Automatic Pose Estimation for Range Images on the GPU

Marcel Germann
Computer Graphics Laboratory
Swiss Federal Institute of Technology (ETH)
Zurich, Switzerland
masi@student.ethz.ch

Michael D. Breitenstein
Computer Vision Laboratory
Swiss Federal Institute of Technology (ETH)
Zurich, Switzerland
breitenstein@vision.ee.ethz.ch

In Kyu Park
Inha University
Incheon, Korea
pik@inha.ac.kr

Hanspeter Pfister
Mitsubishi Electric Research Laboratories
Cambridge, MA, USA
pfister@merl.com

Abstract

Object pose (location and orientation) estimation is a common task in many computer vision applications. Although many methods exist, most algorithms need manual initialization and lack robustness to illumination variation, appearance change, and partial occlusions. We propose a fast method for automatic pose estimation without manual initialization based on shape matching of a 3D model to a range image of the scene. We developed a new error function to compare the input range image to pre-computed range maps of the 3D model. We use the tremendous data-parallel processing performance of modern graphics hardware to evaluate and minimize the error function on many range images in parallel. Our algorithm is simple and accurately estimates the pose of partially occluded objects in cluttered scenes in about one second.

1. Introduction

A common task in computer vision applications is to estimate the pose (location and orientation) of objects. Pose estimation in scenes with clutter (due to unwanted objects and noise) and occlusions (due to multiple overlapping objects) is challenging. Furthermore, pose estimation in 2D images and video is sensitive to illumination, shadows, and lack of features (e.g., objects without texture). Pose estimation from range images – where each pixel contains an estimate of the distance to the closest object – does not suffer from these limitations. Range images can be robustly acquired with active light systems [2, 1]. If a database of 3D models of objects is available, one can use model-based techniques, where the 3D model of the object is matched to the range image of the scene. Model-based pose estimation has been used in applications such as object recognition, object tracking, robot navigation, and motion detection.

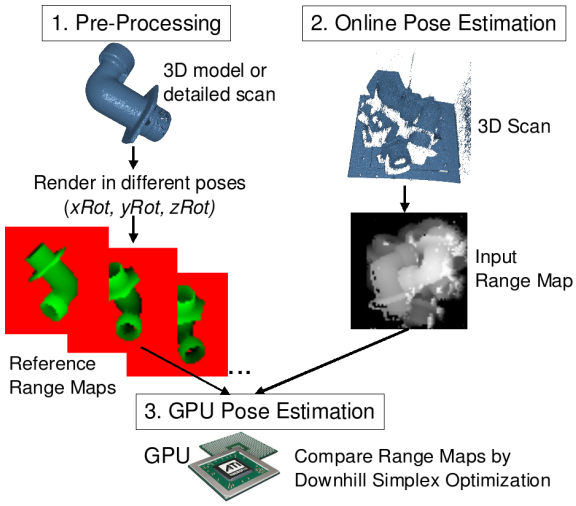

Figure 1. Overview of the algorithm: During pre-processing, a 3D model is rendered in different poses and stored as reference range maps. During online pose estimation, the reference range maps are compared to the input range image using a novel error function. To find the match with the least error we use a parallel implementation of the downhill simplex method on the GPU.

In this paper we present a novel model-based pose estimation algorithm for range images that runs entirely on modern Graphics Processing Units (GPUs). The massive

data-parallel processing on GPUs makes our method over 30 times faster than a comparable CPU implementation. Our method works does not require manual initialization and accurately computes object poses for synthetic or laser scan data in about one second.

Figure 1 shows an overview of our method. We assume that we estimate the pose of a known, rigid (reference) object. In a pre-processing step, we use a 3D model or detailed scan of the object and render it in different poses. Each pose is stored as a *reference range map* in texture memory. This task has to be performed only once per reference object.

During online pose estimation, we acquire a 3D scan of the scene using an active light method (in our case a laser range scan). We smooth the 3D scan on the GPU using a median filter to compute the *input range map*. The task is now to find the best match between reference range maps and input range map through error minimization by pairwise comparisons. The best matching reference range map and its translation with respect to the input range map yield our pose estimation. We devised a novel error function (see Section 4) that uses the range values and Euclidean distance maps. The error function can be evaluated per pixel, which makes it suitable for efficient processing in GPU fragment shaders[1]. To efficiently minimize the error we developed a novel data-parallel version of the downhill simplex algorithm [19] that runs entirely on the GPU (see Section 5).

The novel contributions of our work are a simple error metric to compare the alignment of two range maps, a method to compute signed Euclidean distance transforms of images on the GPU, and a data-parallel implementation of the downhill simplex algorithm on the GPU. We present an efficient implementation of model-based pose estimation for range images that runs entirely on the GPU, and we evaluate the performance and robustness on various synthetic and real-world input scenes with clutter and occlusions.

2. Related Work

The main challenge in pose estimation is invariance to partial occlusions, cluttered scenes, and large pose variations. Approaches based on 2D images and video generally do not overcome these problems due to their dependency on appearance and sensitivity to illumination, shadows, and scale. Among the most successful attempts are methods based on global appearance (e.g., [15]), and methods based on local (2D) features (e.g., [24, 21]). Unfortunately, these methods usually require a large number of labeled training examples.

Recently, model-based surface matching techniques using a 3D model have become popular due to the decreasing cost of 3D scanners. The most popular method for aligning 3D models and range images is the Iterative Closest Point (ICP) algorithm [4] and its variations (e.g., [6, 22, 8, 9]). However, as stated by Rusinkiewicz et al. [23], ICP requires a sufficiently good initial pose estimate to avoid being stalled by local minima. Our method does not require any initial guess and consistently finds the global optimum. Shang et al. [25] use the Bounded Hough Transform (BHT) [12] to compute an initial pose estimate before running ICP. However, their method was developed for object tracking, whereas our method is applicable for pose estimation from a single range image.

Geometric hashing [16] is an efficient method to establish multi-view correspondence and object pose. However, building the hash table is time consuming, and the matching process is sensitive to image resolution and surface sampling. A large class of methods use a deformable (morphable) 3D model and minimize a cost term such that the model projections match to input images (e.g., [14, 5]). Optimizing many parameters while projecting the 3D model is inefficient and also requires an initial pose guess.

Another approach is to match 3D features (or shape descriptors) to range images. Dorai et al. [7] use curvature features by calculating principal curvatures. This requires the surface to be smooth and twice differentiable and thus is sensitive to noise. Moreover, occluded objects cannot be handled. Johnson et al. [13] introduced "spin-image" surface signatures for 3D registration. This yields good results with cluttered scenes and occluded objects. But their method is time-consuming, sensitive to image resolution, and might lead to ambiguous matches. Mian et al. [18] build a multidimensional table representation (referred to as tensors) from multiple unordered range images. They use a hash-table voting scheme to match the tensor to objects in a scene. Compared to spin-images, they report a higher success rate, but the method requires high-resolution geometry and has a runtime of several minutes. Similar to our approach, Greenspan [11] precomputes range maps of the model, and uses a tree structure and Geometric Probing. However, the computation time depends on the object size and he reports at least four seconds for reliable results.

There has been a substantial amount of work to apply the processing power of GPUs to non-graphics applications. A good collection of general-purpose GPU processing (GPGPU) can be found in [10]. To our knowledge there is no previous work on GPU pose estimation.

3 Range Input Processing

Our method starts with an input 3D scan of the scene and a 3D model of the object. Both are first being orthogonally projected into the input range map and the reference range map, respectively. We choose the viewports of these orthogonal projections, the viewing frustum, and the image resolu-

[1] A fragment shader is a piece of user-programmable GPU code that is executed for multiple pixels in parallel.

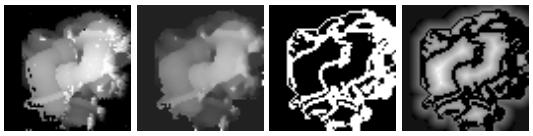

Figure 2. Processing of the input 3D scan: a) original input 3D scan; b) range map after smoothing by a median filter; c) result of the simple edge detection algorithm; d) signed Euclidean Distance Transform.

Algorithm 1 Signed EDT

$coord(p)$ = coordinates of the closest edge e found to far
$value(p)$ = signed distance value to e
Require: $value(b) = -(m+1) \quad \forall b \in$ background
Require: $value(f) = +(m+1) \quad \forall f \in$ foreground
Require: $value(e) = 0 \quad \forall e \in$ edge
Require: $coord(p) = (x_p, y_p) \quad \forall p \in$ image
for all iterations m **do**
  for all pixels p **do**
    for all direct neighbors n of p **do**
      if $distance(p, coord(n)) < |value(p)|$ **then**
        $value(p) = signed\_distance(p, coord(n))$
        $coord(p) = coord(n)$

tion to be the same. The scale factor of physical units (mm) of the scanner to unit distance of the 3D model is readily available from scanner manufacturers. Consequently, the scale of the objects in the reference and input range maps is identical after projection.

The input 3D scan is smoothed by a median filter with a 3×3 mask implemented as a fragment shader (see a) and b) in Figure 2). In a second rendering pass, a simple heuristic is used to detect object edges by comparing range values of neighboring pixels. If the range difference exceeds 4% of the image width, the pixel is marked as an edge (see c) in Figure 2). The potential edge pixels are marked with a bit for consecutive processing.

Next, we compute the *signed Euclidean Distance Transform (EDT)* that assigns to each pixel the signed distance to the edge pixels by adapting an approach called *ping-pong rendering* [20]. It uses two RGBA color-textures and consecutively switches their role as rendering source and target, respectively. In our GPU implementation we use a 32-bit floating point format for each color channel. The values in the first two color channels represent the coordinates of the closest edge pixel found so far, the third channel stores the signed distance, and the fourth channel indicates whether an edge pixel is already found.

Algorithm 1 shows the pseudo-code of our EDT algorithm. The parameter m determines the number of iterations. The distance values are initialized to $-(m+1)$ for background pixels (i.e., range value $= 0$), to $m+1$ for foreground pixels (i.e., range value $\neq 0$), and to 0 for all edge pixels. The first two color channels are initialized to the pixel coordinates. In each iteration, the distance value of each pixel is compared to the values of its eight direct neighbors. The distance value and coordinates of the recent pixel p are updated if the distance from p to the edge pixel saved in a neighboring pixel n is smaller than the value saved at p. This information is iteratively propagated over the entire image at each step, as shown in Figure 3.

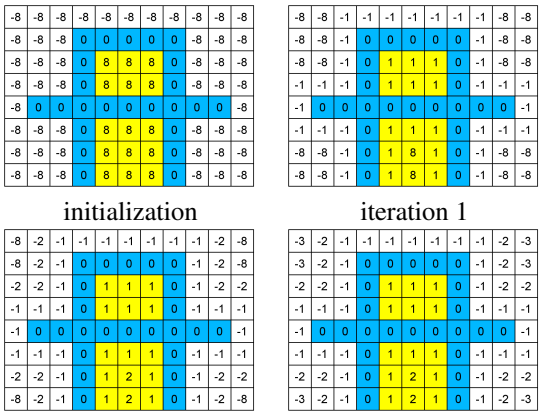

Figure 3. Computation of the signed EDT on the GPU: The values are distances to closest edge pixels. The top left image shows the initialization step. The other images show the first three iterations.

This method assumes square images. The number of iterations m corresponds to the maximum distance of any pixel to its closest edge. For full convergence one chooses m to be half the width of the image. However, to speed up the algorithm we make use of the fact that the distance of each pixel to an object edge is typically much smaller. For our algorithm we also do not need the exact EDT and are willing to use an approximation. We empirically found that $m = 7$ is sufficient for the 64×64 pixel images we are using.

4 Error Function

Equation 1 shows the error function we use to compare a reference range map R and an input range map I:

$$\epsilon(I, R, x, y, z) = \frac{1}{N_{\text{cover}}} \sum_{u,v} \epsilon_{\text{cover}}(u, v, x, y) + \lambda \frac{1}{N_{\text{range}}} \sum_{u,v} \epsilon_{\text{range}}(u, v, x, y, z). \quad (1)$$

It consists of the *Cover Error Term* $\epsilon_{\text{cover}}(u, v, x, y)$ (see Section 4.1) and the *Depth Error Term* $\epsilon_{\text{range}}(u, v, x, y, z)$ (see Section 4.2) that are evaluated at each range map pixel (u, v). The purpose of ϵ_{cover} is to measure the errors in the alignments of the silhouettes, inspired by Lee et al. [17]. The Depth Error Term measures differences in the topology of the aligned objects in 2.5D similar to ICP in 3D.

The translation values (x, y, z) of R determine its position with respect to I. The two error terms are weighted by λ and summed up over all image pixels (u, v). In our experiments we empirically found that $\lambda = 10$ works best for all scenes and objects we tested. The normalization factors N_{cover} and N_{range} – discussed in the following sections – make the error independent of object and image size. The error is minimal if R is perfectly aligned to a – possibly partially occluded – object in I.

4.1 Cover Error Term

The cover error of a pixel (u, v) of the input range map I and a pixel in the reference range map R – translated by (x, y) – is:

$$\epsilon_{\text{cover}}(u, v, x, y) = \begin{cases} |EDT_I(u, v) - EDT_R(u + x, v + y)| \\ \quad \text{if } EDT_R(u + x, v + y) \geq 0 \\ 0 \quad \text{otherwise.} \end{cases} \tag{2}$$

The cover error term is minimal if the silhouettes of the objects in I and R match perfectly. Note that only non-background pixels of R with positive range values are considered. The cover error normalization factor is:

$$N_{\text{cover}} = \| \{(u, v) | EDT_R(u + x, v + y) \geq 0\} \|. \tag{3}$$

4.2 Depth Error Term

The range error term compares the range values of all foreground pixels in I and R that overlap, thus:

$$\epsilon_{\text{range}}(u, v, x, y, z) = \begin{cases} |z_I(u, v) - (z_R(u + x, v + y) + z)| \\ \quad \text{if } EDT_I(u, v) \geq 0 \wedge EDT_R(u + x, v + y) \geq 0 \\ 0 \quad \text{otherwise.} \end{cases} \tag{4}$$

Note that R is translated by (x, y) and that z is added to all range values of R. The range error normalization factor is:

$$N_{\text{range}} = \| \{(u, v) | EDT_I(u, v) \geq 0 \wedge EDT_R(u + x, v + y) \geq 0\} \|. \tag{5}$$

4.3 Implementation on the GPU

The error function in Equation 1 is computed using fragment shaders. Due to the parallel processing on the GPU these pixel-wise comparisons are very fast, especially for low resolution images.

In a first step, the input range map I and the reference range map R are loaded to the GPU. A fragment shader program computes the error terms $\epsilon_{\text{cover}}(u, v, x, y)$ and $\epsilon_{\text{range}}(u, v, x, y, z)$ for each pixel. Two binary bits n_{cover} and n_{range} used for the normalization factors of Equations 3 and 5 indicate whether an error value was computed. All values are stored in the color channels of a 32-bit texture S.

In a second step, the error values are summed up over all pixels of S and the final error is computed. Because this summation has to be done for each optimization iteration (see Section 5) we implemented it on the GPU using ping-pong rendering between S and a temporary texture T.

Beginning with a step size $s = 1$, one color channel of pixel (u, v) stores the sum of the values of the pixels (u, v), $(u + s, v)$, $(u + s, v + s)$, $(u, v + s)$ by rendering from S to T. Subsequently, s is doubled in each iteration, and S and T are exchanged, as illustrated in Figure 4. The final result of the error function is stored at pixel $(0, 0)$ after $s = log(l)$ steps, where l is the image width in pixels. This algorithm is very efficient assuming we have square images.

5 Parallel Optimization Framework

The goal of the error optimization is to find the parameters $(\hat{x}, \hat{y}, \hat{z}, \hat{\theta}, \hat{\phi}, \hat{\sigma})$ that globally minimize the error between the input and reference range maps. Thus, we are solving the following 6-DOF optimization problem:

$$(\hat{x}, \hat{y}, \hat{z}, \hat{\theta}, \hat{\phi}, \hat{\sigma}) = \underbrace{\arg\min_{\theta, \phi, \sigma} \left(\underbrace{\min_{x, y, z} \epsilon(I, R_{\theta, \phi, \sigma}, x, y, z)}_{\text{step 1}} \right)}_{\text{step 2}} \tag{6}$$

$R_{\theta, \phi, \sigma}$ is a reference range map of the 3D model rendered with rotation angles (θ, ϕ, σ). Step 1 computes the error between each reference range map and the input range map using the downhill simplex method [19] for the translation values (x, y, z). Step 2 selects the reference range map $R_{\theta, \phi, \sigma}$ with the lowest global error. The result is the estimated pose $(\hat{x}, \hat{y}, \hat{z}, \hat{\theta}, \hat{\phi}, \hat{\sigma})$.

During pre-processing, we compute one large texture, the *reference texture matrix*, to store all reference range maps. This matrix has to be prepared only once per 3D model (see Figure 5). The more reference range maps we

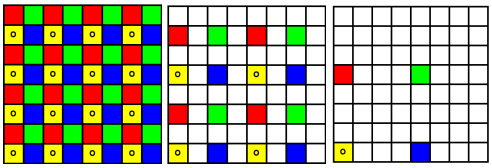

Figure 4. Computation of the final error on the GPU: In each iteration k, information at the current pixel (yellow with circle) is collected from the upper (red), upper right (green), and right (blue) neighbor at distance $s = 2^k$.

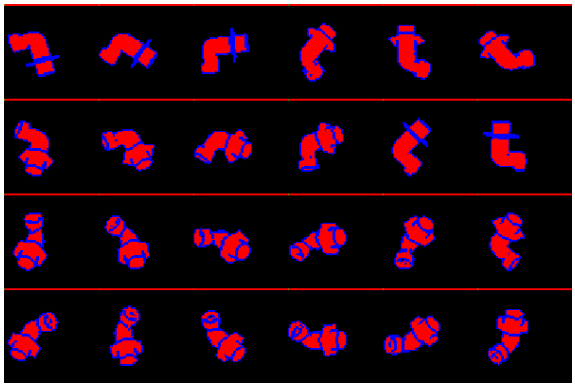

Figure 5. Part of a reference texture matrix with 24 range maps. The topmost row of each range image (in red) is used to store the parameters of the downhill simplex algorithm.

store, the better our angular pose estimate and the slower the algorithm. The texture memory size of the GPU also imposes restrictions on the size of the reference texture map. For example, it would be prohibitive to store all reference range maps for 3-DOF rotations with one degree increments.

To address this issue we use a simple greedy algorithm. We render the object using orthonormal projection and store the z-buffer as a range map. Then we rotate the object by (θ, ϕ, σ) with very small rotation increments (e.g., 5 degrees). For each new reference range map we compute the error according to Eq. 1 with respect to all previously stored range maps. If the error is larger than a user-defined threshold we add the range map to the reference texture matrix. Since we do not replace but only add range maps we have to run the algorithm a few times with increasingly larger thresholds until we can fit all range maps into the reference texture matrix. This algorithm could certainly be improved (e.g., by dynamic programming).

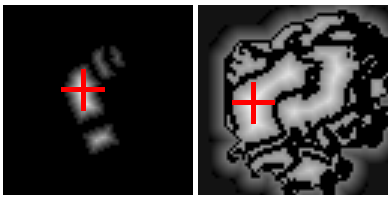

Figure 6. Example of a starting point for a) the reference range map R and b) the input range map I. The initial translation parameters are found by aligning the two starting points.

5.1 Initial Parameters

The number of iteration steps for convergence of the downhill simplex procedure can be drastically reduced by choosing adequate initial parameters. To compute good initial translation parameters (x_0, y_0, z_0) we try to find a pixel in R and I, respectively, that roughly corresponds to the center of gravity of the object (see Figure 6). If I contains multiple objects we choose the one that is closest to the camera, i.e., the one with largest z value. For each range map, we initialize the center of gravity (u, v) to the first pixel and iterate over all pixels (r, s). We update (u, v) to the new pixel position (r, s) if:

$$0.5 \cdot EDT(u, v) + (z(r, s) - z(u, v)) \geq 0.5 \cdot EDT(r, s). \quad (7)$$

The EDT terms force the result to be near the center of an object, and the z term forces it to be close to the camera.

The initial translation parameters are then simply defined by the alignment where these points from both range maps fall together in x, y and z. In our experiments we found that without a good initial guess (e.g., by just aligning the range maps) it took about 30 to 40 iterations for the downhill simplex algorithm to converge. With this simple method we could reduce the number of iterations to 15, which resulted in a speedup of a factor of two.

5.2 Data-Parallel Downhill Simplex on the GPU

To parallelize the downhill simplex method for the GPU, an additional scanline is added to each range map in the reference texture matrix (see Figure 5). We use it to store the parameters of the downhill simplex algorithm and the error values in different color channels.

The vertices of the simplex are initialized to (x_0, y_0, z_0), $(x_0 + d, y_0, z_0)$, $(x_0, y_0 + d, z_0)$ and $(x_0, y_0, z_0 + d)$, where x_0, y_0 and z_0 are the initial parameters described in Section 5.1. We empirically determined that an adequate value for the optimal step size d is 5% of the image width.

	Median	Edge	EDT	Pose	Total
GPU	0.0007	0.0003	0.0018	1.0247	1.0278
CPU	0.0008	0.0002	0.0141	31.8236	31.8393

Table 1. Average times (in seconds) for different parts of the algorithm on GPU and CPU.

The optimization procedure is implemented using three fragment shader programs. The first shader implements the actual downhill simplex algorithm as described by Nelder et al. [19]. The second shader computes the error terms of Equations 2 and 4, and the third shader computes the final error value as described in Section 4.3. This loop is executed for each evaluation of the error function in the the downhill simplex algorithm. Finally, the topmost scanlines of all reference range maps are transferred to the CPU. The parameters $(\hat{x}, \hat{y}, \hat{z}, \hat{\theta}, \hat{\phi}, \hat{\sigma})$ of the reference range map with the lowest error correspond to the estimated pose.

6 Results and Discussion

We tested our algorithm with synthetic data as well as real laser scans. All results were computed using a PC with 3.2 GHz Intel dual-core CPU and nVIDIA GeForce 8800 GTX graphics card with 128 shader processors. We empirically found that the optimal number of parallel downhill simplex iterations is 15 for a range maps with 64×64 pixels.

The execution time – from input of the raw 3D scan to output of the estimated pose – was *0.64 seconds* for 1024 reference range maps (one matrix) and *1.03 seconds* for 2048 reference range maps (two matrices). Using 2048 reference range maps (two matrices) improved pose accuracy. This corresponds to over 29,000 range map comparisons per second. The running time is independent of scene and reference range map complexity. The input processing (edge detection, EDT, and starting point) takes less than 0.1% of the overall time. The rest of the time is used by the data-parallel downhill simplex method. For one iteration step, around 30% of the time is spent on the evaluation of the error function, 50% on collecting the error scores, and 20% on the actual downhill steps. Running the same amount of range map comparisons on the CPU takes 31.8 seconds — about 30 times slower. Table 1 shows a detailed performance comparison. The times for median filtering and edge detection are very similar for GPU and CPU because there is not sufficient parallelism during processing of one 64×64 image. However, computing on the GPU saves time because the image has to be uploaded to the graphics card only once.

To measure the accuracy of the pose estimation we first used synthetic data as ground truth. The 3D models for our experiments are shown in Figure 7. Using 3D modeling software we built 21 test scenes with clutter and partial occlusions, seven for each of the three models (see Figure 8). As real-world test data we used 50 laser range scans – acquired from different viewpoints – of four different bins with bronze pipes. As the example in Figure 9 shows, the laser scans are quite noisy.

Figure 7. The synthetic test models.

Dataset	Avg. / Max. Translation	Avg. / Max. Angular	Avg. / Max. Hausdorff
Pipe	$3.3px/3.8px$	$5.22°/9.22°$	$1.8\%/6.8\%$
Dino	$3.7px/4.9px$	$5.82°/9.10°$	$2.4\%/7.9\%$
Drill	$3.4px/5.1px$	$7.01°/12.77°$	$2.4\%/7.7\%$
Laser scan	N/A	N/A	$2.0\%/8.9\%$

Table 2. Average and maximum pose errors and Hausdorff distances. Translational error is in pixels (px), rotational error in degrees, and Hausdorff distance is in percent of the physical image width. There are no values for the pose errors for the laser scans since ground truth is not available.

We use two different error metrics to evaluate the pose estimation (see table 2). *Pose error* refers to the difference between the computed pose and ground truth. Since there was no ground truth available for the laser scans we also compute the *one-sided Hausdorff distance*. For each pixel in the reference range map we find the closest pixel of the input range map and compute their Euclidean distance. We then take the average and the maximum of these distances, both averaged over all test samples. Note that – after pose estimation – those range maps are overlapping but not necessarily axis aligned. The Hausdorff distances are normalized by the width of the range map. In other words, a Hausdorff distance of one corresponds to the width of the image in physical units. We report them as percentage of physical image width.

In all our experiments, the algorithm correctly detected the pose of the object closest to the camera without manual initialization, even in scenes with noise and partial occlusions. Table 2 shows the average and maximum errors over all experiments using 2048 reference range maps. The average and maximum angular pose error is about 5 and 10 degrees, respectively. As expected, this is about half of the angular difference between reference range maps, which is 22 degrees on average. For 1024 reference range maps, the angular difference is about 28 degrees, and the average error

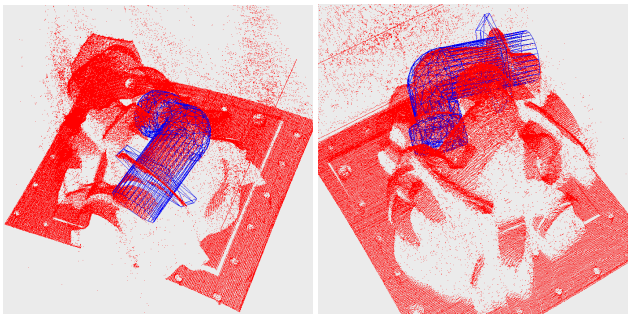

Figure 9. Two different views of the same laser scan (red points). The blue mesh is the result of the pose estimation. This experiment resulted in the worst Hausdorff distance, but with still acceptable pose error.

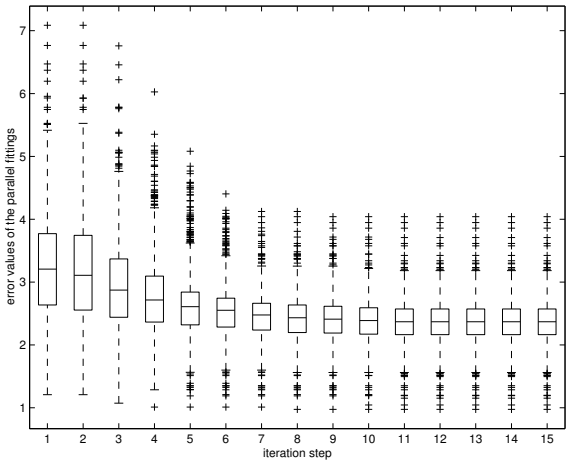

Figure 10. Mean, maximum, minimum, and standard deviation of the error for all 15 iterations of the downhill simplex method. Note the rapid convergence.

increases to about 14 degrees. The translation and Hausdorff distance errors are very small. For the laser range scanner, where we know the scaling from virtual to physical units, the average and maximum Hausdorff distances (averaged over all test samples) correspond to 2.8 mm and 12.46 mm, respectively.

Figure 8 shows some example pose fitting results for the synthetic models. Our method currently can only handle one 3D model in the scene. However, the algorithm could be extended to estimate the pose of n objects by fitting n different models to the scene and choose the one with the overall minimum error. To test this hypothesis we ran an experiment with a mixed scene with all the models (see Figure 8, right). We had to ensure that the model picked by the initial guess corresponded to the one we stored in the reference texture matrix. Once this was the case, the algorithm was able to correctly fit the model to the scene.

Figure 9 shows two views of the same laser scan scene. The algorithm yielded for this experiment the maximum Hausdorff distance (21.7mm). As the reference model (in blue) indicates, this result is acceptable for many applications. Running ICP [4] on our test scenes did not converge because ICP requires a good initial pose guess. In contrast, our method is exploring the complete parameter space *in parallel* on the GPU and always finds the global optimum.

Figure 10 shows a boxplot of the mean, maximum, minimum, and standard deviation of the errors for all 15 downhill simplex iteration steps of a laser-scan test scene with 2048 reference range maps. Note the rapid convergence after only 11 iterations. By looking at the distribution of the error values along each error bar we note the rapid falloff towards the tails (min and max values).

7 Conclusions and Future Work

We presented a very fast method for pose estimation for range images that exploits the parallelism of modern GPUs. We demonstrated its capability to automatically locate objects in complex scenes and to correctly estimate their pose without initial guess. Our method is not affected by local minima since we compute the errors for all reference range maps and then choose the globally best one. For this application this simple, brute-force approach is very successful.

The number of range maps in the reference texture matrix has an impact on the accuracy and on performance. This number is limited by the amount of memory on the graphics card. For 2048 reference depth maps we currently use 256 MB of texture memory. One could devise a better algorithm to find the minimal set of reference poses. For example, the user could decide that certain poses (e.g., frontal views) are more likely and therefore should be represented more often in the reference texture matrix.

Instead of using pre-computation, we could also compute the reference range maps online by rendering a low-resolution version of the 3D model. This would improve the accuracy but reduce the speed. The best solution is probably a hybrid method that uses pre-computed reference textures for initial pose estimation and then online rendering and pose fitting to improve accuracy.

Further improvements could be achieved by exploiting nVIDIA's Compute Unified Device Architecture (CUDA) compiler [3] that allows to run many C-style programs simultaneously on the GPU. In this work we focused entirely on range images. If greyscale or color values are available

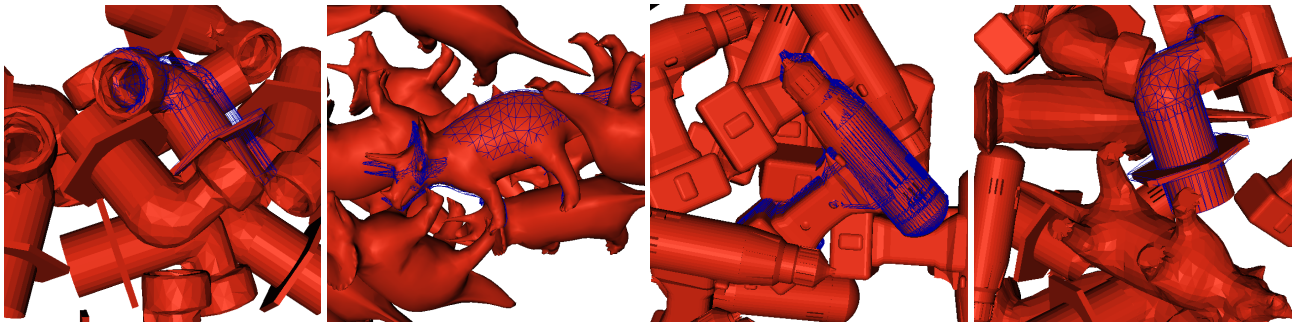

Figure 8. Example pose estimation results for the synthetic data. The reference range maps with minimum error (blue) are overlaid on the input range maps (red).

then we could take the luminance or color gradient into account in the error function. Other simple feature detectors could be implemented on the GPU as well, although one needs to investigate what benefits the added complexity has for performance. We also would like to extend this work to non-rigid objects, such as articulated models (e.g., chain-links or human bodies) or deformable objects (e.g., faces).

8 Acknowledgments

We would like to thank the following persons for support and many fruitful discussions: H. Okuda, Dr. K. Sumi, K. Tanaka of Mitsubishi Electric; Dr. M. Jones, Dr. P. Beardsley, Dr. J. Katz, Dr. K. Kojima, MERL; S. Heinzle, Dr. B. Leibe, Prof. M. Gross, Prof. L. van Gool, ETH Zurich. This project has been funded by Mitsubishi Electric. M. Breitenstein is partially funded by EU project HERMES (IST-027110).

References

[1] 3Q Technologies Ltd. http://www.3q.org.

[2] Cyberware Inc. http://www.cyberware.com.

[3] Nvidia CUDA. http://developer.nvidia.com/object/cuda.html.

[4] P. Besl and N. McKay. A method for registration of 3d shapes. *PAMI*, 1992.

[5] V. Blanz and T. Vetter. A morphable model for the synthesis of 3d faces. *SIGGRAPH*, 1999.

[6] Y. Chen and G. Medioni. Object modeling by registration of multiple range images. *Robotics and Automation*, 1991.

[7] C. Dorai and A. K. Jain. Cosmos - a representation scheme for 3d free-form objects. *PAMI*, 19(10):1115–1130, 1997.

[8] N. Gelfand, L. Ikemoto, S. Rusinkiewicz, and M. Levoy. Geometrically stable sampling for the icp algorithm. *3DIM*, 2003.

[9] N. Gelfand, N. Mitra, L. Guibas, and H. Pottmann. Robust global registration. *Eurographics Symposium on Geometry Processing*, 2005.

[10] GPGPU. http://www.gpgpu.org.

[11] M. Greenspan. Geometric probing of dense range data. *PAMI*, 24(4):495–508, 2002.

[12] M. Greenspan, L. Shang, and P. Jasiobedzki. Efficient tracking with the bounded hough transform. *CVPR*, 2004.

[13] A. Johnson and M. Hebert. Using spin images for efficient object recognition in cluttered 3d scenes. *PAMI*, 21(5):433–449, 1999.

[14] M. J. Jones and T. Poggio. Multidimensional morphable models: A framework for representing and matching object classes. *IJCV*, 29(2):107–131, 1998.

[15] M. J. Jones and P. Viola. Fast multi-view face detection. *CVPR*, 2003.

[16] Y. Lamdan and H. Wolfson. Geometric hashing: A general and efficient model-based recognition sceme. *IJCV*, pages 238–249, 1988.

[17] J. Lee, B. Moghaddam, H. Pfister, and R. Machiraju. Finding optimal views for 3d face shape modeling. *Automatic Face and Gesture Recognition*, pages 31–36, 2004.

[18] A. Mian, M. Bennamoun, and R. Owens. Three-dimensional model-based object recognition and segmentation in cluttered scenes. *PAMI*, 28(12):1584–1601, 2006.

[19] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965.

[20] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[21] F. Rothganger, S. Lazebnik, C. S. J., and Ponce. 3d object modeling and recognition using affine-invariant patches and multi-view spatial constraints. *CVPR*, 2003.

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

[23] S. Rusinkiewicz and M. Levoy. Efficient variants of the icp algorithm. *3DIM*, 2001.

[24] C. Schmid and R. Mohr. Combining greyvalue invariants with local constraints for object recognition. *CVPR*, 1996.

[25] L. Shang, P. Jasiobedzki, and M. Greenspan. Discrete pose space estimation to improve icp-based tracking. *3DIM*, 2005.