and with different applications. Many techniques are based on establishing correspondences between images [Redert et al. 1999]. Common corresponded entities can be points [Kawamoto and Imiya 2001; Tell and Carlsson 2000; Rodrigues et al. 2002], lines or contours [Sato and Cipolla 1999; Schulz-Mirbach and Weiss 1994], rectangular blocks or segments [Smith 1998]. If a pair of entities from two distinct images correspond, and the camera position, orientation and intrinsic parameters for two images are known (or estimated), it is possible to compute the position of those entities in 3D space. This position can be expressed directly in a 3D referential or as a *depth* – the distance relative to a reference camera. However, each of these entity types has some type of ambiguity associated such that, given two entities in different images, it is not straightforward to determine whether they correspond or not [Redert et al. 1999].

Determining correspondences between a large number of entities (possibly thousands), in a large number of images (tens), is a time consuming process. Furthermore, the basic theory does not anticipate the problems one encounters when dealing with real scenes, such as noise in the images, approximate camera calibration and perspective distortions, occlusions and highlights. Methods dealing with contours, for instance [Sato and Cipolla 1999; Schulz-Mirbach and Weiss 1994; Rodrigues et al. 2002], may encounter additional problems with contour extraction.

In addition, for a given technique, the heterogeneity of inputs requires the tuning of a series of parameters (either automatically or interactively) to improve the reconstruction results. Many issues then arise: is the image set sufficient or appropriate for the reconstruction of all the details of the scene? What are are the best parameter settings for a particular scene?

Getting user (or automatic) feedback to deal with these issues calls for *interactive* reconstruction rates. The longer the time a method takes to provide a reconstruction the less interactive it becomes. Hence, usability of these methods is directly related to response time. Achieving real-time 3D reconstruction using off-the-shelf systems would give way to a wide range of applications, namely in terms of image and scene transmission.

Given the amount of geometry processing involved in reconstruction tasks, we propose to use a well known 3D graphics API - OpenGL - as the platform to per-

[*]This work was funded by FCT (the Portuguese Science and Technology Foundation) under Grant Praxis XXI/BD/20322/99. The author also wishes to thank the support of Philips Research Laboratories (Eindhoven, The Netherlands)

form (batch) geometric computations. The advantages are twofold: we will be using a language that (1) is naturally oriented to 3D geometry processing, and (2) serves as a seamless link to hardware acceleration, given the wide API support provided by graphic cards manufacturers, and the ever-growing capabilities of graphic cards.

GPU's have a tremendous computational power, and recently graphic cards allow the programmability of the GPU. Even without resorting to programming the GPU itself, there is a clear performance advantage of using the GPU over the CPU for graphical tasks.

The research community has initiated efforts on the exploitation of the capabilities of GPU's for heavy-load computational tasks other than graphics rendering. Among possible applications, image-based reconstruction and modelling is growing in interest. There are recent results on the hardware-oriented implementation of known view-independent reconstruction algorithms, such as visual hulls[Li et al. 2003], block-based stereo disparity estimation [Zach et al. 2004] and voxel carving [Sainz et al. 2002]. There are also works on view-dependent algorithms, namely by [Lok 2001] and [Yang et al. 2002].

This paper presents a method that accelerates a class of view-independent approaches based on point reconstruction. We propose to explore the potential of the GPU, using projective texturing [Segal et al. 1992], to accelerate some of the tasks relating to 3D reconstruction of points using epipolar geometry. The use of projective texturing in the context of 3D reconstruction has been explored previously[Lok 2001; Li et al. 2003; Sainz et al. 2002], but not in the context of epipolar geometry computation.

Using projective texturing for epipolar geometry related computations, a significant amount of the computational load is transferred from the CPU to the GPU. Tests performed with an implementation based on the point reconstruction method presented in [Rodrigues et al. 2002] show the very significant performance gain obtained – *up to two orders of magnitude* – and the potential of the method proposed in this paper.

Section 2 describes the fundamentals of epipolar geometry and how it is used for 3D or depth reconstruction. The application of projective texturing in the context of epipolar geometry is described in section 3. Tests and results of our proposal using a case study are presented in section 4. Finally conclusions and directions for future work are presented in section 5.

2 Epipolar Geometry for 3D Point Reconstruction

Typical reconstruction algorithms rely on finding correspondences between entities in two images. In this article we focus on the correspondence of points. If a point $\mathbf{c}_{ip}$ in image $\mathbf{I}_i$ is corresponded with a point $\mathbf{c}_{jq}$ in $\mathbf{I}_j$, and camera positions $\mathbf{C}_i$ and $\mathbf{C}_j$ are known, then the corresponding *reconstructed point* (*r-point*) $\mathbf{r}_{ip}$ is (in the ideal case) easily determined by triangulation using the information of the cameras.

However, finding correspondences between entities of different images is not trivial. On the one hand, errors in the process of image acquisition and camera calibration may lead to distortions of the entities, resulting in missed correspondences (false negatives). It may also happen that ambiguities in the entities corresponded lead to incorrect correspondences (false positives). Whenever possible, it is useful to add constraints to help reducing this *correspondence problem*.

Epipolar geometry allows the search for correspondences of a $\mathbf{c}_{ip}$ to be constrained to a subset of the entities in image $\mathbf{I}_j$. In this section we will present the basics of epipolar geometry in the context of our work. A more detailed analysis of epipolar geometry can be found in [Hartley and Zisserman 2000].

2.1 Epipolar Geometry for Two-Image Point Reconstruction

Consider two images $\mathbf{I}_i$ and $\mathbf{I}_j$, with known cameras $\mathbf{C}_i$ and $\mathbf{C}_j$, and a point $\mathbf{c}_{ip}$ of $\mathbf{I}_i$ (see Figure 1). Let $\mathbf{V}_{\mathbf{c}_{ip}}$ be the view ray of $\mathbf{c}_{ip}$ associated to $\mathbf{C}_i$ (the half-line starting in the centre of $\mathbf{C}_i$ and going through $\mathbf{c}_{ip}$). The epipolar line $\mathbf{E}_{\mathbf{c}_{ip},j}$ is the projection of $\mathbf{V}_{\mathbf{c}_{ip}}$ in $\mathbf{I}_j$. Points $\mathbf{v}_{ip_x}$ belonging to $\mathbf{V}_{\mathbf{c}_{ip}}$ are projected in $\mathbf{E}_{\mathbf{c}_{ip},j}$, and points $\mathbf{p}_{\mathbf{v}_{ip_y},j}$ of $\mathbf{E}_{\mathbf{c}_{ip},j}$ can be projected back to $\mathbf{V}_{\mathbf{c}_{ip}}$, using $\mathbf{C}_j$.

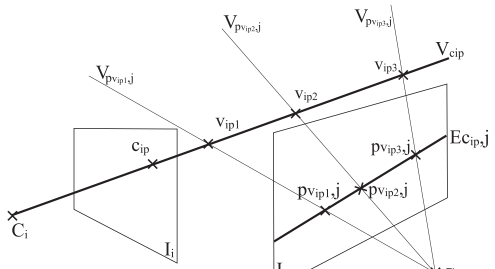


Figure 1: Epipolar Geometry

In terms of correspondence search, this means that the correct correspondence for $\mathbf{c}_{ip}$ in $\mathbf{I}_j$ has to be in $\mathbf{E}_{\mathbf{c}_{ip},j}$.

Assume there is an error function $\epsilon_{\mathbf{v}_{ip_x},j}$ that quantifies for a point $\mathbf{p}_{\mathbf{v}_{ip_x},j}$ the error of corresponding it with $\mathbf{c}_{ip}$ (a possible error function is discussed in section **??**). In this case, determining the best correspondence of $\mathbf{c}_{ip}$ could amount to minimize that error function over a set of candidates $\mathbf{p}_{\mathbf{v}_{ip_x},j}$. In the two-image setting, this could be done by evaluating the error function at the points belonging to the epipolar line $\mathbf{E}_{\mathbf{c}_{ip},j}$, as shown in Algorithm 1.

```
· Let $\mathbf{V}_{\mathbf{c}_{ip}}$ be the view ray of $\mathbf{c}_{ip}$ associated to $\mathbf{C}_i$
    · Project $\mathbf{V}_{\mathbf{c}_{ip}}$ in $\mathbf{I}_j$ as the epipolar line $\mathbf{E}_{\mathbf{c}_{ip},j}$
    · choose a set of candidate image points $\mathbf{p}_{\mathbf{v}_{ip_1},j}$, $\mathbf{p}_{\mathbf{v}_{ip_2},j}$, ... from $\mathbf{E}_{\mathbf{c}_{ip},j}$
    · Store $\epsilon_{\mathbf{v}_{ip_x},j}$ as the error of $\mathbf{p}_{\mathbf{v}_{ip_x},j}$ in $\mathbf{I}_j$
· The correspondence $\mathbf{p}_{\mathbf{v}_{ip_c},j}$ for $\mathbf{c}_{ip}$ is selected amongst the candidates with lower $\epsilon_{\mathbf{v}_{ip_x},j}$
· The reconstructed point (r-point) $\mathbf{r}_{ip}$ corresponding to $\mathbf{c}_{ip}$ is in the intersection of the view rays $\mathbf{V}_{\mathbf{c}_{ip}}$ and $\mathbf{V}_{\mathbf{p}_{\mathbf{v}_{ip_c},j}}$
```

Algorithm 1: Reconstruction using epipolar line candidates

2.2 Epipolar Geometry for Multiple-Image Point Reconstruction

For simple entities such as points, the information provided by two images may be insufficient to determine correspondences reliably. It is possible to reduce ambiguity by extending point correspondence to multiple images. In this case, choosing correspondence candidates from epipolar lines in different images gives rise to the problem of *fusion*: if $\mathbf{c}_{ip}$ corresponds with $\mathbf{c}_{jq}$ in $\mathbf{I}_j$ and with $\mathbf{c}_{kr}$ in $\mathbf{I}_k$, the associated r-points $\mathbf{r}_{ip,j}$ and $\mathbf{r}_{ip,k}$ will in general not coincide, due to multiple error sources. These two points would have to be *fused* together in an additional step.

A possible solution for the fusion problem, as suggested by [Rodrigues et al. 2002], is to choose candidates $\mathbf{v}_{ip_x}$ directly from the view ray, project each of them in the images, and accumulate the corresponding error to $\mathbf{v}_{ip_x}$ (see Algorithm 2).

```
· Let $\mathbf{V}_{\mathbf{c}_{ip}}$ be the view ray of $\mathbf{c}_{ip}$ associated to $\mathbf{C}_i$
· Choose a set of candidate world points $\mathbf{v}_{ip_1}$, $\mathbf{v}_{ip_2}$, ... from $\mathbf{V}_{\mathbf{c}_{ip}}$
· For each image $\mathbf{I}_j$ $(j \neq i)$
    · For each candidate world point $\mathbf{v}_{ip_x}$
        · Project $\mathbf{v}_{ip_x}$ in $\mathbf{I}_j$ as $\mathbf{p}_{\mathbf{v}_{ip_x},j}$
        · Let $\epsilon_{\mathbf{v}_{ip_x},j}$ be the error of $\mathbf{p}_{\mathbf{v}_{ip_x},j}$ in $\mathbf{I}_j$
        · Add $\epsilon_{\mathbf{v}_{ip_x},j}$ to the total error $\delta_{\mathbf{v}_{ip_x}}$ of $\mathbf{v}_{ip_x}$
· The r-point $\mathbf{r}_{ip}$ corresponding to $\mathbf{c}_{ip}$ is selected amongst the candidates with lower total error values $\delta_{\mathbf{v}_{ip_x}}$
```

Algorithm 2: Reconstruction using view ray candidates

This approach has an important advantage: *the correspondence, reconstruction and fusion stages are merged into one*. When the best candidate is chosen, it is already a *reconstructed point* (*r-point*) and that choice already takes into account the contribution of multiple images.

3 Projective Texturing and Epipolar Geometry

In section 2 we have seen the application of epipolar geometry on 3D point reconstruction. Epipolar geometry is traditionally computed on the CPU. We propose to perform in the GPU a significant part of the computations required, using projective texturing [Segal et al. 1992], a technique that allows an image (texture) to be projected in geometry, providing an effect similar to a slide projector.

In the context of epipolar geometry, projective texturing allows one to reverse the common approach, *ie* instead of projecting *the view ray onto an image* we project *an image onto the view ray*.

Assume that the projected image is a precomputed error map $\mathbf{\Delta}$. The part of the error map that is actually projected in the view ray corresponds to the errors associated to that ray's epipolar line.

In this work, we use distance maps as error maps, as discussed in 3.1, although the error function can be based on other criteria besides distance, such as color information. The precomputation of the error map is not a requirement though. When combined with GPU programming the error can be computed on the fly on the GPU, based on the original image set.

In this section we will show how to apply projective texturing in the context of the work in [Rodrigues et al. 2002]. For a two-camera situation, the process is described in Algorithm 3. This can be considered the GPU-oriented version of Algorithm 1.

In the case of multiple cameras, the error values from multiple error maps have to be accumulated in the same buffer, in order to minimize the number of buffer reads. Two issues have to be dealt with:

- How to accumulate errors from a significant number of cameras given the constraints on buffer attributes/ properties / pixel formats
- How to define the intrinsic and extrinsic camera parameters of the virtual camera used to render the view ray buffer, such that given a candidate selected on the buffer, the corresponding reconstructed point is efficiently computed

3.1 Distance maps as error functions

The error function chosen to assess the correspondence between points in different images is the *distance of the projected candidate to the closest contour* (as suggested in [Rodrigues et al. 2002]). This means that, for a given candidate $\mathbf{v}_{ip_x}$ of a contour point $\mathbf{c}_{ip}$ from $\mathbf{I}_i$, the individual error $\epsilon_{\mathbf{v}_{ip_x},j}$ contributed by $\mathbf{I}_j$ is the distance of

```
· Let $\mathbf{V}_{\mathbf{c}_{ip}}$ be the view ray of $\mathbf{c}_{ip}$ associated to $\mathbf{C}_i$
· Set the texture matrix for projecting the error map $\mathbf{\Delta}_j$
  relating to camera $\mathbf{C}_j$ as a texture.
· Render $\mathbf{V}_{\mathbf{c}_{ip}}$, textured with the projection of $\mathbf{\Delta}_j$, into the
  buffer
· Read the buffer from the graphics card memory into a
  main memory array
· Select the best candidate from those with the lowest errors
  in the array
· Compute the 3D reconstruction based on the candidate
  selected
```

Algorithm 3: Error projection using projective texturing

$\mathbf{p}_{\mathbf{v}_{ip_x},j}$ (the projection of $\mathbf{v}_{ip_x}$ in $\mathbf{I}_j$) to its' closest contour in $\mathbf{I}_j$.

There are several advantages in the use of distance as the correspondence error function. First, it introduces an implicit tolerance to contour displacements originated by faulty contour detection or errors in camera calibration. Second, this distance function can be easily and quickly computed in the form of a distance map using distance transforms [Borgefors 1984]. Furthermore, since it is independent of the original point $\mathbf{c}_{ip}$, the distance map can be pre-computed, and it only has to be computed once for each image, regardless of the number of points to be reconstructed, or the number of candidates per point.

In this work, the error maps that will be projected over the view rays are the distance maps generated from the contours of all images. Those maps are precomputed once and used for the reconstruction of any point.

3.2 Error Accumulation for Multiple Cameras

Buffer reading is an expensive operation, and therefore we want to minimize the number of buffer readings for each point $\mathbf{c}_{ip}$ to be reconstructed. To do this, we draw the view ray $\mathbf{V}_{\mathbf{c}_{ip}}$ multiple times in the same buffer, one for each error map $\mathbf{\Delta}_j$, $j \neq i$. On each iteration, the projected values of $\mathbf{\Delta}_j$ are accumulated in the buffer using *additive blending*.

Additive blending adds the texture values of newly rendered geometry to the values already existing in the buffer. Thus, rendering the same view ray multiple times, each time with a different error map projected from its corresponding camera, incrementally accumulates errors in the buffer.

However, since we are using textures and rendering buffers, there are limits to the maximum error value that can be represented.

In the case of occlusion of a correct candidate in a given image $\mathbf{I}_o$, its' projection could lie at a considerable distance from a contour in $\mathbf{I}_o$. This would result in a high individual error for that candidate. Since there are limits in the accumulators, as is the case in the implementation proposed, a very high error would be a problem. We propose to clamp the distance function to an upper limit. This ensures an upper limit in the accumulated error as well, while keeping one of the main advantages of the use of the distance function: limited tolerance to errors and contour displacement.

A possible solution to overcome this problem would be to take advantage of GPU programmability to perform an average of the error instead of the additive solution, and therefore individual sporadic high errors would not become a problem regarding the maximum error that can be accumulated. Notice however that this solution may prevent the correct reconstruction of contour points for which the correct correspondence is occluded in a significant number of images. These contour points would have meaningless high errors associated with the correct candidate. This is because in these images, due to occlusion, the candidate is not a contour. Therefore the error is being measured against an unrelated contour, ie. some other contour which happens to be closest to the candidate's projection (usually a contour from the occluder). Clamping the errors to a small upper limit has the advantage of limiting the influence of occlusions as discussed above.

In an RGB buffer, for instance, values are limited in the range of (0, 0, 0) to (255, 255, 255). Since each color channel is independent, accumulation can be done on a per-channel basis. To achieve this, we use color channel masking: error maps are coded as gray-scale maps, and each error map is only projected in one of the buffers' color channels (red, green or blue). The cameras' distribution between the different channels is not relevant, as long as it is guaranteed that accumulation overflow does not occur. The fact that the error values are clamped provides a reference limit for this. For instance, if error values are clamped to an upper limit of 10 units, at least 25 cameras can be safely accumulated in a single color channel ($25 \times 10 < 255$), resulting in at least 75 cameras considering the three channels. If necessary, this limitation could be relaxed with GPU programming.

3.3 Definition of the View Ray Buffer

After having a set of candidates rendered in a buffer, we have to be able to convert the 2D window position of a given candidate to the corresponding 3D point in the view ray. An appropriate setup of an orthographic camera $\mathbf{C}_O$ allows to efficiently perform this conversion.

For reconstruction purposes, only a segment of the view ray is used. The *start* and *end* points of the segment can be defined based on previous knowledge of the dimension of the scene being reconstructed. If such information is not available, a large segment can be used initially to obtain a conservative estimate of the scene's dimensions.

The intrinsic parameters of $\mathbf{C}_O$ (namely the clipping planes and the viewport) define the segment's length in 3D, *segLen3D*, and the segment's size in pixels, *segSize2D*. Each pixel corresponds to a 3D candidate of the view ray. Therefore, *segSize2D* is equivalent to the number of candidates tested. This number is directly related to the final precision of the reconstruction itself. For a given predefined *segLen3D*, the larger the set of candidates, the higher the reconstruction precision. These intrinsic parameters allow to establish a linear relation between the position of each pixel on the rendered buffer and the 3D coordinates of the candidates, defined in Equation 1.

The extrinsic parameters of $\mathbf{C}_O$ are defined as a function of the position, direction and length of the segment (Figure 2). Let ***dir*** be the normalised vector of the direction of the view ray. The *lookat* point is the middle point of the segment in 3D. The camera can be placed in any position *pos* in the plane containing the *lookat* point and perpendicular to ***dir***. The ***look*** vector of $\mathbf{C}_O$ is defined as the unit vector pointing from *pos* to the *lookat* point. The ***right*** vector of $\mathbf{C}_O$ is parallel to ***dir***. The ***up*** vector of $\mathbf{C}_O$ is set perpendicular to ***right*** and ***look***. A graphical representation of the setup of the orthographic camera $\mathbf{C}_O$ is depicted in Figure 2. Assuming the view ray is parallel to the paper plane, ***look*** and ***right*** are also parallel to the paper, and ***up*** is perpendicular to the paper.

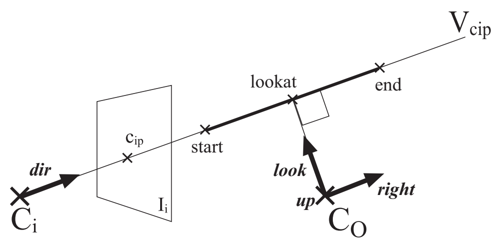


Figure 2: Orthographic camera ($\mathbf{C}_O$) for view ray segment rendering.

The operations described in Pseudo-Code 1 and Pseudo-Code 2 allow the setup in OpenGL [Shreiner 2000] of the intrinsic and extrinsic parameters of an orthographic camera as described in this section.

```
segLen3D = |end - start|;
glViewport(0,0,segSize2D,1);
glOrtho( -segLen3D/2 , segLen3D/2 , -0.5 , 0.5 , 0.5 , 2 );
```

Pseudo-Code 1: Orthographic camera's intrinsic parameters' setup in OpenGL

Having defined $\mathbf{C}_O$ as above, the segment's projection will completely fill the buffer, which will be one-pixel tall. In those conditions, a 3D point $\mathbf{r}_{ip}$ in the view ray can be easily derived from its corresponding pixel with 2D window coordinates *(x,0)* using equation (1).

```
dir = normalize(end - start);
look = dir × $\mathbf{C}_i$.up;
lookat = (start + end) / 2;
pos = lookat - look;
up = dir × look;
gluLookAt(pos, lookat, up);
```

Pseudo-Code 2: Orthographic camera's extrinsic parameters' setup in OpenGL

$$\mathbf{r}_{ip} = start + x * \left(\frac{segLen3D}{segSize2D}\right) * \textbf{\textit{dir}} \qquad (1)$$

4 Tests and Results

To assess the performance improvements achieved with projective texturing we implemented a point reconstruction algorithm based on [Rodrigues et al. 2002], with two variants: a CPU-based one, and a GPU-based (using projective texturing). These variants were submitted to two tests.

The first test serves to show the influence of varying the *number of candidates* used. The number of candidates plays a major role in the computational load of the reconstruction, but also in its quality, as referred in section 3.3. The scene's setup consists of a set of images of 512 x 512 pixels of a textured version of the Stanford model repository' bunny, taken from 70 virtual viewpoints around the bunny (Figure 3 (a)).

Scenes with high level of detail may require a large number of candidates for a proper reconstruction. For this reason, reconstruction was performed using 400, 800 and 1200 candidates, for 16826 contour points of the first image. (Figure 3 (b) illustrates the achieved final reconstruction of contour points from multiple images).

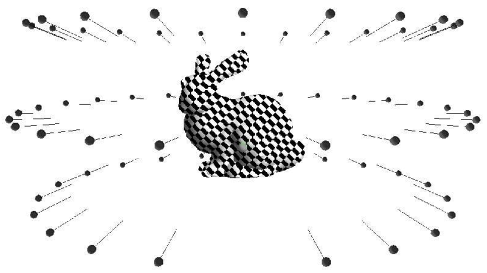
The results in Table 1 show the time in milliseconds required for reconstructing *all* the 16826 contour points evaluating 400, 800 and 1200 candidates in 69 images each, using the two implementation variants. The test was performed on System A, a Pentium IV computer running at 2.53 GHz, with a GeForce FX 5900 graphics card.

Nr. of Candidates	400	800	1200
CPU time	177171	347461	520482
GPU time	4807	5938	6574
CPU/GPU	36.85	58.51	79.17

Table 1: Performance results for reconstructing 16826 contour points, evaluating 400, 800 and 1200 candidates in 69 different images (time in milliseconds)

From the table is clear that the time required by the

(a)



(b)

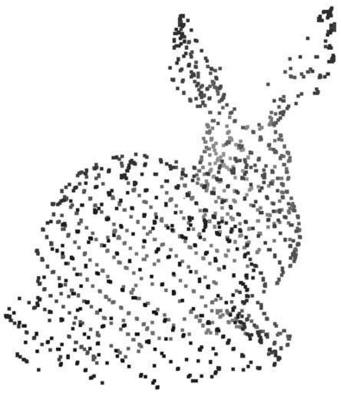


Figure 3: "Bunny": (a) The synthetic test model and the camera positions and orientations; (b) a view of multi-image point reconstruction

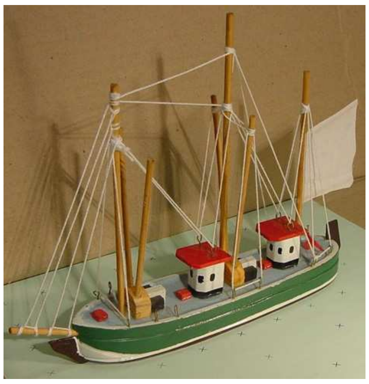
GPU implementation does not increase linearly with the number of candidates, as opposed to the CPU variant. In fact, using the largest set of candidates more than doubles the advantage of using projective texturing over the CPU implementation, when compared to the smallest set.

The second test compares the performance under different hardware settings, to show that even with older GPU's, there is still a major advantage in using projective texturing. Two hardware systems were used: *System A* is the same system used in the first test. *System B* consists of a Pentium IV Celeron computer running at 2.4 GHz, with a GeForce 256, an older graphics card.

In this test, a real scene consisting of a small wooden boat was used (see Figure 4 (a)). A set of 39 images of 1024 x 768 pixels was captured around the boat using a digital camera.

Reconstruction was performed for 21169 contour points of the first image in each of the systems, and using both implementation variants. Table 2 shows the time results in miliseconds of both implementation variants in the two hardware systems. These times include the reconstruction of all the 21169 contour points, by evaluating 1200 candidates for each point, in 38 images. Figure 4 (b) shows a virtual view of the final reconstruction using contour points from multiple images.

(a)



(b)

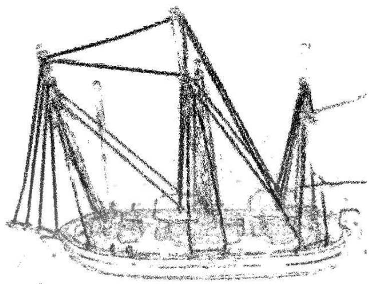


Figure 4: "Boat": (a) an image of the test scene; (b) a view of a multiple-image point reconstruction

System	A	B
CPU time	343036	377052
GPU time	6451	69874
CPU/GPU	53.17	5.39

Table 2: Performance results of the reconstruction of 21169 contour points, by evaluating 1200 candidates for each point in 38 images, in different hardware systems (time in milliseconds)

The results show that, even when using older graphics systems, significant performance gains are obtained with the GPU based implementation, although to a lesser extent.

5 Conclusions and Future Work

Improving performance of 3D reconstruction methods is of growing importance, both as a step towards real-time applications and as a mean for better exploration and interactivity of reconstruction methods. The process of 3D reconstruction is a complex one, and many methods often have several fine tuning parameters. Hence the feasibility of a method is also related to its response time. Fast response times allow the user to fine tune the parameter settings to obtain the best reconstruction results. We proposed the application of projective texturing – a technique commonly available in current hardware graphic cards and API's – to perform some crucial steps of reconstruction, namely the computation of epipolar geometry.

The technique was tested using CPU-based and GPU-based implementations (without and with projective texturing, respectively) of a modified version of the point reconstruction algorithm presented in [Rodrigues et al. 2002]. The results show gains of up to two orders of magnitudes in response time, when applying the method in current off-the-shelf CPU's and GPU's. As an example, computing the three tested reconstructions of the synthetic scene, with 400, 800 and 1200 candidates, takes less than 18 seconds using the GPU-based implementation, versus over 1000 seconds on the CPU-based implementation. The improved response time provided by the use of projective texturing enables more extensive adjustment of reconstruction settings, when comparing to the CPU-based implementation. We conclude that this leads to an increase in interactivity and usability of 3D reconstruction methods.

We now plan to explore multi-texturing and graphics hardware programming to further enhance performance, and extend the role of the GPU to other stages of reconstruction. Another possible extension to the algorithm is to compute the 3D reconstructions of multiple points simultaneously, by projecting textures in multiple view rays at a time in the same buffer and reading them in a single pass. This may seem a trivial solution for increasing performance even further, since the projective texturing would be faster for simultaneous multiple rays. However, two problems arise: how to define an appropriate buffer to record the errors of multiple view rays simultaneously and how to read back and interpret the errors. We would like to explore methods to solve these problems.

References

BORGEFORS, G. 1984. Distance transformations in arbitrary dimensions. *Computer Vision, Graphics, and Image Processing 27*, 3, 321–345.

FAUGERAS, O. 1993. *Three-Dimensional Computer Vision: A Geometric Viewpoint.* MIT Press, Cambridge, Massachusetts.

HARTLEY, R., AND ZISSERMAN, A. 2000. *Multiple View Geometry in computer vision.* Press Syndicate of the University of Cambridge.

KAWAMOTO, K., AND IMIYA, A. 2001. Detection of spatial points and lines by random sampling and voting procedure. *Pattern Recognition Letters 22*, 2 (February), 199–207.

LI, M., MAGNOR, M., AND SEIDEL, H.-P. 2003. Improved hardware-accelerated visual hull rendering. *Proc. Vision, Modeling, and Visualization (VMV-2003), Munich, Germany* (November), 151–158.

LOK, B. 2001. Online model reconstruction for interactive virtual environment. In *Proceedings 2001 Symposium on Interactive 3D Graphics*, 69–72.

REDERT, A., HENDRIKS, E., AND BIEMOND, J. 1999. Correspondence estimation in image pairs. *IEEE Signal Processing Magazine 16*, 3, 29–46.

RODRIGUES, R., FERNANDES, A., VAN OVERVELD, K., AND ERNST, F. 2002. Reconstructing depth from spatiotemporal curves. In *Proceedings of the 15th International Conference on Vision Interface*, 252–259.

SAINZ, M., BAGHERZADEH, N., AND SUSIN, A. 2002. Hardware accelerated voxel carving. In *1st Ibero-American Symposium in Computer Graphics (SIACG 2002)*, 289–297.

SATO, J., AND CIPOLLA, R. 1999. Affine reconstruction of curved surfaces from uncalibrated views of apparent contours. *IEEE Trans. Pattern Analysis and Machine Intell. 21*, 11 (November), 1188–1198.

SCHULZ-MIRBACH, H., AND WEISS, I. 1994. Projective reconstruction from curve correspondences in uncalibrated views. Technical Report TR-402-94-014, Technical University of Hamburg-Harburg.

SEGAL, M., KOROBKIN, C., VAN WIDENFELT, R., FORAN, J., AND HAEBERLI, P. 1992. Fast shadows and lighting effects using texture mapping. In *Proceedings of SIGGRAPH'92*, ACM, E. E. Catmull, Ed., 249–252.

SHREINER, D., Ed. 2000. OpenGL Reference Manual, OpenGL Architecture Review Board, Addison-Wesley.

SMITH, S. M. 1998. ASSET-2: Real-time motion segmentation and object tracking. *Real-Time Imaging 4*, 1, 21–40.

TELL, D., AND CARLSSON, S. 2000. Wide baseline point matching using affine invariants computed from intensity profiles. In *ECCV (1)*, 814–828.

YANG, R., WELCH, G., AND BISHOP, G. 2002. Real-time consensus-based scene reconstruction using commodity graphics hardware. In *Proceedings of Pacific Graphics 2002*.

ZACH, C., KARNER, K., AND BISCHOF, H. 2004. Hierarchical disparity estimation with programmable 3d hardware. In *WSCG'2004 SHORT Communication papers proceedings*.