Sampling feature points for contour tracking with graphics hardware

Erwin Roth, Giorgio Panin, Alois Knoll

Technische Universität München, Fakultät für Informatik
Boltzmannstrasse 3, 85748 Garching bei München, Germany
Email: eroth@mytum.de, {panin,knoll}@in.tum.de

Abstract

We present in this paper a GPU-accelerated algorithm for sampling contour points and normals from a generic CAD model of a 3D object, in order to aid contour-based real-time tracking algorithms. The procedure achieves fast computation rates for generic meshes consisting of polyhedral, non-convex as well as smooth surfaces. This method is part of a general purpose, multi-camera and multi-target framework, supporting rigid and articulated objects, in order to achieve a high degree of generality for different tracking scenarios.

1 Introduction

Contour-based tracking is a class of methodologies that make use of contour models from an object in order to estimate its position and orientation in space. This information is obtained by projecting the wireframe model at a given pose hypothesis onto the current image, and identifying the visible *feature edges* [12] suitable for tracking: these can be defined as a subset of the visible object contours at a given viewpoint, that can be reliably identified because located on a significant color or shading discontinuity; for example, silhouette and flat surface boundaries, sharp internal edges of polyhedra, and texture edges.

From the visible feature edges at a given viewpoint, a set of sample points and screen normals is usually selected and matched with the image data, by means of a *likelihood* function that can be defined in several possible ways [6, 10, 9].

All of these algorithms require real-time processing capabilities in order to achieve robustness against object motion, for complex models consisting of possibly thousands of polygons, as well as multi-target scenarios. Moreover, especially for non-convex models, they usually must be formulated in a multi-camera setting [4], which can solve the inherent localization ambiguities arising from a single view of the object's silhouette.

However, visibility computation and edge sampling can result in a very expensive procedure, if performed on the CPU with standard computational geometry tools [11, 16], whereas on modern graphics cards this procedure may be tremendously accelerated, thanks to the hardware-accelerated polygon rendering and depth testing capabilities.

Several requirements can be identified concerning the algorithm for sampling visible edges in an object tracking context.

First of all, the method should be able to handle generic CAD wireframe data for non-convex models, preferably without requiring wireframe preprocessing or simplification, in order to keep precision of pose estimation.

For real-time applications, the overall computational time should also be fast (within $5ms$) and model-independent.

The algorithm should further be able to distinguish between edge types, which may require a different handling related to the tracking method or imaging properties: most notably boundary, contour, crease, marked and silhouette edges [12].

Concerning pose estimation, it is also important to get a uniform sampling of contour points in image space. This allows a better conditioning of the optimization algorithm, and exploits in the best way the information obtained from image-detected edges.

Moreover, in order to keep uniform computational requirements, as well as a uniform estimation precision across different object postures, the method should support a dynamic sampling interval in order to achieve a more or less constant number of sample points, independent on the projected object size.

For many purposes, image sample points should

also keep the information about the original points in model space, from which they are projected.

Concerning overlapping, parallel edges that belong to the surface horizon (i.e. with normal vector almost orthogonal to the viewing ray) or parallel edges which are very close in image space with respect to the pixel resolution, the method should be able to suppress them automatically.

A crucial requirement for multi-target scenarios, within a unified framework like the one described in [13], is the ability to detect and handle mutual occlusions, eventually varying the sampling density on a per-target base, up to the sensor's physical resolution limit for occluded or clipped objects.

Finally, a very important requirement for tracking articulated structures [3, 5] is the support of differentiated contour point sampling from different object parts.

In the computer graphics literature, there are several works [8] dealing with the problem of efficiently identifying and rendering feature edges, in order to obtain non-photorealistic rendering (NPR) for cartoons and artistic drawings.

Following the ideas proposed by the above mentioned works, we developed a GPU-accelerated algorithm that copes with the above mentioned requirements, while achieving the required speed for real-time tracking tasks.

The paper is organized as follows: Section 2 describes the general multi-camera setting and matrix notation; Section 3 gives an overview of the algorithm, and Sections 4 and 5 describe more in detail the off-line and on-line programming steps for both vertex and fragment shaders; experimental results and computational times are given in Section 6, and Section 7 concludes the paper and proposes future developments.

2 Camera views and pose parameters

Our target scenario consists of multiple, calibrated cameras for tracking the pose of an object in 3D space, with respect to a *world* reference frame (Fig. 1).

In a general setting, the current (4×4) transformation matrix ${}^{W}_{O}T$ between world and object is obtained by a *prediction* step from the previous frame estimate, and is updated by the tracking algorithm according to the measurement likelihood.

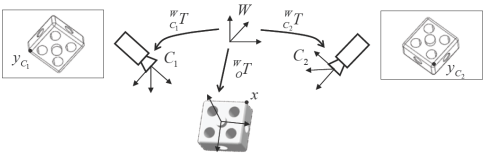

Figure 1: Coordinate frames for the multi-camera mapping

Depending on the required class of transformations (similarity, affine, articulated etc.), we can express the homogeneous (4×4) transformation matrix ${}^{W}_{O}T$ in terms of a corresponding minimal set of *pose parameters* p

$$ {}^{W}_{O}T(p) = {}^{W}_{O}\overline{T}\delta T\,(p) \tag{1} $$

where the incremental matrix δT is function of the pose parameters, and ${}^{W}_{O}\overline{T}$ is a reference transform, estimated from the previous frame.

For each camera C_j, the following information is supposed to be provided off-line, via a standard calibration procedure: the intrinsic (3×4) projection matrices K_{C_j}, and the extrinsic transformations ${}^{W}_{C_j}T$ between world and camera frames.

Therefore, we define a *warp* function, that maps object points to image pixels

$$ y = W(x, p, C_j) \tag{2} $$

which is computed in homogeneous coordinates as

$$ \begin{aligned} \overline{y} &= K_{C_j}\left({}^{W}_{C_j}T\right)^{-1}\,{}^{W}_{O}T(p)\overline{x} \\ y &= \left[\frac{\overline{y}_1}{\overline{y}_3}\;\frac{\overline{y}_2}{\overline{y}_3}\right]^T \end{aligned} \tag{3} $$

Many pose estimation procedures [6] are based on nonlinear LSE cost functions, which also require first-order derivatives of W in $p = 0$, given by

$$ \begin{aligned} \left.\frac{\partial y}{\partial p_i}\right|_{p=0} &= \frac{1}{\overline{y}_3^2}\begin{pmatrix} w_{i,1}\overline{y}_3 - w_{i,3}\overline{y}_1 \\ w_{i,2}\overline{y}_3 - w_{i,3}\overline{y}_2 \end{pmatrix} \\ w_i &= K_{C_j}\left({}^{W}_{C_j}T\right)^{-1}\,{}^{W}_{O}\overline{T}\frac{\partial\,(\delta T)}{\partial p_i}\overline{x} \end{aligned} \tag{4} $$

Uncalibrated, monocular 2D tracking tasks can be dealt with, by letting $K = [I\ 0]$ and defining the 2D transformation (similarity, affine, etc.) and Jacobians accordingly.

3 Overview of the sampling algorithm

The main target of our work consists in sampling good features for tracking from the object model, under a given pose and camera view.

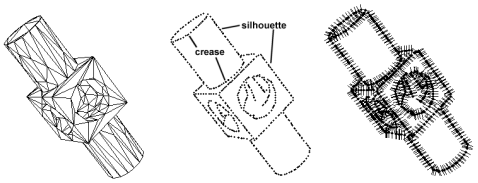

Figure 2: Sampling visible feature points and normals from a wireframe CAD model.

Starting from a polygonal mesh model (Fig. 2), we first identify the visible feature edges at pose ${}^{W}_{O}T$:

- *crease*: sharp edges between front-facing polygons of very different orientation
- *boundary*: boundary edges of flat surfaces
- *silhouette*: boundary lines, located on the surface horizon

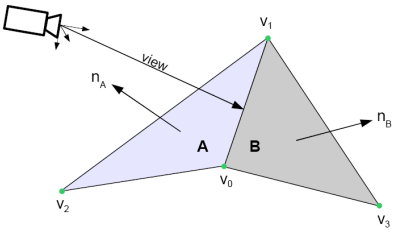

Figure 3: Edge from v_0 to v_1 with adjacent vertex information

Afterwards, feature points h_i are uniformly sampled (right side of Fig. 2) in image space, also providing screen normals n_i, as well as Jacobians $\frac{\partial h_i}{\partial p}$.

For this purpose, we developed a GPU-based procedure inspired by [12], that efficiently makes use of vertex and fragment shaders in order both to compute visible edges, and to subsample them uniformly in image space, also providing their location in 3D object space (which is required for computing Jacobians).

The algorithm is implemented platform independently with OpenGL 2.1.2, and makes use of the OpenGL shading language GLSL 1.20 (`http://www.opengl.org/documentation/glsl/`) for implementing vertex and fragment *shader* programs. Furthermore, it requires that the graphics adapter driver supports the vendor-independent OpenGL extensions *GL_EXT_texture_integer* [1] and *GL_ARB_occlusion_query* [2].

4 Offline processing steps

The overall procedure can be split into offline and online computational steps (Fig. 4). Offline procedures are executed only once, before the tracking sequence starts, and are independent on the object pose parameters.

The first offline step builds a list of unique edges for each object (an articulated object can consist of multiple parts) from the CAD wireframe model, which is stored in an XML COLLADA (`http://www.collada.org`) file format. This list contains the geometry data of the two vertices v_0 and v_1 in object space, along with the edge tangent vector from v_0 to v_1, and a reference to all polygons which share the edge; additionally, the face normals of the referring polygons are also stored.

Henceforth we will refer to the object-specific list index as *shape ID*, and to the edge position within the list as *unique edge index*. Furthermore, the unique edge count per object is stored (or, in case of an articulated object, the sum over all body parts).

In a second step, we encode the above mentioned data into an *edge mesh*, similar to the one described in [12]. This is obtained by iterating over all unique edges of each object, and storing the respective adjacency information as *per-vertex* attributes (Fig. 3): object space coordinates of v_0, v_1, v_2 and v_3; normals of adjacent polygons A and B (or just A for a boundary edge) n_A, n_B; the unique edge index; per-edge vertex array indices (0 for v_0 and 1 for v_1).

The latter information is important (see Sec. 5.2), since in the edge mesh each line is actually encoded with identical vertex attributes, apart from the vertex array index, so that it must be called twice (first with index 0, then 1), in order to obtain a real line primitive for OpenGL. The edge mesh of each object is then compiled into a line primitive *display list*.

A third step consists in creating and compiling the standard polygon display list for each object

(and their respective parts). The two list types are assigned to the respective objects by using the shape ID.

Finally, for the last offline step, the OpenGL framebuffer and shader programs are prepared as required for the subsequent online operations of the rendering pipeline.

In order to perform the off-screen texture rendering, a framebuffer object is instantiated, with three integer-valued textures and a depth buffer texture. All textures have same size as the output image, for which we do in a later step the visible feature matching.

5 Online processing steps

Five runs of the OpenGL rendering pipeline are required by our algorithm, each one with different vertex and fragment shaders, as described hereinafter. In the following, we will denote a specific instance of an object to be tracked as a *target*, consisting of a single or multiple parts, the latter in case of an articulated object.

Online processing starts from the world pose parameters for all targets, by setting a projection matrix for each camera, all target-specific model-view matrices, and clearing all color textures and the depth buffer.

Off-line

1. Generate a list of unique edges per object from the CAD wireframe model, with adjacency data
2. Build *edge meshes* from the unique edge lists and compile it as OpenGL display lists
3. Generate and compile an OpenGL polygon display list for each object (and object part)
4. Compute the minimum bounding sphere for all object meshes
5. Initialize the OpenGL framebuffer contexts

On-line Update projection and model view matrices for all targets and cameras, and execute the following shader runs:

1. First run: draw all targets as filled polygons
2. Second run: select feature edges for tracking; use the edge mesh list in order to draw selected edges onto the texture produced by the first run, with depth test enabled; count the number of visible edge pixels per target
3. Third run: compute the potential number of visible sample points per unique edge (optionally: filter non-silhouette edges)
4. Fourth run: sum up the sample point numbers for multiple unique edges of the previous run
5. Fifth run: for each target, adjust the sampling density per edge, in order to reach the desired number of sample points per target; create for each visible sample point a fragment in the output texture vector (optionally: suppress visible sample points from nearby edges with similar orientation)
6. Copy visible sample point data back to the CPU
7. On the CPU: recover sample point locations in object space, by using the interpolation coefficient, and optionally compute screen Jacobians

Figure 4: The overall contour sampling algorithm.

5.1 First shader run

Object meshes are drawn as filled polygons into the depth and color buffer (Fig. 5a), with depth test enabled, where the fill colors encode the respective target ID; polygonal faces also receive a small depth offset during rendering, in order to ensure that all visible edges of the next rendering step pass the depth test (avoiding the *stitching* phenomenon).

In order to get the highest depth buffer resolution, far and near *clipping planes* are set as the distance of the farthest and nearest target in camera space respectively, by taking into account the respective bounding sphere radii. For articulated models, the overall radius must be re-computed at every pose update.

5.2 Second shader run

In the second shader run, the edge mesh display list (from off-line step 2) is called, and the output is stored as an integer texture (Fig. 5b). In particular, each edge is drawn with a *thickness* value larger than one pixel, in order to be able to successfully subsample it during the next run.

Depth testing is also enabled, using the depth buffer data created during the first run. Most computations here are done inside the vertex shader.

First, a test is performed in order to select feature edges, satisfying at least one of the following properties. When the silhouette is required, only boundary and contour edges are considered as fea-

ture edges.

$$\begin{array}{ll} \text{Contour} & [n_A \cdot view < 0] \ XOR \ [n_B \cdot view < 0] \\ \text{Ridge} & [n_A \cdot n_B < -\cos\theta_R] \ AND \ [(v_3 - v_2) \cdot n_A \leq 0] \\ \text{Valley} & [n_A \cdot n_B < -\cos\theta_V] \ AND \ [(v_3 - v_2) \cdot n_A > 0] \\ \text{Boundary} & v_3 = v_0 \end{array} \quad (5)$$

In this equation, n_A and n_B are the unit polygon normals, *view* is the camera-to-object ray, θ_R and θ_V are the ridge and valley angle thresholds (Fig. 3).

Only feature edges are furtherly processed by the vertex and fragment shader, while the others are discarded, by putting the respective vertices onto a clipping plane.

Afterwards, the two vertices (v_0 and v_1) are projected onto the screen, by using the model view and projection matrices.

The most front-facing of the two adjacent polygons that share the edge is selected, according to the scalar product between normal and view vector. Then, the edge screen normal related to this polygon is computed, and its direction angle, with respect to the horizontal axis, is stored as a single integer value.

The choice of the normal direction is done in order to get uniformly (inward or outward) pointing normals, an approach which is advantageous for region-based matching algorithms using the silhouette contour (e.g. the CCD method [14]).

During the subsequent rasterization process of the OpenGL rendering pipeline, the distance of the current edge pixel in image space from the edge vertex v_0, is interpolated and encoded inside *fragments*, where the interpolation coefficient is quantized, according to the bit size of the output color channel.

Subsequently, the fragment shader accesses the above interpolated value, and writes the following output per fragment:

- normalized distance of projected edge pixel from v_0
- target ID
- unique edge index
- edge screen normal direction

Finally, each fragment undergoes the depth buffer testing of the last pipeline stage, so that mutual- and self-occlusions between objects are handled.

The number of visible edge fragments per target which pass the depth test is computed, by using the OpenGL extension *GL_ARB_occlusion_query*. The resulting values are returned to the CPU, and used as input for shader run three and five. Since our algorithm should support multiple instances of the same shape, we also pass a dynamically generated target ID as an additional vertex attribute, besides the pre-compiled edge mesh list.

5.3 Third shader run

At this level, we compute the number of *potentially visible* sample points per edge, for each target. Also in this run, most of the processing is performed by the vertex shader, which again uses the edge mesh.

Each unique edge of this mesh is mapped to a fragment in the output texture, so that the overall number of unique edges is only limited by the texture resolution. The resulting fragment data for multiple targets are organized as one long vector, wrapped into multiple lines of the output texture (Fig. 5d).

The one-to-one fragment to pixel mapping is achieved in a standard way, by using an orthogonal projection model in place of the real perspective. In this run, only feature edges selected by (5) are furtherly processed, while the others receive a zero counter.

Subsequently, we check whether the edge lies completely or partially outside the viewing *frustum*, by projecting v_0 and v_1 onto the image and comparing their coordinates with the clipping planes: if both vertices were clipped, we set the value to zero. For partially clipped edges, their clipped length is also computed.

The output fragment position is obtained by adding a target-related offset to the unique edge index.

Individual edge points are here *subsampled* from the large amount obtained in the previous run (Fig. 5c); the resulting number of points will be encoded in the output fragment color (Fig. 5d).

In order to obtain a point count close to the desired number per target, we compute a target-specific uniform sampling distance, in image space.

This requires a careful computation, since the distance depends on several factors: the desired number of points per target, the visible edge fragments per target (from the previous run), the horizontal and vertical resolution of the texture, the ren-

dered line thickness, the unclipped length, and its orientation (since the pixel discretization of a segment is also orientation-dependent).

The sampling distance provides finally the number of *potentially visible* points per edge, which are looked-up in the input texture, and added if the corresponding pixel is found.

Feature edges longer than one pixel, but shorter than the sampling distance, will receive at least one sample point. As a consequence, the desired number of points per target must always be higher than the amount of visible edges, at any given pose.

5.4 Fourth shader run

In the fourth run, the sample point data of the previous run are compressed, by summing up the counters for multiple edges (Fig. 5e). This is done in order to minimize expensive texture accesses, during the subsequent run.

The compressed output is designed in order to occupy a single row in the output texture. This is obtained by adjusting the sum interval, according to the overall number of unique edges and the texture width.

The next run will use these data in order to compute the drawing offset for each edge, as well as the overall number of estimated points per target.

The main challenge in the fourth run lies in the fact that we cannot simply iterate over all edges of the previous texture, but we also have to take care of the target boundaries. Moreover, an accumulation over the edges of a target cannot be parallelized in a single shader run, and represents therefore a bottleneck for GPU computations that should be avoided.

5.5 Fifth shader run

The goal of the fifth run is to create a tightly packed vector of sample point data to be returned to the CPU (Fig. 5f,g), since large data transfers from the GPU have a severe performance impact.

The number of sample points per target should be as close as possible to the desired sample point value, yet never exceed this limit, independently of the projected area of the targets.

As optional user requirement, sample points of edges with similar orientation in a close neighborhood should be suppressed.

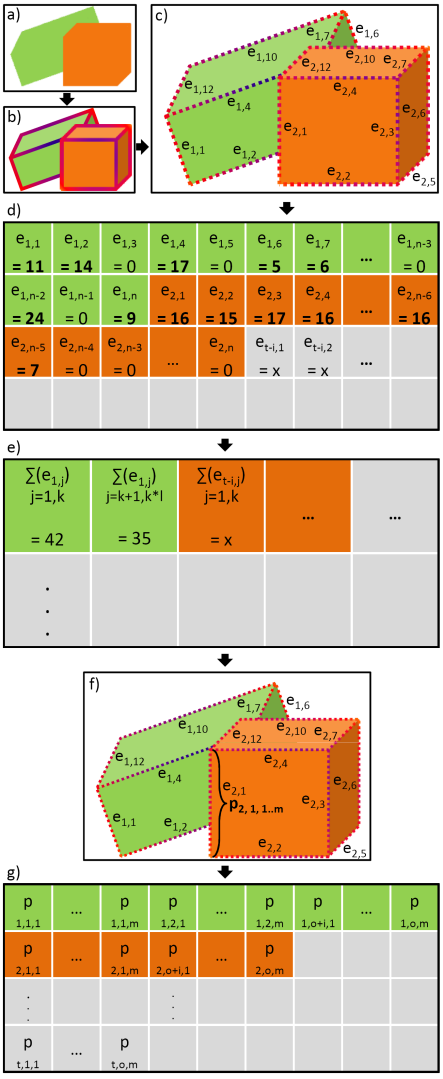

Figure 5: Shader textures: a) 1^{st} run; b) 2^{nd} run; c) 3^{rd} run edge sampling; d) 3^{rd} run output; e) 4^{th} run output; f) 5^{th} run adjusted sampling; g) 5^{th} run result; $e_{t,j}$ = potential edge sample points; t = target ID; j = unique edge index; k = edge accumulation interval ($k = 5; k * l \neq n$); m = edge-specific visible point counter; n = unique edge count per target; o = visible feature edge count; $p_{t,o,m}$ = sample point data for the respective position;

5.5.1 Vertex shader

This shader uses data generated by runs two, three and four. The first steps, in particular, are identical

to the ones of the previous run.

Afterwards, the potentially visible point number per edge is obtained from the corresponding texture pixel of run three (Fig. 5d). Processing is continued only for edges with at least one sample point.

We compute the horizontal offset of the first point, by accumulating the visible points for all edges belonging to this target with an increasing index. Subsequently, the overall sample points for this target are updated, by summing up the edge counters.

An initial sampling distance per target is computed as in shader run three; the overall samples are then compared with the desired value, and the distance adjusted on basis of the resulting ratio (plus a safety factor), as well as the corresponding horizontal offset in the output texture.

As in run three, we sample along the current edge in the texture generated of the second run, according to the above adjusted sampling distance (Fig. 5f), and count the successful hits. At the same time, we encode the hits profile into a bit array, with a bit set to 1 for successful samplings. This information is passed on to the fragment shader, in order to avoid expensive texture lookups.

The number of hits is also used for repositioning the edge mesh vertices v_0 and v_1, in order to form a horizontal line (vector) in the output texture, which consists of exactly as many fragments as successful samples. This counter is also used as the upper limit for a *varying* variable, whose interpolated value (after rasterization) is used by the fragment shader, in order to identify the bit array value to work on.

The horizontal offset of the left line vertex v_0 is given by the previously computed horizontal offset, while its vertical offset is instead defined by the target ID number.

5.5.2 Fragment shader

The offset of the current fragment within the horizontal line is defined by the rasterization process. For each fragment, the following sample point data are encoded into the color channels:

- normalized distance of successful sample point from v_0 (in image space)
- unique edge index
- direction of the screen normal

By using the corresponding sample point offset, we can retrieve the edge data from the texture of run two. This offset is computed by the respective bit array index value, and the sampling distance calculated in the vertex shader.

An additional user option allows to suppress sample points which are close to other edges of the same target with similar orientation, but closer to the camera.

If this option is disabled, the fragment shader performs only a single task, which consists of returning the data of the specific sample point of the current edge. Otherwise, the following steps are additionally executed before writing the output data.

Starting from the image coordinates of the current sample point, we search along both sides of the edge normal, up to a user defined distance, for other edges belonging to the same target, with similar orientation but a smaller depth value.

If a point which meets the above conditions is found, we suppress the current fragment output. This depth comparison requires also the depth buffer, which has been generated in the first run.

5.6 Data transfer and CPU-related computations

At this level, we copy the output of the last shader run back to the CPU, and store the sample point data into a hierarchical storage structure, ordered both by target and unique edge indices. The edge-wise organization of sample points is beneficial, for example in case of edge-based outlier removal methods like RANSAC [7].

Based on the vertical offset of the output data, we identify the target ID related to each sample point, and consequently the corresponding target mesh data.

From these data, we compute the object space coordinates of the point, by using the unique edge index, the edge tangent vector and the normalized distance of the sample point from v_0.

Finally, we can apply the warp function (2) to project it again on image space (which actually is a redundant step), as well as to get the Jacobians $\frac{\partial h_i}{\partial p}$. If required, the image space normal is also decoded and stored.

6 Experimental results

This Section presents experimental results, showing the algorithm's flexibility and performance, as

well as its behavior under special conditions like mutual object occlusions. The following tests were executed both under Windows XP and Linux OS, although in the following we present the results for Linux OS only.

Our test machine has been equipped with an Intel Core 2 Duo CPU, running at 2.13 GHz, with 2 GB RAM, and a NVIDIA 8600GT graphics adapter with 512 MB RAM. A Linux OS has been installed, together with the NVIDIA graphics driver version 169.12, supporting OpenGL version 2.1.2.

The timing results presented in the following, reflect the average time in milliseconds required by the algorithm to execute all of the online processing steps (see, Fig. 4). All test runs include computation of Jacobians for a 6-dof rigid body pose representation.

The optional suppression of nearby edges is disabled. All measurements, besides the ones shown in Table 2 were executed with a *mech-part* CAD model (`http://www-c.inria.fr/gamma/download/download.php`) (see Fig. 2) converted to the COLLADA format, with 358 triangles and 537 unique edges.
Table 1 shows performance results for different number of sample points per target, at an image resolution of 1024x768 pixel.

Sample point count	50	100	200	400	1000
Avg. time [ms]	1.40	1.46	1.56	1.74	2.19

Table 1: Sample point count comparison

Table 2 presents results for different object model sizes. The first column shows the timing for a simple cubic object, while the remaining ones use the mech-part object model, or subdivided versions of it. Image space resolution was set to 640x480 and the sample point threshold was set to 400. The current algorithm implementation is not optimized for meshes with less than 100 unique edges and a single target only.

Triangle count	12	358	1432	5728	22912
Unique edge count	18	537	2148	8592	34368
Avg. time [ms]	1.05	1.32	2.10	5.73	15.04

Table 2: Results for different model sizes

The algorithm can handle object meshes of high complexity, as show in Fig. 6.

Table 3 presents results for an increasing number of simultaneous targets within the same scene. To simplify testing, we used the same object model for all targets, although our algorithm is not limited to this case. Tests were executed with an image resolution of 1024x768, and a desired sample point number of 200 points per target.

Target count	1	2	5	10	50
Avg. time [ms]	1.56	2.16	3.70	6.37	27.09

Table 3: Results for multiple targets

In Table 4, results are shown for different resolutions, again with a single target and 200 sample points.

Image size-x	320	640	800	1024	1280	1600
Image size-y	240	480	600	768	1024	1200
Avg. time [ms]	0.92	1.13	1.29	1.56	2.02	2.57

Table 4: Results for different image sizes

Table 5 shows furthermore the behavior of the algorithm in a multi-camera setup using a single GPU. Tests were executed under the same conditions as before.

Camera count	1	2	3
Avg. time [ms]	1.56	3.15	4.68

Table 5: Multiple cameras comparison

Fig. 7a and Fig. 7b show the object pose, with occlusion-dependent dynamic sampling, that tries to keep the number of points per target as close as possible to the desired value, by adjusting dynamically the sampling distance. Fig. 7c shows the same scene as Fig. 7a, with only silhouette edges.

Additionally, Fig. 7d shows an example of active suppression of nearby edges with similar orientation.

Finally, Fig. 9 and Fig. 8 show edge sampling results for articulated 2D and 3D models.

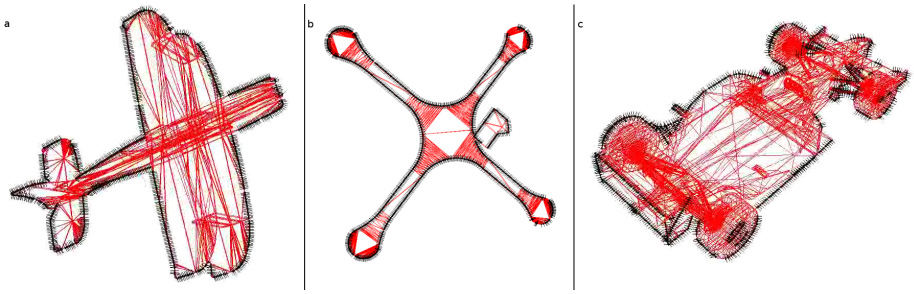

Figure 6: Silhouette contour sampling and normal computation for objects with complex meshes; a) Biplane airplane; b) Quadcopter; c) Formula 1 car

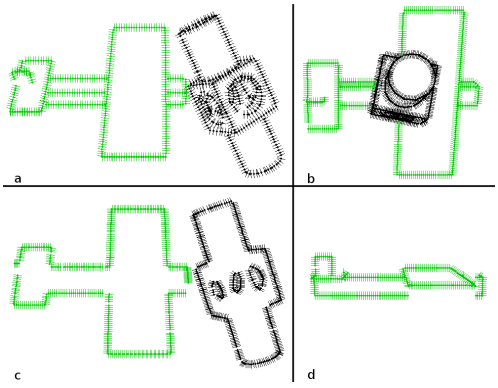

Figure 7: a,b) Pose and occlusion dependent visibility checking and sampling; c) sampling of silhouette edges only; d) sampling with activated suppression of vicinity edges

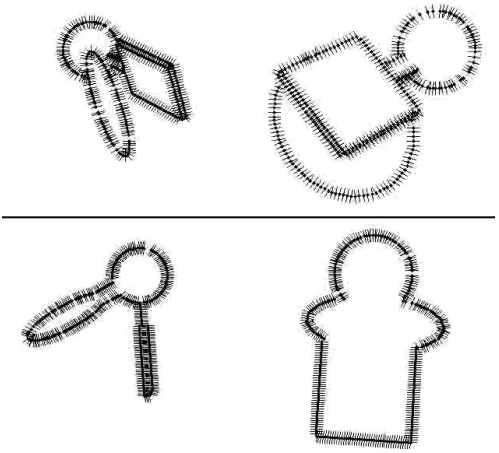

Figure 8: Visibility checking and sampling for an articulated object in 3D for a multi-camera setup (computation of front and side view): sampling of all feature edges (top row), sampling of silhouette feature edges only (bottom row)

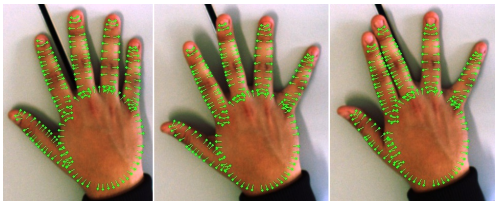

Figure 9: Edge sampling and normal computation for an articulated 2D hand model

Results of the algorithm's application to real-time 3D tracking tasks, involving different contour-based likelihood functions, can be found in [15].

7 Conclusions

We developed a novel GPU-accelerated visibility computing and feature edge sampling algorithm, which is capable to support real-time tracking applications of multiple, rigid or articulated 3D objects from generic CAD data. Future developments include the computation of image Jacobians directly on the GPU, and the integration of this algorithm with feature matching and likelihood computation directly on the graphics card.

References

[1] Pat Brown and Michael Gold. EXT_texture_integer (OpenGL Extension Registry).

[2] Ross Cunniff, Matt Craighead, Daniel Ginsburg, Kevin Lefebvre, Bill Licea-Kane, and Nick Triantos. ARB_occlusion_query (OpenGL Extension Registry).

[3] Teo de Campos. *3D Visual Tracking of Articulated Objects and Hands*. PhD thesis, University of Oxford, 2006.

[4] Tom Drummond and Roberto Cipolla. Real-time tracking of multiple articulated structures in multiple views. In *ECCV (2)*, pages 20–36, 2000.

[5] Tom Drummond and Roberto Cipolla. Real-time tracking of highly articulated structures in the presence of noisy measurements. In *ICCV*, pages 315–320, 2001.

[6] Tom Drummond and Roberto Cipolla. Real-time visual tracking of complex structures. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(7):932–946, 2002.

[7] Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, 1981.

[8] Bruce Gooch. Theory and practice of non-photorealistic graphics: Algorithms, methods, and production system. SIGGRAPH 2003 Course notes 10. course organized by M. C. Sousa.

[9] Robert Hanek and Michael Beetz. The contracting curve density algorithm: Fitting parametric curve models to images using local self-adapting separation criteria. *Int. J. Comput. Vision*, 59(3):233–258, 2004.

[10] Chris Harris. Tracking with rigid models. In *Active Vision*, pages 59–73, Cambridge, MA, USA, 1993. MIT Press.

[11] Aaron Hertzmann and Denis Zorin. Illustrating Smooth Surfaces. pages 517–526, New York, 2000.

[12] Morgan McGuire and John F. Hughes. Hardware-determined feature edges. In *NPAR '04: Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering*, pages 35–47, New York, NY, USA, 2004. ACM.

[13] G. Panin, C. Lenz, S. Nair, E. Roth, M. Wojtczyk, T. Friedlhuber, and A. Knoll. A unifying software architecture for model-based visual tracking. In *IS&T/SPIE 20th Annual Symposium of Electronic Imaging*, San Jose, CA, 2008.

[14] Giorgio Panin, Alexander Ladikos, and Alois Knoll. An efficient and robust real-time contour tracking system. In *ICVS '06: Proceedings of the Fourth IEEE International Conference on Computer Vision Systems*, page 44, New York, USA, 2006.

[15] Giorgio Panin, Erwin Roth, and Alois Knoll. Robust contour-based object tracking integrating color and edge likelihoods. In *Proceedings of Vision, Modeling, and Visualization*, 2008.

[16] M. S. Paterson and F. F. Yao. Binary partitions with applications to hidden surface removal and solid modelling. In *SCG '89: Proc. of the fifth annual symposium on Computational geometry*, pages 23–32, New York, NY, USA, 1989. ACM Press.