NARROW BASELINE GLSL MULTIVIEW STEREO

Pekka Paalanen
Machine Vision and Pattern
Recognition Laboratory (MVPR)
Lappeenranta Univ. of Technology, Finland
paalanen@lut.fi

Joni-Kristian Kamarainen
MVPR/Computational
Vision Group, Kouvola
Lappeenranta Univ. of Technology, Finland
jkamarai@lut.fi

Abstract

Our ultimate goal is a system capable of on-line 3D reconstruction from a monocular video and running on commodity hardware. One intrinsic module in such a system is a fast stereo algorithm, which produces depth maps from given views with known camera calibration and pose. Plane-sweep stereo algorithms are well-suited for real-time GPU implementation and can easily be generalised to true multi-image matching, which makes them attractive.

This paper presents a simple multiview stereo algorithm implemented for GPUs: GPU-MVS. The details of realising the algorithm with standard (and extended) OpenGL with GL Shading Language (GLSL) are given along with hints on how to make the implementation co-operate well with other, CPU-critical, program modules. The stereo implementation is evaluated on the Middlebury Stereo Evaluation framework for two-view case. The effects of adding more views are investigated, and the final experiment shows some reconstruction results from a complete from-camera-to-3D reconstruction pipeline. To facilitate fair comparisons we will publish the GPU-MVS source code in the Web.

1. Introduction

The demand for 3D models in computer graphics, games, virtual reality, communication and education has grown rapidly. As creating 3D models is becoming more useful for the common person, for example scanning real objects into a virtual on-line world or a game, or recording a car crash site, a cheap 3D scanning device is needed. The goal of our research is to create an on-line 3D reconstruction system with a standard monocular (colour) camera as its sole sensor. The system must run on a modern consumer grade PC providing an interface to a camera (*e.g.* USB web camera or a cheap IEEE1394 camera) and a relatively modern Graphics Processing Unit (GPU). As a user likely already has a suitable PC, the missing parts are the software and perhaps a camera. The on-line (real-time) performance of the software would allow the user to see the reconstruction in progress, guiding the scanning. This paper attempts to assist the machine vision community in creating such software by describing a popular piece of the dense 3D reconstruction pipeline: a multiview stereo algorithm.

An architecture for such a dense 3D reconstruction pipeline has been proposed in [15]. Also other systems [8, 16, 17, 18, 22] have been presented, but they either use more or different sensors, exploit artificial markers, or employ off-line methods. In [15], visual simultaneous localisation and mapping [2, 10] (VSLAM) framework estimates camera poses for each video frame, and automatically selected sets of frames are processed by a multiview stereo (MVS) algorithm to create 3D point clouds. The 3D point clouds are then merged into a 3D model. As CPU time is mostly consumed by the VSLAM (or any other egomotion tracking system, *e.g.* [9]), MVS is implemented on the GPU.

The standard stereo reconstruction considers two (rectified) views and produces a dense depth map for one or both of the views. By definition this is a narrow baseline case. The views are from the same side of a scene, and a single depth map $d = f(x, y)$ is sufficient to represent the visible part of the 3-dimensional scene. In contrast, a wide baseline reconstruction from several views may produce a fully 3-dimensional surface model which cannot be represented simply as $d = f(x, y)$ modulo a Euclidean transformation. Multiview stereo as described here is the narrow baseline case but with several views and with the goal of computing a depth map for one of them.

The literature [20, 12, 13] on two-view stereo is vast and out of the scope of this paper. Benchmarking has been a problem in the past, but nowadays several multiview data sets are available, *e.g.* Seitz *et al.* [23] for the wide baseline reconstruction and Strecha *et al.* [24] for high resolution depth map evaluation. Probably the most well-known stereo benchmarking data sets are the Middlebury sets [19, 21, 6].

East Finland Graduate School in Computer Science and Engineering is acknowledged for funding.

The contribution of this paper is the detailed description of a basic multiview stereo algorithm for colour images (Section 3) and its implementation for a GPU (Section 4). The implementation is written in OpenGL and GL Shading Language. Practical notes on achieving real-time performance in a multithreaded 3D reconstruction system are given in Section 4.1.

2. Related Work

Our multiview stereo approach was inspired by Yang and Pollefeys [30, 31]. Their approach has several benefits. The images need not be rectified and therefore an epipole in an image is not a problem. The approach generalises easily to an arbitrary number of views (multiview). It can be efficiently implemented on a graphics processing unit (GPU) offering excellent performance. The approach is fast enough for on-line systems and a GPU implementation leaves the CPU for other computations.

The basic idea of plane sweep (space-sweep) was first introduced by Collins [1]. He defines the term true multi-image matching, and presents a space-sweep approach adhering to the definition. Another plane sweep approach is used in [30, 31] as follows. Hypothesise a plane in 3-space. Images taken by known cameras can be back-projected onto the plane. If the scene has a surface point on the plane, the back-projected images should agree at that point. A matching cost measuring the dissimilarity of the back-projected images is computed for each point (pixel) on the hypothesised plane. The costs are computed for several planes at different depths, and a depth map is produced by selecting the best depth hypothesis for each pixel by the cost.

GPU-MVS differs from [31] in matching cost computation. They used absolute differences where GPU-MVS computes the total colour variance (as proposed in [31]). GPU-MVS implements the cost aggregation using a proper Gaussian window instead of multiresolution approximations.

3. Multiview Stereo Algorithm

The high level description of multiview stereo is given in Alg. 1. The relation of the resulting depth map to the scene geometry is defined by the reference camera. The reference camera may be chosen freely, provided a depth map can be reconstructed for that point of view. The depth map is accumulated while iterating over different plane depths. On each iteration, a pixel's depth and cost in the depth map are updated. The main parts of the proposed multiview stereo method are presented next.

The planes for the plane sweep approach are orthogonal to the principal ray of the reference camera. Depth d of a plane is the orthogonal distance from the reference camera centre. The depth range from d_{near} to d_{far} is discretised to

Algorithm 1 Plane sweep algorithm for depth map from multiple views.

```
1: Choose the reference camera.
2: Init the depth map with unknown depths and infinite costs.
3: for all depths $d$ do
4:     Project each image onto the plane at depth $d$.
5:     Compute per pixel cost values from the projections into a
       cost image.
6:     Filter the cost image (blur).
7:     for all pixels in the depth map do
8:         Update the depth and cost values, if the cost in the cur-
           rent cost image is less than the stored in the depth map.
9:     end for
10: end for
```

$$d_m = \left(\frac{m}{N_{\text{d}}} \cdot \frac{1}{d_{\text{near}}} + \frac{N_{\text{d}} - m}{N_{\text{d}}} \cdot \frac{1}{d_{\text{far}}} \right)^{-1} \quad , \tag{1}$$

where $m = 0, \ldots, N_{\text{d}}$. The division is linear in inverse depth, which is linear in disparity, and fits well for the pixel based processing. Precision of a depth estimate is better near than far, which is natural for stereo.

A simple example of a matching cost between two grey images $I_i(x, y)$ is the squared difference $(I_1(x, y) - I_2(x, y))^2$. Using a larger support than just one pixel helps to remove false matches (sum of squared differences, SSD) and it is commonly used with few images. SSD does not generalise directly to the multiview case. By Yang and Pollefeys' [31] suggestion the (colour) variance over images is used instead. The cost image for depth d_m is

$$\zeta_m(x, y) = \frac{1}{N_{\text{V}}} \left(\sum_{i=1}^{N_{\text{V}}} \boldsymbol{I}_i(u_i, v_i)^{\text{T}} \boldsymbol{I}_i(u_i, v_i) \right) - \boldsymbol{s}^{\text{T}} \boldsymbol{s} \ ,$$

$$\boldsymbol{s} \equiv \frac{1}{N_{\text{V}}} \sum_{i=1}^{N_{\text{V}}} \boldsymbol{I}_i(u_i, v_i) \ , \tag{2}$$

where N_{V} is the number of views. (x, y) are pixel coordinates on the depth map, and are congruent with the pixel coordinates on the reference view. (x, y) are first back-projected from the reference camera to the plane at depth d_m and then projected to each of the images $\boldsymbol{I}_i$ producing image coordinates (u_i, v_i). The value $\boldsymbol{I}_i(u_i, v_i)$ for an RGB-image is a vector in $[0, 1]^3$. The cost $\zeta_m(x, y)$ is the total variance of the RGB-colours, a scalar.

The cost image $\zeta_m(x, y)$ contains cost values computed from single pixels. To gain a larger support $\zeta_m(x, y)$ is filtered with a circular Gaussian filter Ψ. The final cost image

$$\xi_m = \zeta_m * \Psi \tag{3}$$

for depth d_m is effectively computed over a Gaussian window instead of just one pixel.

The depth map $d(x, y)$ for the reference view is

$$d(x, y) = d_{\hat{m}} \mid \hat{m} = \arg\min_m \xi_m(x, y) \ . \tag{4}$$

Some depth estimates $d(x, y)$ are discarded based on the following criteria. The cost must be below a constant threshold to detect clearly erroneous matches, and it must significantly differ from the average cost over all depth hypotheses. The latter is determined as the average cost minus a constant τ_{uniq} times the cost standard deviation,

$$\xi_{\hat{m}}(x, y) < \mathop{\mathrm{E}}_m(\xi_m(x, y)) - \tau_{\text{uniq}} \mathop{\mathrm{Std}}_m(\xi_m(x, y)) \ . \tag{5}$$

4. The GPU Implementation

The GPU implementation of the multiview stereo algorithm sketched above is written in C++ using standard OpenGL, some OpenGL extensions and GLSL (GL Shading Language). No proprietary computational frameworks have been used, *e.g.* Nvidia's CUDA. In theory this should allow portability over to other manufacturers' GPUs. The implementation was first written for Nvidia GeForce 6600 and then used and refined with GeForce 8500 GT. The presented algorithms are mostly realised as shader programs, partly as fixed function graphics processing (stencil and depth buffer operations), and some CPU operations (*e.g.* forming transformation matrices and updating τ_{stl}).

The implementation uses half precision floating point (16 bits: a sign bit, 10-bit precision and 5-bit exponent) textures. The conventional 8-bit integer format would have required careful truncation and scaling of cost values for reasonable accuracy in depth map estimation [4]. The three half precision 4-channel textures are `varTex`, `depTex` and `tmpTex`. In addition, an 8-bit stencil buffer `stlBuf` and 24-bit depth buffer `depBuf` are used. In the following, notice the difference between a pixel's depth as related to the sweeping plane (Z-coordinate), and a value in the depth buffer which will actually be the cost value.

All input images (the views) are uploaded to the graphics card as 8-bit/channel RGB textures. Lens distortions are corrected at the upload phase by rendering the views through an undistorting fragment shader, that computes accurate sampling positions on the fly for a distortion model. Each input image is assigned to a different texturing unit so that all images can be sampled in a single rendering pass in the cost image computation step. The number of texturing units sets the limit on how many views can be handled in this implementation. With modern consumer hardware the limit can be as high as 32.

The algorithm for a GPU is described in Alg. 2. For each plane hypothesis a cost image is computed into `varTex`, filtered in two passes using the temporary `tmpTex`, and then merged into `depTex` with `stlBuf` and `depBuf`. Pruning poor depth estimates and repacking the depth image are the last passes before downloading the depth map

Algorithm 2 Multiview stereo on a GPU.

```
Require: 16-bit floating point 4-channel textures varTex, depTex and tmpTex. 8-bit stencil buffer stlBuf and 24-bit depth buffer depBuf. 1D Gaussian filter $\psi(\delta)\ \mathcal{N}(0, \sigma^2)$.
Input: Set of $N_V$ views $\{V_i\}$, $V_i = \{\boldsymbol{I}_i(u, v), {}^{W}\boldsymbol{T}_{C_i}\}$ (image, and camera-to-world Euclidean transformation), camera calibration $\boldsymbol{K}$, depth range $[d_{\text{near}}, d_{\text{far}}]$ with $N_d$ depth steps.
Output: Depth map $D(u, v)$.
 1: Choose ref. view $i$ (define coordinate frame R), ${}^{W}\boldsymbol{T}_R \leftarrow \boldsymbol{T}_i$.
 2: for all $V_i$ do
 3:     Upload $\boldsymbol{I}_i$ as an RGB-texture.
 4:     Undistort, if necessary.
 5:     ${}^{C_i}\boldsymbol{T}_R = {}^{C_i}\boldsymbol{T}_W\ {}^{W}\boldsymbol{T}_R$
 6: end for
 7: depTex$(*, *) \leftarrow (0.0, 0.0, 0.0, 0.0)^T$,
        depBuf$(*, *) \leftarrow 1.0$
 8: for $m = 0$ to $N_d$ do
 9:     $d \leftarrow \left(\frac{m}{N_d} \cdot \frac{1}{d_{\text{near}}} + \frac{N_d - m}{N_d} \cdot \frac{1}{d_{\text{far}}}\right)^{-1}$
10:     call Alg. 3 // Compute a cost image into varTex.
11:     Filter varTex with $\psi(\delta)$ horizontally and vertically, updating stlBuf and $\tau_{\text{stl}}$.
12:     for all $(u, v) \in$ depTex | stlBuf$(u, v) \geq \tau_{\text{stl}}$ do // update best depth
13:         $(-, q, -, -)^T \leftarrow$ varTex$(u, v)$
14:         if $q <$ depBuf$(u, v)$ then
15:             depBuf$(u, v) \leftarrow q$
16:             depTex$(u, v) \leftarrow (-, -, -, d)^T$
17:         end if
18:     end for
19:     for all $(u, v) \in$ depTex | stlBuf$(u, v) \geq \tau_{\text{stl}}$ do // accumulate statistics
20:         $(-, q, -, -)^T \leftarrow$ varTex$(u, v)$
21:         depTex$(u, v) \leftarrow$ depTex$(u, v) + (1.0, q, q^2, -)^T$
22:     end for
23: end for
24: call Alg. 4 // Prune the depth map (depTex) into tmpTex.
25: $(D(u, v), -, -, -)^T \leftarrow$ tmpTex$(u, v)$   $\forall u, v$
```

from the graphics card into system memory. In the notation, on line 13 in Alg. 2 only the second component of the 4-vector `varTex`(u, v) is read into q, and on line 16 only the fourth of `depTex`(u, v) is written.

The computation of a cost image for a given depth d is described in Alg. 3. The input is the set of views $\{V_i\}_1^{N_V}$ each containing the (undistorted) image $\boldsymbol{I}_i$ and the transformation ${}^{C_i}\boldsymbol{T}_R$ from the reference to the camera i coordinate frame. The intrinsic matrix $\boldsymbol{K}$ is assumed common for all cameras. Each pixel (u, v) in `varTex` is back-projected to the point $\boldsymbol{X}$ of depth $Z = d$. $\boldsymbol{X}$ is projected into every view. If any of the images of $\boldsymbol{X}$ is not inside the corresponding view (image), the pixel (u, v) is left blank (undetermined). Otherwise, the pixel is assigned a cost value q, see Eq. 2. A blank pixel is indicated by the first component of `varTex` not being 1.0, and the stencil value not incremented. In later phases when stencil values are compared

Algorithm 3 Cost image computation.

```
Require: Texture varTex, stencil buffer stlBuf.
Input: Set of $N_\mathrm{V}$ views $\{V_i\}$, camera calibration $\boldsymbol{K}$, depth hypothesis $d$.
Output: Texture varTex, stencil buffer stlBuf, threshold $\tau_\mathrm{stl}$.
 1: varTex$(*,*) \leftarrow (0.0, 0.0, -, -)^\mathrm{T}$,
      stlBuf$(*,*) \leftarrow 0$
 2: $\tau_\mathrm{stl} \leftarrow 0$
 3: for all $(u, v) \in$ varTex do
 4:   $\boldsymbol{x} = \boldsymbol{K}^{-1}(u, v, 1)^\mathrm{T}$
 5:   $\boldsymbol{X} = (\frac{d}{\boldsymbol{x}_\mathrm{w}}\boldsymbol{x}^\mathrm{T}, 1)^\mathrm{T}$
 6:   for all $V_i$ do
 7:     $\boldsymbol{x}' = \boldsymbol{K}\left[\boldsymbol{I} \middle| \boldsymbol{0}\right] {}^{\mathrm{C}_i}\boldsymbol{T}_\mathrm{R}\boldsymbol{X}$
 8:     $(u', v') = (\boldsymbol{x}'_\mathrm{x}/\boldsymbol{x}'_\mathrm{w}, \boldsymbol{x}'_\mathrm{y}/\boldsymbol{x}'_\mathrm{w})$
 9:     if $(u', v') \notin \boldsymbol{I}_i$ then
10:       skip to next $(u, v)$
11:     end if
12:     $\boldsymbol{c}_i \leftarrow \boldsymbol{I}_i(u', v')$
13:   end for
14:   $\boldsymbol{s} = \frac{1}{N_\mathrm{V}}\sum_i \boldsymbol{c}_i, \qquad q = \frac{1}{N_\mathrm{V}}\left(\sum_i \boldsymbol{c}_i^\mathrm{T}\boldsymbol{c}_i\right) - \boldsymbol{s}^\mathrm{T}\boldsymbol{s}$
15:   varTex$(u, v) \leftarrow (1.0, q, -, -)^\mathrm{T}$
16:   stlBuf$(u, v) \leftarrow$ stlBuf$(u, v) + 1$
17: end for
18: $\tau_\mathrm{stl} \leftarrow \tau_\mathrm{stl} + 1$
```

against τ_stl, the pixel will be automatically skipped. As the result, Alg. 3 produces a cost image from single pixels, and a stencil mask in `stlBuf` that together with τ_stl offers an efficient way to ignore blank pixels.

The circular Gaussian filter of Eq. 3 is separable, and therefore the filtering is done in two one-dimensional passes (line 11 in Alg. 2). The horizontal filtering step reads from `varTex` and writes to `tmpTex`, the vertical filtering step does the opposite. The 1D filter is effectively 9-tap, but the implementation takes advantage of linear texture interpolation, so that the source texture needs to be sampled only five times per destination pixel. Any blank pixel on the filter's total 9×9 area causes the current pixel to also turn blank, since the filter response is undetermined. The stencil buffer and τ_stl are updated to account for the new blank pixels. Filtering provides an enlarged matching window for stereo correspondence and erosion as a side-effect.

In Alg. 2, `depTex` starts zeroed and `depBuf` is initialised to the maximum (OpenGL depth) value. For each depth d, the cost image is computed and filtered into `varTex`. The last per-depth phase is merging the cost image into `depTex`. The merge is done in two passes: the first pass updates the pixel depth values and costs, and the second accumulates statistics for later use. The stencil mask (`stlBuf` and τ_stl) is used to skip blank pixels in `varTex`. The cost value from `varTex` is routed to the fragment depth component to be used in depth testing. In the first pass, if the fragment depth (cost) is less than the value in depth buffer `depBuf`, the fragment depth is written into `depBuf` and the depth d is written to the fourth component in `depTex`. Over all values of d, this results in the minimum cost values in `depBuf` and the corresponding values of d in `depTex`. The second pass ignores depth testing and uses blending to accumulate statistics over those depth hypotheses, where the pixel in `varTex` is not blank. The first component in `depTex` is the number of such hypotheses, the second component is the sum of costs, and the third component is the sum of squared costs. The mean and variance can be computed from these three numbers.

Although the statistics accumulation into `depTex` is correct in theory, one practical issue is worth noting. The precision of 16-bit floating point values is poor, only 10 bits. For a large number of depths d, the rounding errors may become significant enough to make the final processing phase to fail. According to experimentation, a few hundred depths is supportable, but few thousand is not.

The final GPU processing phase is Alg. 4, discarding poor depth estimates. The contents of `depTex` are rendered into `tmpTex` again through a fragment shader program. There are several requirements for depth estimates:

- $n \geq 30$, at least 30 non-blank depth hypotheses to allow meaningful statistics.
- The depth estimate may not be at the ends of the swept depth range, otherwise the correct depth is likely to be outside of the range.
- The average cost μ over the depth hypotheses must be at least τ_avg. This discards featureless points.
- The depth estimate cost must be below the hard limit τ_cost, or the match is likely incorrect.
- The depth estimate cost must be significantly below the average cost to discard unstable matches (Eq. 5).

The parameters τ_avg, τ_cost and τ_uniq are user definable. In the output the blank and discarded pixels are set to depth zero, which is an invalid value in a depth map. The stencil mask is not used in this phase, since no stencil mask exists for `depTex`. Requirement tests 4 and 5 take advantage of OpenGL depth testing in line 14 of Alg. 4.

4.1. Implementation Real-time Considerations

The only Nvidia-specific OpenGL extension used in the implementation is GL_NV_fence. Fences are markers that are inserted into the GPU command stream. A fence is signalled when the GPU execution has passed it. Fences offer a way to know when the GPU has executed all operations queued up to that point.

The multiview stereo GPU implementation is used as a part of a larger soft-real-time system, which has several threads of execution. Some of the threads are "more real-time" than others, and the GPU-MVS thread is more of a background task. The GPU-MVS thread needs to wait for the GPU to finish processing before using the results. Setting up a fence and putting the thread to sleep for a short

Algorithm 4 Discarding poor depth estimates.

```
Require: Texture tmpTex.
Input: Texture depTex, depth buffer depBuf, thresholds $\tau_{\text{uniq}}$, $\tau_{\text{avg}}$ and $\tau_{\text{cost}}$.
Output: Texture tmpTex
 1: tmpTex(*,*) $\leftarrow (0.0, 0.0, 0.0, 0.0)^{\mathrm{T}}$
 2: $d_{\min} \leftarrow d_{N_\mathrm{d}-2}$,   $d_{\max} \leftarrow d_2$ // See Eq. 1
 3: for all $(u, v) \in$ tmpTex do
 4:     $(n, \Sigma_\mathrm{q}, \Sigma_{\mathrm{q}^2}, d)^{\mathrm{T}} \leftarrow$ depTex$(u, v)$
 5:     if $n < 29.5$ or $d < d_{\min}$ or $d > d_{\max}$ then
 6:         skip to next $(u, v)$
 7:     end if
 8:     $\mu \leftarrow \Sigma_\mathrm{q}/n$
 9:     if $\mu < \tau_{\text{avg}}$ then
10:         skip to next $(u, v)$
11:     end if
12:     $\sigma \leftarrow \sqrt{\Sigma_{\mathrm{q}^2}/n - \mu^2}$
13:     $r = \min\{\tau_{\text{cost}}, \max\{0, \mu - \tau_{\text{uniq}}\sigma\}\}$
14:     if $r >$ depBuf(u,v) then
15:         depBuf$(u, v) \leftarrow r$
16:         tmpTex$(u, v) \leftarrow (d, -, -, -)^{\mathrm{T}}$
17:     end if
18: end for
```

time releases the CPU for other threads. The fence is tested and until it is signalled, the sleep cycle repeats. This method was chosen, because the Linux kernel scheduler settings either were not sufficient, or they required root privileges and posed the danger of locking up the whole operating system (true real-time processes).

Another reason to use fences is related to the GPU command stream. The GPU command buffer has a finite size, and when it fills, the CPU has to stop feeding commands and wait for the buffer to drain. Usually OpenGL drivers aim for maximum performance, so the wait is likely implemented as a busy-wait. Running the CPU in a busy-wait loop does not give other threads a chance to run, therefore hurting the system performance as a whole. Fences may introduce waits, but during the wait other threads can execute.

Fences are set and waited for at two points of the GPU-MVS implementation. The first point is after line 22 of Alg. 2, where the fence is on every eighth cycle. A fence on every cycle would incur needless overhead, and on the other hand the GPU command buffer should not be exhausted. The second point is just before reading the repacked depth map data into system memory.

5. Experiments

Three types of test results are reported for GPU-MVS. First, the canonical Middlebury Stereo Evaluation [19] compares the accuracy to other real-time methods in the classic stereo setup. Second, using the Middlebury multiview image sets, performance is studied with respect to the number of views. Last, GPU-MVS is presented as a part of the real-time monocular 3D-reconstruction system.

All the experiments have been executed in an x86_64 Linux environment, on a machine with AMD Athlon64 3500+ processor and Nvidia GeForce 8500 GT using the proprietary Nvidia graphics drivers version 185.18.31. Note, that the graphics card is a low-end one. For comparison, a GeForce 9800 GTX+ is 2–6 times as fast in the multiview tests (Table 2) as the 8500.

5.1. Middlebury Stereo Evaluation

The multiview stereo algorithm was evaluated on the standard Middlebury Stereo Evaluation image pairs, and the results from the web service [19] are in Table 1. Fast stereo methods, where the publication was available, were selected for comparison. Since the Middlebury evaluation regards a missing disparity estimate as an error, an ad hoc left–right consistency check and hole filling were implemented for these results only.

For the consistency check, depth maps are generated for both views independently. If the left–right disparity mismatch is greater than 0.5 pixels, the depth estimate is removed, creating a hole. Holes in the depth maps are filled by simple linear or constant scanline interpolation. Scanline interpolation often produces streaking artifacts. The consistency check and hole filling are not part of the multiview stereo implementation, because scanline-based algorithms are not suitable for a general configuration of multiple views. Hole filling is not needed in our target application, where multiple stereo reconstructions are generated anyway, but is essential for success in [19].

Since the multiview stereo implementation requires full camera poses and produces depth maps instead of disparity maps, some impedance matching is required to run disparity-based tests. Artificial camera poses are manufactured based on the rectified view geometry, and a resulting floating point depth map is converted to an 8-bit per pixel disparity map as required by [19]. The depth range for the plane sweep was computed from the disparity range for each pair. The disparity ranges are: Tsukuba 0.01–15.5, Venus 0.01–19.5, Teddy 0.01–59.5, and Cones 0.01–59.5. Resolution of the plane sweep is 0.1 pixels of disparity. The implementation cannot handle a disparity of exactly zero pixels (infinite depth), so 0.01 is used as the minimum.

Table 1 contains three results for the proposed multiview stereo implementation denoted as ”Our method”. Method A is the plain multiview stereo without any post-processing. Method B contains the hole filling without left–right consistency check, and Method C uses both the consistency check and hole filling. None of the fast methods rank within the top 30 methods on average, but this is explained by the trade-off with speed. Our error scores, however, are comparable to other fast methods despite the fact that it is not specifically dedicated for this benchmark problem.

Table 1. Middlebury Stereo Evaluation, version 2 results. Error threshold 0.5, ordered by average non-occluded region error score. [20, 19]

	Avg.	Tsukuba			Venus			Teddy			Cones		
Algorithm	rank	n.occ	all	disc	n.occ	all	disc	n.occ	all	disc	n.occ	all	disc
PlaneFitBP [28]	36.7	12.7	13.6	16.2	8.58	9.12	12.7	18.4	24.3	31.4	15.3	21.9	23.9
SegTreeDP [3]	39.8	25.4	26.0	24.6	6.23	6.59	10.8	19.7	25.9	30.9	10.9	16.5	17.7
RealtimeBP [29]	48.4	19.9	21.6	22.2	8.68	9.93	20.1	19.2	24.8	33.8	14.2	21.2	25.9
RealtimeBFV [32]	45.6	24.5	25.0	21.3	8.64	9.12	13.5	18.1	24.2	32.1	15.0	20.6	22.9
RealTimeGPU [27]	57.3	24.2	26.0	24.9	10.9	12.1	27.6	19.6	27.0	33.0	16.5	23.7	29.5
FastAggreg [25]	53.5	23.1	23.9	19.6	15.8	16.6	19.6	21.1	27.4	34.0	15.5	22.0	23.1
ReliabilityDP [5]	56.1	19.0	20.7	17.5	12.7	14.0	26.1	26.3	32.5	36.8	23.7	29.9	31.5
Our method C	65.6	23.5	25.3	41.0	14.6	15.7	42.6	25.8	32.2	47.0	18.2	24.9	38.3
TreeDP [26]	60.7	22.4	23.1	22.3	12.1	12.9	21.7	32.4	38.9	45.6	23.7	30.8	31.7
Our method B	69.3	25.2	26.9	41.4	20.3	21.6	45.8	27.7	35.2	49.3	18.6	27.9	39.2
Our method A	71.2	32.3	33.9	43.4	32.4	33.5	46.6	28.8	36.2	49.8	19.8	29.1	40.1

5.2. Multiview results

The web service at [19] is strictly for classic two-view stereo and does not discriminate between missing and erroneous estimates. Therefore further tests were run with the Middlebury 2003 (Teddy, Cones) and 2005 (Art, Books, Dolls, Laundry, Moebius, Reindeer) sets [21, 6]. The performance numbers are based on classifying a disparity difference > 0.5 as an error. Pixels without ground truth are ignored and pixels without a disparity estimate (but with ground truth) are classified as misses instead of errors.

The results are in Table 2 containing the percentages of good, bad and missed disparities. Each row is a single execution. Column # denotes the number of views used and the views are listed in the first column. Computational speed is given as millions of disparity evaluations (MDE) and millions of pixel reads (Mpx) per second. MDE is the depth map size in pixels $\times$ #disparity levels. Mpx is MDE $\times$ #views, relating to the total number of input pixels processed over the disparity range. The Teddy and Cones image sets are quarter-size (450×375) with ground truth disparities given for views 2 and 6. The other six image sets are half-size, mostly 695×555, with ground truth for views 1 and 5. The depth map is created for the view 2 or 1, respectively. The Teddy and Cones two-view sets are identical to the stereo case.

Table 2 shows that adding views between the outermost views increases the good percentage, but adding views for wider baseline reduces it. The percentage of bad disparities decreases as more views are added, regardless of the new views' position, with Art the only exception.

Table 2. Middlebury multiview results with varying sets of views.

			percent		MDE	Mpx
Set (views)	#	Good	Bad	Miss	per sec.	
Teddy `(2,6)`	2	63.7	26.8	9.5	27	54
`(..2.4.6..)`	3	65.2	25.2	9.6	44	131
`(..23456..)`	5	65.9	24.4	9.8	33	163
`(.12.4.67.)`	5	61.1	22.8	16.0	33	164
`(.1234567.)`	7	62.7	21.6	15.7	32	226
`(012345678)`	9	56.8	19.9	23.4	25	222
Cones `(2,6)`	2	71.1	20.8	8.1	44	87
`(..2.4.6..)`	3	71.3	20.2	8.5	44	131
`(..23456..)`	5	72.5	19.2	8.3	33	164
`(.12.4.67.)`	5	63.7	19.2	17.1	33	164
`(.1234567.)`	7	64.5	18.7	16.9	32	225
`(012345678)`	9	57.7	15.9	26.4	25	223
Art `(1,5)`	2	43.1	51.0	5.9	45	89
`(.1.3.5.)`	3	43.8	48.8	7.4	49	146
`(.12345.)`	5	44.1	47.6	8.3	38	190
`(01.3.56)`	5	39.0	49.3	11.7	37	185
`(0123456)`	7	39.6	48.3	12.2	36	249
Books `(1,5)`	2	52.4	37.1	10.5	55	109
`(.1.3.5.)`	3	56.0	33.0	10.9	49	146
`(.12345.)`	5	55.9	33.0	11.2	38	191
`(01.3.56)`	5	53.2	31.4	15.5	42	211
`(0123456)`	7	53.6	31.0	15.4	35	248
Dolls `(1,5)`	2	56.9	36.3	6.9	54	109
`(.1.3.5.)`	3	57.4	34.9	7.7	49	146
`(.12345.)`	5	57.6	34.5	8.0	41	204
`(01.3.56)`	5	54.6	33.7	11.7	42	212
`(0123456)`	7	54.8	33.5	11.7	36	249
Laundry `(1,5)`	2	38.8	54.4	6.8	58	117
`(.1.3.5.)`	3	43.6	49.1	7.3	50	150
`(.12345.)`	5	44.5	47.7	7.8	42	209
`(01.3.56)`	5	42.1	48.2	9.7	44	218
`(0123456)`	7	42.3	48.0	9.7	33	229
Moebius `(1,5)`	2	58.8	34.0	7.3	54	109
`(.1.3.5.)`	3	60.9	31.4	7.7	49	146
`(.12345.)`	5	61.3	29.8	9.0	41	204
`(01.3.56)`	5	58.4	30.8	10.8	42	211
`(0123456)`	7	58.7	29.7	11.6	36	249
Reindeer `(1,5)`	2	52.4	37.5	10.1	58	115
`(.1.3.5.)`	3	54.9	34.5	10.6	50	150
`(.12345.)`	5	55.2	33.4	11.4	42	209
`(01.3.56)`	5	51.6	33.8	14.6	44	219
`(0123456)`	7	52.4	32.1	15.4	33	229

5.3. Monocular 3D Reconstruction

The target application of our GPU-MVS implementation is the real-time monocular 3D reconstruction where the GPU-MVS can replace the classic two-view disparity computation of [11, 14] and moves the computational burden of stereo from the CPU to the GPU. Being a multiview stereo algorithm, more views are gathered into a single reconstruction set, here a maximum of eight.

The reconstruction system contains a post-processing step for depth maps. A plane is least-squares fitted to the (regular) 3×3 neighbourhood of each point (x, y) in a depth map $d(x, y)$, and the surface normal is approximated from the fit corrected by the back-projection partial derivatives. This approximation is used instead of back-projecting all 9 points into 3D to speed up the plane fit computation. There are two conditions based on the plane fit. The sample $d(x, y)$ is discarded if $d(x, y)$ is too far from the point pre-

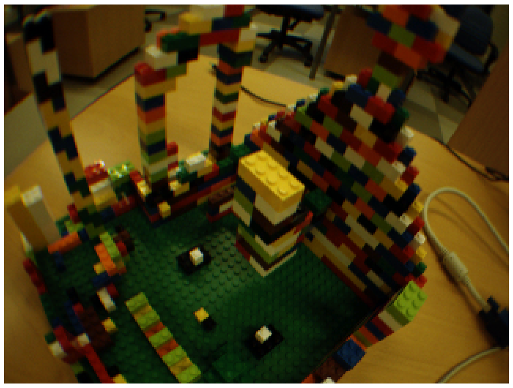
Figure 1. 320×240 frame from the 3D reconstruction input video.

dicted by the neighbourhood. If the plane normal direction differs too much from the camera principal ray, the sample is discarded as being seen from a grazing angle, which indicates a poor match. Particularly, GPU-MVS may produce points with normals pointing away from the camera, which is clearly amiss. These conditions compensate for the missing left–right consistency check.

Without proper ground truth, only qualitative results can be presented. Fig. 1 shows a frame from the input video for the monocular 3D reconstruction experiment. The video is shot of an object made of Legos, 382 frames at 30 frames per second equals 12.7 seconds in length. The system runs GPU-MVS seven times on different sets of video frames, and combines the resulting point clouds without further 3D registration, similar to [15], except that our underlying data structure is an octree with limited depth. In this example, the GPU-MVS runs approximately for one second per an eight view set, with 300 depth hypotheses.

The reconstruction is seen in Fig. 2 from a novel viewpoint. The reconstruction is simply a coloured point cloud, each point rendered as a circular sprite with the size inversely proportional to the eye distance. The two views are crossed-eye stereo pairs. With some practise, a reader may experience depth by crossing her eyes and relaxing until the eyes become focused. The sense of depth is proportional to the viewing distance.

6. Discussion

This paper presented a simple multiview stereo method and its implementation for narrow baseline depth inference. The goal was to develop a reconstruction method that is fast and accurate, but is not required to produce a fully dense depth map. For the primary application of incremental 3D reconstruction, the partial reconstructions are not required to be complete, but they should not contain false information. One of the implementation requirements was very low CPU load—therefore the GPU was utilised.

The GPU-MVS implementation was tested against the de facto standard Middlebury Stereo Evaluation [19], where it performed only moderately when compared to the other real-time methods. That evaluation framework however has a different goal compared to our work: it aims for a complete dense depth map from two views. First, GPU-MVS can use more than two views, and second, most importantly, we are not interested in guessing depths for pixels where the depth cannot be reliably determined. The evaluation simply considers missing depth estimates as errors.

The multiview experiments investigated the effect of the number of views to depth map accuracy and computational speed. The interesting result in Table 2 is that adding views between the two outermost views improved accuracy, but extending the baseline with new views deteriorated it. The speed of disparity evaluations per second went down when views are added, but the GPU architecture made the processing speed of input pixels per second go up. Moreover, preliminary results of the method in a complete incremental on-line 3D reconstruction were demonstrated and a portable demo system on a laptop exists.

GPU-MVS does not explicitly take into account occlusions or depth discontinuities, nor does it do any left–right consistency kind of checking to prune the depth map. Experience shows that these features would be very useful. Furthermore, the stereo matching could possibly be improved by adapting the results from [4, 7] to colour and multiview processing. All source code will be published at http://www2.it.lut.fi/project/rtmosaic/.

References

[1] R. Collins. A space-sweep approach to true multi-image matching. In *CVPR*, 1996. 2

[2] A. Davison, I. Reid, N. Molton, and O. Stasse. Monoslam: Real-time single camera slam. *IEEE TPAMI*, 29(6):1052–1067, Jun 2007. 1

[3] Y. Deng and X. Lin. A fast line segment based dense stereo algorithm using tree dynamic programming. In *ECCV*, 2006. 6

[4] M. Gong, R. Yang, L. Wang, and M. Gong. A performance study on different cost aggregation approaches used in real-time stereo matching. *IJCV*, 75(2):283–296, 2007. 3, 7

[5] M. Gong and Y.-H. Yang. Near real-time reliable stereo matching using programmable graphics hardware. In *CVPR*, 2005. 6

[6] H. Hirschmüller and D. Scharstein. Evaluation of cost functions for stereo matching. In *CVPR*, Jun 2007. 1, 6

[7] H. Hirschmüller and D. Scharstein. Evaluation of stereo matching costs on images with radiometric differences. *IEEE TPAMI*, 31(9):1582–1599, Sep 2009. 7

[8] A. Hogue, A. German, and M. Jenkin. Underwater environment reconstruction using stereo and inertial data. In *IEEE SMC*, pages 2372–2377, 2007. 1

[9] G. Klein and D. Murray. Parallel tracking and mapping for small ar workspaces. In *ISMAR*, pages 1–10, 2007. 1

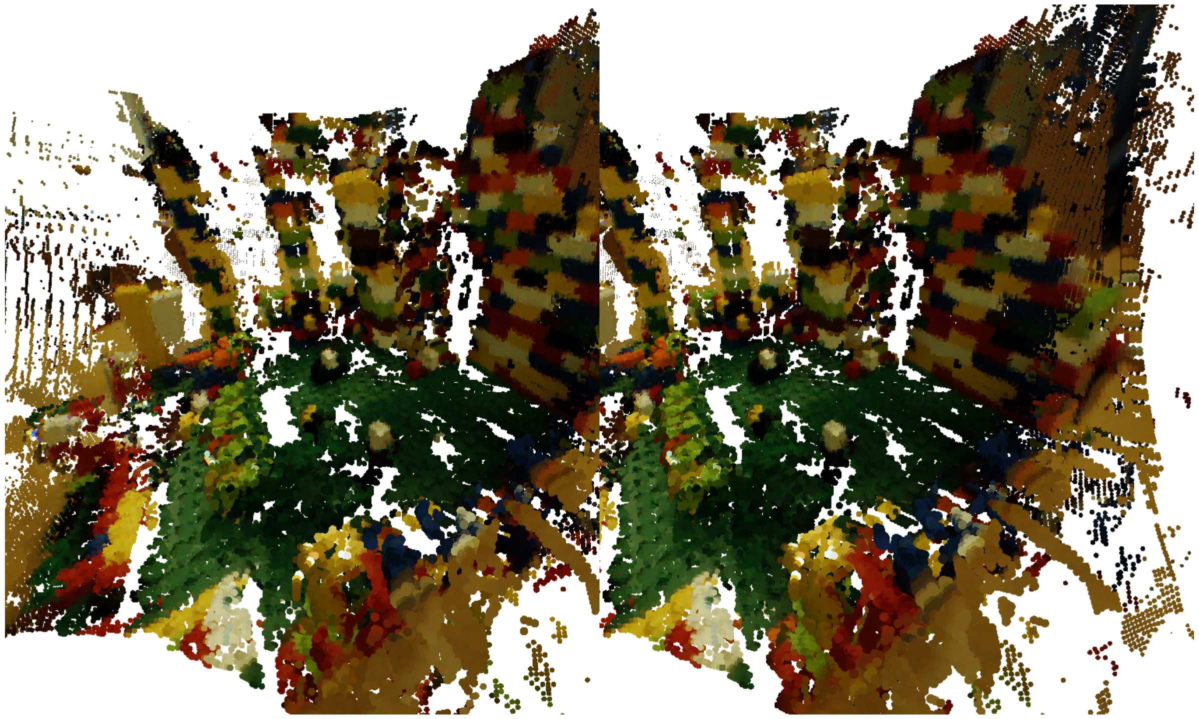

Figure 2. A screenshot from the real-time monocular 3D reconstruction software. The picture is intended to be viewed in colour and with crossed eyes for the stereo effect (the left eye seeing the right half and vice versa). This is easiest to achieve by looking at your finger tip and adjusting the distance until you perceive three images—the middle one will be 3D.

[10] J. Montiel, J. Civera, and A. Davison. Unified inverse depth parametrization for monocular slam. In *RSS*, 2006. 1

[11] A. Ogale and Y. Aloimonos. Robust contrast invariant stereo correspondence. *ICRA*, pages 819–824, 2005. 6

[12] A. Ogale and Y. Aloimonos. Shape and the stereo correspondence problem. *IJCV*, 65(3):147–162, 2005. 1

[13] A. Ogale and Y. Aloimonos. A roadmap to the integration of early visual modules. *IJCV*, 72(1):9–25, Apr 2007. 1

[14] A. Ogale and J. Domke. Openvis3d, open source 3d vision library. Website, 2007. Referred Dec 2nd, 2009. `http://code.google.com/p/openvis3d/`. 6

[15] P. Paalanen, V. Kyrki, and J.-K. Kamarainen. Towards monocular on-line 3d reconstruction. In *ECCVW*, 2008. 1, 7

[16] M. Pollefeys, D. Nistér, et al. Detailed real-time urban 3d reconstruction from video. *IJCV*, 78(2-3):143–167, 2008. 1

[17] M. Pollefeys, L. Van Gool, et al. Visual modeling with a hand-held camera. *IJCV*, 59(3):207–232, Sep 2004. 1

[18] S. Rusinkiewicz, O. Hall-Holt, and M. Levoy. Real-time 3d model acquisition. *ACM Trans. Graph.*, 21(3):438–446, 2002. 1

[19] D. Scharstein and A. Blasiak. Middlebury stereo evaluation - version 2. Website. Referred Oct 19, 2009. `http://vision.middlebury.edu/stereo/`. 1, 5, 6, 7

[20] D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *IJCV*, 47(1):7–42, Apr 2002. 1, 6

[21] D. Scharstein and R. Szeliski. High-accuracy stereo depth maps using structured light. In *CVPR*, volume 1, pages 195–202, Madison, WI, Jun 2003. 1, 6

[22] S. Se and P. Jasiobedzki. Photo-realistic 3d model reconstruction. In *ICRA*, pages 3076–3082, May 2006. 1

[23] S. Seitz, B. Curless, et al. A comparison and evaluation of multi-view stereo reconstruction algorithms. In *CVPR*, 2006. 1

[24] C. Strecha, W. von Hansen, et al. On benchmarking camera calibration and multi-view stereo for high resolution imagery. In *CVPR*, 2008. 1

[25] F. Tombari, S. Mattoccia, et al. Near real-time stereo based on effective cost aggeration. In *ICPR*, 2008. 6

[26] O. Veksler. Stereo correspondence by dynamic programming on a tree. In *CVPR*, 2005. 6

[27] L. Wang, M. Liao, et al. High-quality real-time stereo using adaptive cost aggregation and dynamic programming. In *3DPVT*, 2006. 6

[28] Q. Yang, C. Engels, and A. Akbarzadeh. Near real-time stereo for weakly-testured scenes. In *BMVC*, 2008. 6

[29] Q. Yang, L. Wang, et al. Real-time global stereo matching using hierarchial belief propagation. In *BMVC*, 2006. 6

[30] R. Yang and M. Pollefeys. Multi-resolution real-time stereo on commodity graphics hardware. In *CVPR*, 2003. 2

[31] R. Yang and M. Pollefeys. A versatile stereo implementation on commodity graphics hardware. *Real-Time Imaging*, 11:7–18, 2005. 2

[32] K. Zhang, J. Lu, et al. Real-time accurate stereo with bitwise fast voting on CUDA. In *ICCVW*, 2009. 6