A Matching Shader Technique for Model-Based Tracking

Martin Schumann, Jan Hoppenheit, Stefan Müller
University of Koblenz-Landau
Institute of Computational Visualistics
56070 Koblenz, Germany
{schumi, silver, stefanm}@uni-koblenz.de

ABSTRACT

We present a line feature matching method for model-based camera pose tracking. It uses the GPU for computing the best corresponding image line match to the edges of a given 3D model on a pixel basis. Further, knowledge about the model is considered to improve the matching process and to define quality criteria for match selection. Each edge is rendered several times with image offsets from the last estimated position of the model. The shader counts the number of pixels in an underlying canny-filtered camera input image. Returning the best fit by pixel count can be done applying occlusion queries. A speed-up can be achieved using a more elaborate shader with texture read-back reducing the number of rendering passes. The matching shader is not limited to work with lines and can be extended to other structures as well.

Keywords

Model-Based Camera Pose Tracking, Line Feature Matching, GPU Shader.

1 INTRODUCTION

Camera pose tracking is the process of estimating the viewing position and orientation of a camera. This can be performed using a model of the environment represented by 3D data available from a modeling process or created online. Using a model leads to more stable tracking without drift occurrence, as it is the case for frame-to-frame tracking. Further the model serves as an absolute reference for initialization.

The pose estimation problem is based upon establishing 2D-3D correspondences between features of the model and features in the camera image that may be points, lines or higher structures. The aim is to minimize the distance between projected 3D features and their 2D correspondences in the camera image. Establishing these correspondences is crucial for estimating a good camera pose. False matches lead to shifting in the pose, jittering or even loss of the tracking.

Tracking on CAD models was realized by [Com03], and respectable success in combination of edges with texture information could be demonstrated by [Vac04]. Current research is focused on SLAM (Simultaneous Localization and Mapping) algorithms [Kle07], where feature cloud maps are reconstructed from the visible surroundings.

In the approach of analysis-by-synthesis even further knowledge about the model is used for the tracking process. Beginning from an initially estimated pose or the pose of the given 3D model in the last image, a rendered image or a structure of features is synthesized together with a collection of additional information available from rendering process or from global knowledge. In the analysis step these are compared to a real camera image to estimate the current camera pose. In [Wue07] they use depth and normal information to derive the line trait of the model. The work of [Sch09] analyzes similarity-based and feature-based methods for comparing synthetic and real image and [Bra11] simulate the lightning conditions to improve tracking.

We present a method for matching model edges to lines in the camera image using the GPU. It uses the model knowledge to define the quality of the matches for match selection. In our approach we work with straight lines but the technique is not limited to this type of feature and can be used for other structures as well.

2 RELATED WORK

The problem of matching and registration of images does not only appear in camera pose tracking but also in applications of object recognition and image registration e.g. for medical purposes. In our approach we want so solve for the 3D pose of a camera in a model-based tracking system. What we are focusing on, is a method for line feature-based model-image matching so that the knowledge about the model geometry and

perspective can be used to improve the correspondences and to define quality criteria, which is not a usual task.

Possible approaches for matching are intensity-based similarity measures regarding the entire image or patches of it, analysis of the image in the frequency domain or discrete image features like points and lines, describing visually perceivable structures in the image.

Detection and matching of feature points has been developed for a long time. First, the locations of interesting points like edge crossings or corners are detected in the image. The pixel surrounding of an interest point is then described by a vector of intensities, and may also include scale and orientation as SIFT [Low99] and SURF [Bay08] do. Matching is realized by comparing the entries of these descriptors which may be very time consuming due to scale space calculation. For acceleration there exist GPU-based implementations of feature detection, matching or tracking algorithms as the well-known KLT [Shi94] by [Sin06].

Feature edges can be detected by common image processing filters like the Sobel operator or more advanced developments as the canny algorithm [Can86]. Sobel and Canny implementations using the GPU in the context of a particle filter framework are shown in [Kle06] and [Bro12]. Line matching is mainly realized by minimizing the Euclidean distances between the projected model edges and corresponding gradients in the image. A simple distance measure may be gained by projecting the start and end point of the model edge to the image line or matching in parameter space. However, this requires a parameter transformation as Hough [Dud72], which may be expensive. In [Low91] simply the perpendicular distances of the projected model and the 2D image segments are used and in [Low92] a combination of distance and orientation is proposed.

Another popular distance-based matching method is the Moving Edges algorithm [Bou89]. It is used in various tracking frameworks as shown in [Har90],[Dru02],[Com03] or [Vac04] to name some of them. The model edge is sampled for control points and from these, orthogonal search lines are spanned in both directions. Alongside these line normals the gradient maximum of the image is calculated and the distance between 3D control point and 2D image point found is minimized. To deal with possible multiple gradient maxima the approach can be improved using multiple hypotheses for each sample point which provides higher stability [Vac04][Wue05].

3 THE MATCHING SHADER

3.1 Shader Outline

The model-based tracking approach uses a 3D model of the object to be tracked. Model edges can be obtained from this model by rendering an image with the last pose, detecting lines in the image and back-projecting to the model in order to gain 3D coordinates. Instead we use the model data structure directly by selecting individual edges and performing a visibility test. So the image processing step on the rendered image can be omitted. The advantage of a candidate edge list is that the matching result can be sorted and weighted by the quality of the matches.

For these 3D model edges corresponding 2D line matches should be found in the camera image. This camera image is canny-filtered so that natural structures are expressed as a binary image. The model edges selected for matching are projected and rendered with a matching shader from the pose of the last estimation with frame buffer write disabled. For each drawn pixel of the model edge the called pixel shader reads the value of the underlying canny image at the pixel position. If there is a black edge pixel in the canny image, the shader outputs a color. Otherwise it is discarded and the render pass will interrupt. The concept is displayed in figure 1 and listing 1 shows the matching shader in GLSL code.

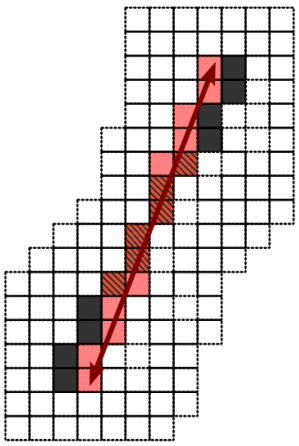

Figure 1: Rendered edge (red), image pixel edge (black) and common pixel to be counted (hatched).

The number of successful render passes now corresponds to the number of image line pixel counted. Retrieving this result number can be done by running occlusion queries while rendering (Section 3.3). The pixel count itself tells us about the probability that a found line in the canny image corresponds to a model edge. The number of counted pixel is a measure of the line length. Ideally the matching shader count equals the model edge length.

Using information of the model can help to improve the matching process. From the known pixel length of the rendered model edge we can expect a certain length of the image line response and thus define a threshold for a minimum pixel count. If the image line found does not fulfill this minimum length, it will be rejected as corre-

Vertex shader

```
void main()
{
    gl_Position = ftransform();
    gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

Fragment shader

```
uniform sampler2D cannyImg;
vec3 val;
void main()
{
  val = texture2D(cannyImg, gl_TexCoord[0].st).xyz;
  if(( val != vec3(1.0,1.0,1.0) ))
      gl_FragColor = vec4(0.0,0.0,0.0,1.0);
  else
      discard;
}
```

Listing 1: Counting shader.

spondence. In the next section we show further criteria for evaluation of the matches like depth and distance.

3.2 Sample Edge Generation

While we assume small movements of the camera from one image to the next, the model edges must be varied in position and orientation covering translations and rotations of the camera. This is done by sampling several new image edges around the known projected model edge. For each model edge start and end point in the image are known. Around these new points are sampled with an offset, e.g. on a 3x3 window around the central model edge start and end point, 8 new possible points are generated for each one. All generated points, including the original ones, are then connected to new edges, resulting in a total of 81 candidate edges in this case. Figure 2 and 3 show an example for some sample edges covering possible translations and rotations of the model edge. Notice that the sampling is done in 2D image space for projected edges. The 3D model data itself remains rigid.

For all of these candidate sample edges the matching shader returns a number of pixel counted as described in Section 3.1. The candidate edge returning the highest pixel count can be regarded as the best fitting match. Beneath the expected pixel length, it is even possible to consider the distance between the pixel count results of each candidate as quality criterion for matching. Similar parallel lines will return almost equal numbers of pixel count and thus can be recognized as ambiguous features. Such results may be rejected for matching.

Choosing the offset depends on accuracy and computational speed. A higher offset covers stronger move-

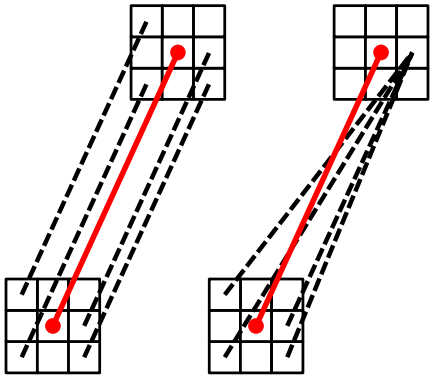

Figure 2: Generating edge samples (dotted) for a given model edge on a 3x3 window.

Figure 3: Sampled edges from the model in 2D image space. Only some random edges are shown for better visibility.

ments of the line features because more lines are sampled at greater distances and with wider angles, but this leads to a decrease in performance, especially when using occlusion queries. From the model knowledge the information about the depth of the 3D edge can be used to improve the sample edge generation. Movements far away from the camera lead to smaller shifting in pixel space while the same movement next to the camera is expressed in a large shift in pixel space. Knowing the depths of start and end point from the model edge, we can define different sizes for the sampling windows for both points, i.e. perspectively dependent generation of sample edges. If the depths of both points differ more than a threshold, for the nearer point more sample points are generated than for the farther point. This reduces the total number of sample edges to be rendered.

3.3 Occlusion Query Management

Retrieving the pixel count from the matching shader by occlusion queries affords a management process that can handle multiple queries to be executed fast. We have a list of model edges to be rendered and for each one a separate occlusion query has to be run. But the

graphics hardware limits the number of queries that can efficiently return a result in sequence. Trying to retrieve the counter result immediately after each query has finished would stall the CPU [Fer04].

While there are more edges to be rendered than queries can be executed, the task has to be splitted in several passes. A set of n maximal queries is created. A part of the model edges can be rendered until the maximum number of n available queries is reached. Then the results of all n queries have to be retrieved before a new block of n queries can be started for the remaining edges. The result with the highest count is stored. Alternatively an ordered list of the results can be created for better comparing of the results. The absolute number of query calls is also counted and the process finishes, when all edges have been drawn (See listing 2).

```
create n query objects
generate sample edges
enable shader
load canny texture
disable color and depth buffer write
while( query count != number of edges ){
    for n queries{
        start query
        render edge
        end query
        query count++
    }
    for n queries{
        retrieve result
        if query result > last query
            save result
    }
}
enable color and depth buffer write
disable shader
```

Listing 2: Using managed occlusion queries.

3.4 Advanced Texture Read-Back

The results showed that using a simple shader with occlusion queries does not perform very well with large sets of edges to match (see section 4). Therefore, we developed a more sophisticated shader for matching a significant amount of edges in short time by extending our first shader approach. It is based on texture read-back incorporating the sample edge generation. Opposing to the occlusion query approach the sample edges are not precomputed on CPU. The generation and computation of the sample edges is entirely transferred to graphics hardware. Thus the number of render passes is reduced to the number of model edges, instead of rendering each sample edge in its own pass. Further this solves the problem of stalling the CPU while waiting for the occlusion query result. The pixel count of all sample edges belonging to one model edge can be retrieved with one texture read-back.

In addition to the canny texture the pixel shader now gets the coordinates of projected 2D start and end point of the model edge and the offset for sample edge generation as input variables. As described below, the shader calculates new sample points in a window with the given offset around the start and end point of the model edge.

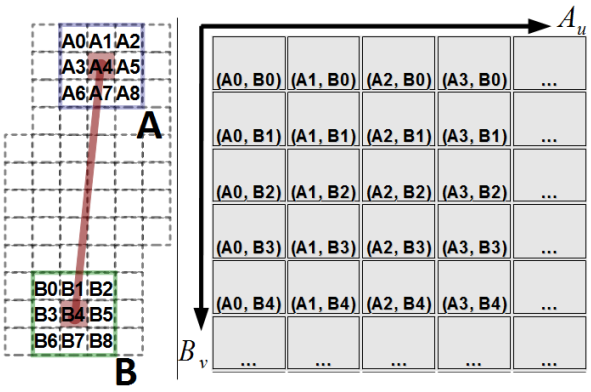

Figure 4: Shader target texture organization.

Each new start and end point is then connected to a sample edge. This is done by the shader performing the Bresenham line algorithm [Bre65a] between every start and end point generated. The pixel coordinates resulting from the line calculation are used to search for corresponding pixels on the canny-texture. The number of counted pixels is written as output value on the render-target texture. One render-target texture can store all counter results of the sample edges generated for one model edge. The texture has the size of all possible sample edges, e.g. when 9 sample points are generated in a 3x3 window for every start and endpoint, 81 sample edges are checked and this number of results has to be stored in the texture. Thus the texture must have size 9x9 for 81 entries.

Figure 4 shows the organization of the texture. For the start points A and end points B every column and its u-coordinate correspond to one start point in the sample window and every row and its v-coordinate correspond to one end point. Every pixel in the texture is now addressed for the result of one sample edge. The pixel shader is aware of the texture coordinate (u,v) it is going to write its value to, so it can use this information to apply an offset to the start and end point of the input model edge to generate the sample points. Thus, each pixel shader call calculates one sample edge depending on its writing position as follows.

Subtracting the offset from the x and y coordinates of the model edge start point A gives us the position of the first start point A0 with the lowest coordinates in the

sample window. From that point, adding the modulo of the u coordinate and window width s to the x coordinate and adding the division of the u coordinate by the window width s to the y coordinate results in the new sample start point:

$$s = 2*offset+1$$
$$sampleStartX = modelStartX - offset + (u \% s)$$
$$sampleStartY = modelStartY - offset + (u / s)$$
$$sampleEndX = modelEndX - offset + (v \% s)$$
$$sampleEndY = modelEndY - offset + (v / s)$$

Doing the same for the end point B and v coordinate returns the corresponding end point. Between these the Bresenham line will be calculated by the shader and the pixel count is written to the current position.

After the shader run the texture is read-back to CPU and the maximum value is determined by comparing all pixel values. From the pixel position (u,v) of the maximum and the offset applied, the pixel coordinates of the resulting sample edge can now be identified by the formulas shown above. It is also possible to use parallel reduction as described in [Fer04a] to directly obtain the maximum on the texture instead of using the CPU. Advanced computation of the resulting texture, like applying a non-maximum suppression leads to further quality criteria beneath the length of the matching line. The counting results of parallel edges are ordered diagonally on the target texture, which enables a quick check for this second quality criterion, e.g. given the case of figure 4 having a maximum at pixel position (2,2) and one or more significant high counts on one of the pixels in the diagonal from (0,0) to (8,8) this shows the existence of at least one parallel line with similar length and may lead to rejection of this match.

3.5 Optical Flow Support

Strong shaking of the camera may introduce a high level of motion blur. Due to the limited search area defined by the generated sample edges, the correct matching and subsequently the tracking may be lost when the image line is shifted too far. To prevent this, strong movements can be detected by estimation of optical flow [Bea95]. Calculating optical flow predicts the displacement of pixels in between two frames of an image sequence. The movement of the distinct start and end points of each projected model edge between the last and the current camera frame can be determined using a sparse optical flow function from OpenCV [Ocv11] that accepts an array of feature points as input. The predicted new start and end point positions corrected by optical flow are then used for sample edge generation. Matching with optical flow support allows reducing the distribution of sample edges because the matching can be performed in a smaller region. Using fewer sample edges leads to faster computation time. Another method to overcome motion blur is using additional inertial sensors [Rei06] to estimate rapid camera motion.

4 RESULTS

We tested our shader-based matching approach by tracking simple and complex objects on indoor and outdoor scenes (figure 5). As camera input we used video streams at a resolution of 640x480 pixels with varying lighting conditions. The canny filter applied to the camera images is taken from the OpenCV [Ocv11] implementation with standard parameters (threshold1 = 50, threshold2 = 200, aperture = 3).

The initial camera pose is assumed to be roughly known at the start of the sequence, which is a prerequisite for model-based tracking. This may be done by manually aligning the model in the camera frame. The intrinsic parameters of the video camera delivering the input stream are gained from previous calibration. Models of the tracking scenes are available and from these lists of the model edges are built to be used for the matching process. Each model edge is projected from the current camera pose and the matching shader returns the corresponding image line. For the computation of the new camera pose from the line correspondences we use a non-linear Levenberg-Marquardt optimization.

Figure 5: Test scenes (video and rendered model).

We compared our shader approach to two other distance-based matching methods. One method is to parameterize the binary canny image by a Hough transform [Dud72]. The model edge is projected into the image plane and a window around this edge defines a region of interest where the Hough transform is run. The output is a list of straight image lines defined by the parameters of line angle to the y-axis and line distance to the image origin. These can be directly compared to the parameters of the corresponding

model edge. However, matching in two dimensional parameter space proves to be very unstable. Neither length nor line similarity is judged this way, so we did not further consider this approach. Measuring the distance in image space can be done by projection of the start and end point of the model edge to the straight image line found by the Hough transform. The problem is the high dependency of the results on the chosen parameters of the transform. Possible matches are extremely ambiguous and lead to jittering in the estimated pose. At strong motion some matches completely fail.

Another popular approach is to set control points along the model edge and search for strong image gradients on orthogonal lines through these control points. We used an implementation from [Vis11] for our tests. The pose becomes more stable, but movements or interruptions in the image lines corrupt the matching result. Generally, these distance measures in image space can lead to stable camera pose estimation when the camera movement is slow and smooth enough, which is the case in the controlled indoor test scenario (figure 5 top, middle). However, at fast camera movements inducing motion blur the matching fails. We will show this on examples of the outdoor scene (figure 5 bottom).

Our matching shader has proven to deliver good matching results with minimal error even in the worst scenario of a video captured with a strongly shaking hand camera. The following figures illustrate the results of three matching approaches on a test sequence after the occurrence of strong motion. The motion blur occurs for duration of 6 frames while the image content is shifted over 80 pixels in this time. We regard the matching error in the frame right after this strong motion. Figure 6 shows the shift of the image within the 6 frames and the sub-picture the moment of strongest motion blur in the video sequence which disturbs the canny image to a large extent. Image lines are only partly visible and hard to handle by the matching methods.

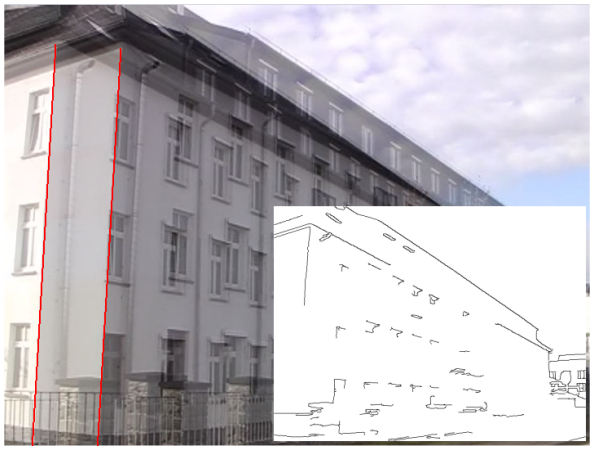

Figure 6: Image shift with strong motion blur.

The line projection approach (figure 7) obviously gets distracted by the parallel pipe next to the house corner, which is an ambiguous feature. The error ends up with a maximum displacement of 37 pixels. The matching with orthogonal search (figure 8) is more precise concerning ambiguities but also gets disturbed by the motion blur up to an error of 24 pixels. The matching shader approach with optical flow support (figure 9) overcomes the blur and results in an error of 3 pixel displacement.

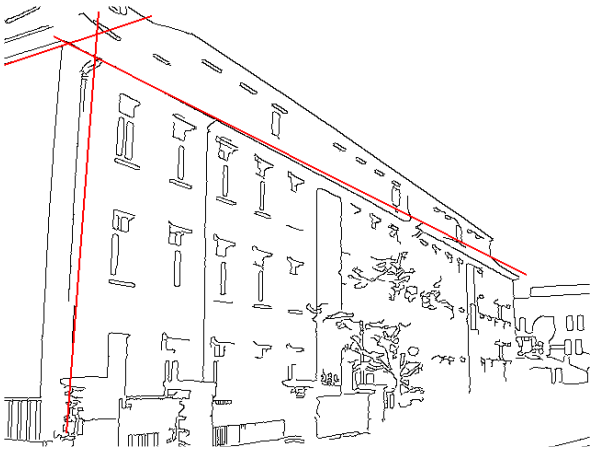

Figure 7: Line projection results after motion.

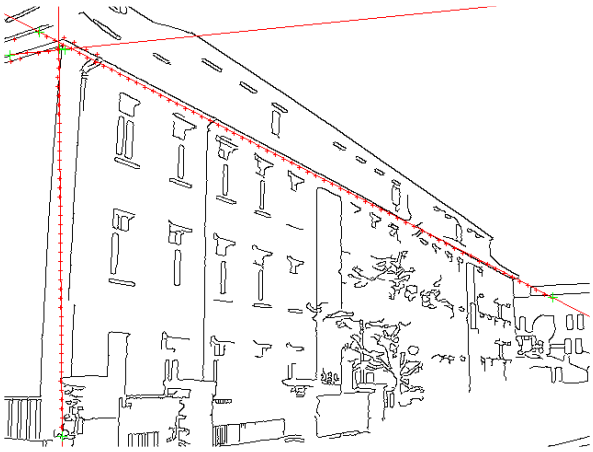

Figure 8: Orthogonal search results after motion.

In figure 10 the moment of strongest motion blur in the outdoor scene can be seen together with the resulting pose estimation overlay from the matches proposed by our method. Concerning the parameters, in our tests we found a good window size for the generation of sample edges at 4x4 with supporting optical flow. Without optical flow the best trade-off between computation time and matching quality could be reached with generating sample edges at 7x7 windows. The best threshold for rejection of the image line length as match is $\frac{3}{4}$ of the model line length.

Table 1 lists the average computation time in milliseconds of the components for our matching approach on

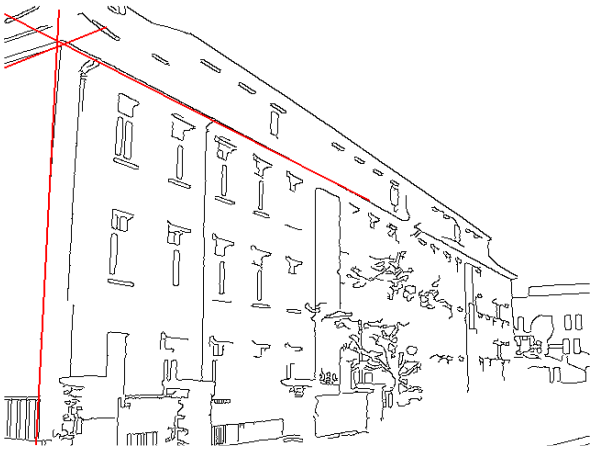

Figure 9: Matching shader results after motion.

Figure 10: Correct camera pose computation at strong motion blur (top) and for other scenes.

a Intel Core2Duo 3.2GHz with nVidia GeForce GTX 285. The canny filter and optical flow calculation step are called only once for a new frame, independently of the number of edges to match. Although time consumption for the canny filter is not too high, this additional step can be reduced by implementing gradient search inside the shader instead.

Next, times for the occlusion query approach and the texture read-back variant are compared. The number in brackets names the amount of edges to render. Obviously using a query executes fast only when a very little number of edges is used. But enlarging the number of model edges the texture approach quickly outperforms the query usage. Overall, real-time capability for tracking is ensured, however the implementation is not yet optimized.

Canny filter	5 ms
Optical flow	4 ms
Occlusion query	97 ms (11), 54(7), 7(2)
Texture read-back	18 ms (11), 14,(7), 7(2)

Table 1: Computation time.

5 CONCLUSION

We presented a shader approach for matching corresponding image lines to model edges in a model-based tracking scenario. The knowledge about the model is used to improve the matching and to define criteria for match selection. For a given model edge based on the last pose several sample edges are generated and rendered with a matching shader. The shader counts underlying pixels of a canny-filtered camera input image at the position of the edges. The image line with highest accordance to criteria of length and distance is chosen as match. This procedure delivers good matching and results in a correct camera pose estimation even at occurrence of strong motion blur.

We compared two methods to realize the matching shader. Using a simple counting shader and occlusion queries to retrieve the pixel count result is straightforward but significantly lowers the frame rate when many sample edges are generated because each render pass. The more sophisticated way is to read-back a texture value which can be done quite fast. The whole process of sample edge generation can be transferred into the shader, so a render pass is only called once per model edge instead for each sample edge.

Although we used straight lines from our testing models, the work is not limited to this type of feature. The counting shader can be extended to run on other renderable structures as well, like circles, curves or NURBS. For this purpose the sample generation algorithm has to be adapted to the wanted structure to match. A further advancement could be the integration of gradient calculation into the shader. This would save the canny preprocessing step to the camera image. Additionally, when calculating the gradient, the gradient orientation is also known. This could be used by the matching shader to count those pixels only, which have the same gradient direction and thus belong to the same line.

6 ACKNOWLEDGMENTS

This work was supported by grant no. MU 2783/3-1 of the German Research Foundation (DFG). We also want

to thank our former colleague Niklas Henrich for his support on GPU programming and Carsten Neumann from University of Louisiana at Lafayette for technical discussions.

7 REFERENCES

[Bay08] H. Bay, A. Ess, T. Tuytelaars and L. Van Gool. SURF: Speeded Up Robust Features. Computer Vision and Image Understanding (CVIU), 110(3), pp. 346–359, 2008.

[Bea95] S.S. Beauchemin and J. L. Barron. The computation of optical flow. ACM Computing Surveys, 27(3), pp. 433-466, 1995.

[Bou89] P. Bouthemy. A Maximum Likelihood Framework for Determining Moving Edges. IEEE Transactions on Pattern Analysis and Machine Intelligence, 11, pp. 499-511, 1989.

[Bra11] A.K. Braun and S. Mueller. GPU-assisted 3D Pose Estimation Under Realistic Illumination. 18th WSCG International Conference on Computer Graphics, Visualization and Computer Vision, Plzen, Czech Republic, 2011.

[Bre65] J.E. Bresenham. Algorithm for Computer Control of a Digital Plotter. IBM Systems Journal, 4(1) , pp. 25-30, 1965.

[Bro12] J.A. Brown and D.W. Capson. A Framework for 3D Model-Based Visual Tracking Using a GPU-Accelerated Particle Filter. IEEE Transactions on Visualization and Computer Graphics, 18, pp. 68-80, 2012.

[Can86] J. Canny. A Computational Approach To Edge Detection. IEEE Trans. Pattern Analysis and Machine Intelligence, 8(6), pp. 679-698, 1986.

[Com03] A.I. Comport, E. Marchand and F. Chaumette. A Real-Time Tracker for Markerless Augmented Reality. ACM/IEEE Int. Symp. on Mixed and Augmented Reality, pp36-45, Tokyo, Japan, 2003.

[Dru02] T. Drummond and R. Cipolla. Real-time visual tracking of complex structures. IEEE Transactions on Pattern Analysis and Machine Intelligence, 24(7), pp. 932-946, 2002.

[Dud72] R.O. Duda and P.E. Hart. Use of the Hough Transformation to Detect Lines and Curves in Pictures. Communications of the ACM, 15(1), pp. 11-15, 1972.

[Fer04] R. Fernando. GPU Gems. Programming Techniques, Tips, and Tricks for Real-Time Graphics. Addison-Wesley, Longman, Amsterdam, 2004

[Har90] C. Harris and C. Stennet. RAPID - A Video Rate Object Tracker. In Proc. British Machine Vision Conference, pp. 73-77, Oxford, UK , 1990.

[Kle06] G. Klein and D. Murray. Full-3D Edge Tracking with a Particle Filter. Proc. British Machine Vision Conference (BMVC'06), 3, pp. 1119-1128, 2006.

[Kle07] G. Klein and D. Murray. Parallel Tracking and Mapping for Small AR Workspaces. ACM/IEEE Int. Symp. on Mixed and Augmented Reality, pp. 225-234, Nara, Japan, 2007.

[Low91] D.G. Lowe. Fitting Parameterized Three-Dimensional Models to Images. IEEE Transactions on Pattern Analysis and Machine Intelligence, 13(5), 1991.

[Low92] D.G. Lowe. Robust model-based motion tracking through the integration of search and estimation. International Journal of Computer Vision, 8(2), pp. 113-122, 1992.

[Low99] D.G. Lowe. Object Recognition From Local Scale-Invariant Features. International Conference on Computer Vision, pp. 1150-1157, Corfu, Greece, 1999.

[Ocv11] OpenCV library, version 2.3.1, taken from http://opencv.willowgarage.com/wiki/

[Rei06] G. Reitmayr and T.W. Drummond. Going out: Robust Tracking for Outdoor Augmented Reality. International Symposium on Mixed and Augmented Reality (ISMAR06), pp. 109-118, 2006

[Sch09] M. Schumann, S. Achilles and S. Mueller. Analysis by Synthesis Techniques for Markerless Tracking. Virtuelle und Erweiterte Realität, 6. Workshop der GI Fachgruppe VR/AR, Braunschweig, Germany, 2009.

[Shi94] J. Shi and C. Tomasi. Good Features to Track. IEEE Conference on Computer Vision and Pattern Recognition, pp. 593-600, 1994.

[Sin06] S. N. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc. GPUbased video feature tracking and matching. Workshop on Edge Computing Using New Commodity Architectures (EDGE 2006), 2006.

[Vac04] L. Vacchetti, V. Lepetit and P. Fua. Combining Edge and Texture Information for Real-Time Accurate 3D Camera Tracking. ACM/IEEE Int. Symp. on Mixed and Augmented Reality, pp. 48-57, Arlington, USA, 2004.

[Vis11] ViSP Visual Servoing Platform library, version 2.6.1, taken from http://www.irisa.fr/lagadic/visp/visp.html

[Wue05] H. Wuest, F. Vial and D. Stricker. Adaptive Line Tracking with Multiple Hypotheses for Augmented Reality. ACM/IEEE Int. Symp. on Mixed and Augmented Reality, pp. 62-69, Santa Barbara, USA, 2005.

[Wue07] H. Wuest and D. Stricker. Tracking of Industrial Objects by Using CAD Models. Journal of Virtual Reality and Broadcasting, 4(1), 2007.