Efficient Rasterization for Edge-Based 3D Object Tracking on Mobile Devices

Etan Kissling
ETH Zurich

Kenny Mitchell
Disney Research Zurich

Thomas Oskam
Disney Research Zurich

Markus Gross
Disney Research Zurich
ETH Zurich

Abstract

Augmented reality applications on hand-held devices suffer from the limited available processing power. While methods to detect the location of artificially textured markers within the scene are commonly used, geometric properties of three-dimensional objects are rarely exploited for object tracking. In order to track such geometry efficiently on mobile devices, existing methods must be adapted. By focusing on key behaviors of edge-based models, we present a sparse depth buffer structure to provide an efficient rasterization method. This allows the tracking algorithm to run on a single CPU core of a current-generation hand-held device, while requiring only minimal support from the GPU.

CR Categories: H.5.1 [Information Interfaces and Presentation]: Multimedia Information Systems—Artificial, augmented, and virtual realities I.4.8 [Image Processing and Computer Vision]: Scene Analysis—Tracking;

Keywords: augmented reality, rasterization, pose tracking

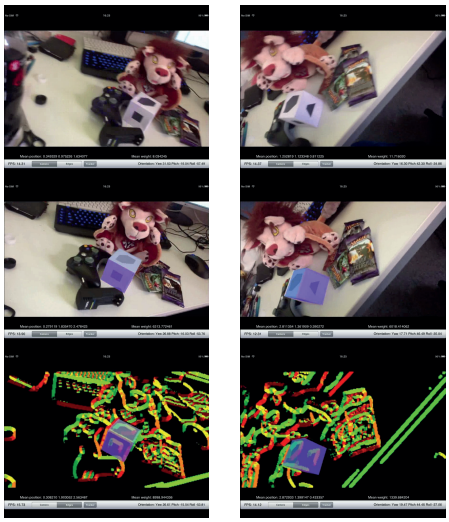
1 Introduction

Augmented reality modifies the perception of reality by embedding computer-generated information into images or videos. Because real-time operation is the key to provide an optimal user experience, algorithms have to be optimized for the specific device characteristics to still allow for the processing of gameplay logic and physical simulations.

Commonly, augmented reality applications provide printed cards that are used to estimate the device's position within the scene. Using this position, virtual contents can be placed in the correct locations in the device's video stream. This approach is very restricted, as the cards often incorporate specific textures, in order to be easily detectable by the system. These textures can look unnatural, and the question arises, whether actual real-world objects could be used as tracking targets.

One research paper that describes the tracking of three-dimensional rigid objects is "Full-3D Edge Tracking with a Particle Filter" [Klein and Murray 2006]. First, edge features are extracted from the camera image. By matching them with renderings of the object's known geometric shape from multiple viewpoints, weights for the different camera pose estimates are calculated. A particle filtering technique is then applied to further investigate likely pose estimates (see Figure 3).

SIGGRAPH Asia 2012, Singapore, November 28 – December 1, 2012.
© 2012 ACM 978-1-4503-1757-3/12/0011$15.00



(a) (b)

Figure 1: *Cluttered environment result. The top row shows the images directly after a fast movement. The result after tracking succeeds is shown in the middle row. The bottom row shows the overlay rendered on top of the extracted edge features from the camera image. The colors indicate the direction of the edge gradients. In* (a)*, the cube's position is detected after* 1.1 *seconds. In* (b)*, the cube's position is detected after* 0.7 *seconds.*

In contrast to much conventional work, the authors base their algorithm on edge-based models, as edge features are robust to image distortions, such as motion blur or noise. Edge features also consume more screen space, resulting in better handling of partial occlusions than corner-based algorithms. While Klein and Murray demonstrate their algorithm on a desktop workstation, we extend it to work on current hand-held devices that expose the *Open Graphics Library for Embedded Systems (OpenGL ES) 2.0* interface. Although being powerful, this interface does not include all features that are available in its desktop counterpart. This is problematic, as the `GL_OCCLUSION_QUERY` extension is not available, that is used by Klein and Murray to read back certain essential pixel counts from the GPU at real-time frame rates.

To still implement their algorithm on hand-held devices, we tried to apply an image pyramid-based approach to emulate the behaviour of the missing extension. However, this approach failed as not enough performance is offered in reading back data from the GPU to the CPU to meet the algorithm's needs. Another attempt incorporated the use of large textures with multiple renderings on it, as well as a method where the different color channels of the textures contained renderings for multiple pose estimates. This allows the evaluation of multiple pose estimate likelihoods in parallel. Unfortunately, the graphics hardware of consumer-grade mobile devices is not optimized for high-resolution texture processing, and the color channels do not offer the required bit-depth, leading to less accurate results.

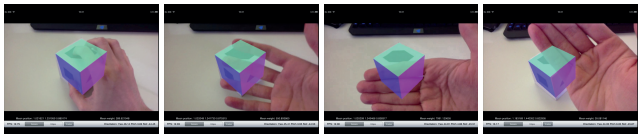


Figure 2: *Partial occlusions result. The images visualize how the tracker reacts to different types of occlusions. Even with major occlusions, the cube can still be tracked.*

In order to succeed with an implementation of the particle-based algorithm, we devised an efficient CPU-based rasterization method to replace the workstation GPU feature. Instead of using advanced optimizations applied to general purpose software rasterization algorithms, we employ a custom technique that focuses on reducing the necessary memory bandwidth by relying on a sparse depth buffer structure. This structure allows our rasterizer to run on a single CPU core at interactive frame rates.

2 Related Work

One of the earliest algorithms to achieve edge-based tracking is the *RAPiD tracker* [Harris 1993], where control points are inserted on model edges. To estimate the new camera pose, a search perpendicular to the edge direction is performed starting at the control points until an edge is found in the image. The camera pose is then updated by minimizing the distance between the previous and the current frame. This approach works only with small positional changes between the video frames, which we cannot guarantee in an augmented reality application on a hand-held device.

Other early approaches such as [Lowe 1990] try to detect the object in fast motion. Measurement and matching errors are modeled to accomplish this task. [Armstrong and Zisserman 1995] propose adding of redundant measurements to verify pose estimates and to allow the system to handle incorrect measurements [Rosten and Drummond 2005] use corner-features as a fallback mechanism when no accurate prior pose estimate is available. In [Klein and Drummond 2004], gyroscopes, which sense angular velocity, are used to further increase the accuracy between video frames.

We also make use of motion sensors in our implementation. By relying on the produced orientation data to track the device's orientation, we critically reduce the algorithmic complexity from six degrees of freedom down to three degrees of freedom, as only the position is required to be detected through image processing. We further exploit the motion data to react to shaking by increasing the estimated motion, and to slow down when the device is held still to reduce jitter effects.

Particle filters use an alternate approach. Instead of trying to increase the accuracy of a single pose estimate, multiple estimates in parallel are used to drop wrong estimates. Important work in this area is described in [Isard and Blake 1998] and visualized in Figure 3. [Klein and Murray 2006] extend this algorithm to provide three-dimensional edge tracking of complex models.

In order to evaluate the likelihood for a given camera pose, [Klein and Murray 2006] first rasterize the faces to the depth buffer. In a second stage, the edges are rasterized with activated depth testing. Therefore, only edge pixels that are visible are rendered. Another pass is made that leaves only edge pixels which also match the gradient directions of the camera image. The resulting likelihood is then directly deduced from the number of pixels that are left by these tests. This is convenient, as the `GL_OCCLUSION_QUERY` extension reveals these numbers without major drawbacks.

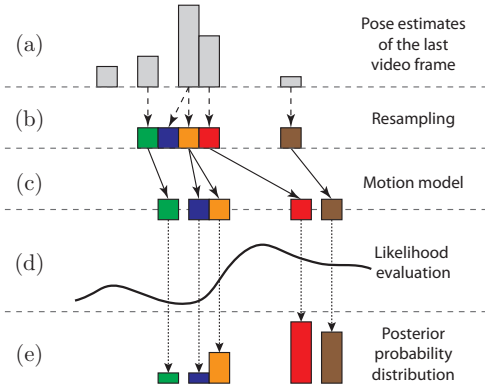


Figure 3: *Visualization of the coarse structure of a particle filter algorithm. Multiple camera pose estimates that are tracked in parallel are visualized as different bars. The height of the bars visualize the different weightings. In* (a)*, the estimates of the last frame are shown. They approximate the posterior distribution of the last frame. This distribution is then resampled in* (b) *to reduce the time spent on unlikely estimations. After applying a motion model to the different pose estimates in* (c)*, the new likelihoods are evaluated in* (d)*. The final probability distribution is shown in* (e)*.*

3 Efficient Rasterization

On *OpenGL ES 2.0* based mobile devices, *OpenGL Occlusion Queries* are not available, leaving only `glReadPixels` to retrieve data back from the GPU. Because this leads to a synchronization of the GPU with the CPU, the pipeline is stalled, leading to massive performance drops. On the device where we tested our algorithm, *OpenGL texture caches* allow to access the camera memory directly from *OpenGL*. However, this does not addresses the problem of synchronization, as graphics operations must still complete before the results can be retrieved. In order to achieve interactive frame rates, we propose an approach based on software rasterization.

Our implemented software rasterizer works in multiple stages. For each likelihood evaluation, we start by projecting the vertices into the corresponding camera coordinate system. Secondly, we rasterize the edges to pixels, and store their depth values. In the third step, the faces are also rasterized and used to detect and remove hidden lines, that exist on complex objects. The required pixel quantities to calculate the likelihood of the camera pose can finally be retrieved from the depth buffer structure.

The key problem with software rasterization on a mobile device is that memory fill rate becomes a critical performance factor. When a simple array is used for the depth buffer structure, the clearing operation at the beginning of each rendering already consumes too much processing time. We circumvent this issue by, first, reducing the size of the depth buffer, and, second, exploiting that edges consume far fewer pixels on the screen than filled faces. By rendering the edges before the faces (as opposed to the approach described in [Klein and Murray 2006]), we only have to check a few pixels for visibility during the face rasterization stage. This leads to fewer memory accesses (depending on the object's model and the viewing angle) and, therefore, allows this technique to work well.

During edge processing, we use the Cohen-Sutherland algorithm for clipping. The Bresenham line algorithm handles rasterization. Faces are rasterized as their corresponding collection of triangles. We start at the top corner of the triangle, and follow the two adjacent edges until reaching the center corner of the triangle. For each pixel row, the depth buffer structure is used to detect what pixels within

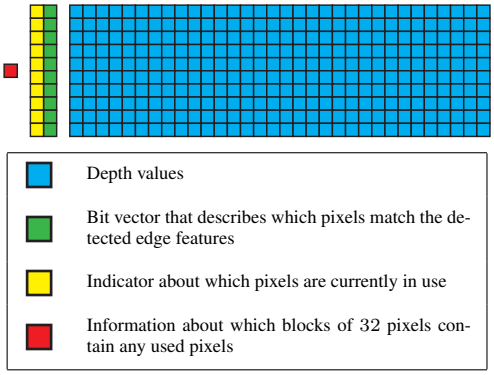


Figure 4: *One row of the depth buffer structure. Each square corresponds to* 4 *bytes of memory. Additional records are packed into* 84 *bytes per pixel row.*

it are hidden by the processed face. The bottom half of the triangle is processed analogously.

3.1 Depth Buffering

When implementing a software rasterizer on a mobile device, one core issue is the memory performance. Therefore, we propose multiple adjustments to make CPU-based depth buffering feasible.

First, the depth buffer has to be chosen small enough to fit into the CPU cache. We chose a size of 320×180 pixels for the buffer, where each pixel can be assigned a 4 byte long floating point number. In addition, an acceleration structure has to be built to allow fast resets of the depth buffer. It is also important to allow the skipping of connected pixels that are unused to reduce the required fill rate during face rasterization.

In order to achieve this, we split the depth buffer into 180 pixel rows, with each row structure managing its own set of 320 pixels. This row is then segmented into 10 blocks of 32 connected pixels each. Two bit vectors are stored per block in addition to the pixel's depth values. The first one indicates which contained pixels are currently in use. The second one stores which pixels were successfully matched with the camera's edge image. An additional bit vector is stored for the whole pixel row. It represents which of the pixel blocks are completely unused. This additional information consumes minimal extra memory, as shown in Figure 4.

3.1.1 Depth Buffer Insertion

During edge processing, each rasterized pixel is processed by the optimized depth buffer algorithm.

The first step is to check whether an entry is already stored at the position of the input pixel. This is achieved by looking at the bit vectors located at the corresponding row in the depth buffer data structure. If an entry already exists, an additional check is performed to determine whether the existing entry is visible or the new one. This decision is made by comparing the individual depth values. If the new entry is invisible, no further action is required and processing can be aborted. Otherwise, the insertion into the depth buffer structure of the old value has to be reverted before writing the new depth value to ensure consistency. The new depth value is then copied to the correct location in the depth value array, and the bit vectors of the buffer structure are updated.

Next, the edge corresponding to the stored pixel is compared to the one in the camera image at the same location. Finally, the corresponding pixel counts necessary for the likelihood evaluation are updated, that is the data which is previously retrieved by relying on Occlusion Queries. By attaching a profiler, we revealed that in many cases, the necessary accesses into the acceleration structure take less time to process than the texture lookup into the camera image.

3.1.2 Hidden Pixel Removal

After all edges have been processed, the rasterized faces are used to detect which edge pixels are invisible due to overlap. Because we rasterize the faces in a row-by-row manner, we always have two points available that describe the bounds of the face on the corresponding row. Everything between the two points is inside the face, everything else is outside the face. The depth values are interpolated linearly between those two points.

To remove hidden pixels, it is necessary to visit all pixels within the range spanned by these two points and compare their depth value to the interpolated one. To perform this task efficiently, we make use of the `CLZ` operation that is part of the ARM instruction set, which is available on most hand-held devices. This instruction is performed in hardware and can, therefore, execute fast and in constant time. Given an arbitrary 32 bit value, `CLZ` calculates the number of bits that are set to 0 before encountering the first 1. If none are set, 32 is returned.

Since we use bit vectors to store information about which pixels are currently in use, we can skip over all cleared bits and stop only at the 1 values, that correspond to stored depth values. These values were previously used during the edge rasterization stage. Because the depth buffer structure is sparse, as only edges were rendered into it, most pixels are skipped that would have been accessed in a nave implementation. Additional speedup is achieved by using the bit vector that contains consolidated information about entire pixel blocks to skip over whole blocks of 32 connected pixels.

4 Results

The tracking algorithm is implemented on an Apple iPad 2 and tested on a stationary cube with different geometric shapes on its sides. The border-lines of the shapes are stored as edges to be tracked. The cube's faces are each split into two triangles and used for hidden edge removal. The edges between the faces are not tracked, as their visibility depends on the lighting conditions. The model consists of 25 edges and 12 triangles. The cube serves as a simple edge-based model, and can be readily replaced with more realistic real-world geometry - because we use depth buffering, objects with hidden lines are also supported.

Tracking is performed at a resolution of 320×180 pixels on a single CPU core. 300 pose estimates are tracked in parallel. 200 of them are taken from the posterior probability distribution of the previous video frame. 100 estimates are repositioned randomly after each frame, without accounting for the tracking history. This allows the re-detection of the object after tracking failure. The overlay consists of a half-transparent colored version of the cube, that is rendered at its detected location. The transparency level describes the tracking accuracy. The more accurate the tracking is, the more opaque the overlay is rendered. This ensures that the overlay becomes invisible when the object is not present in the scene.

One of our tests focuses on the handling of partial occlusions by moving a hand in front of the object (see Figure 2). The tracking algorithm handles this case well and works also, when half of the object is occluded. This leads to the conclusion that only significant edges of an object have to be included in the geometric model,

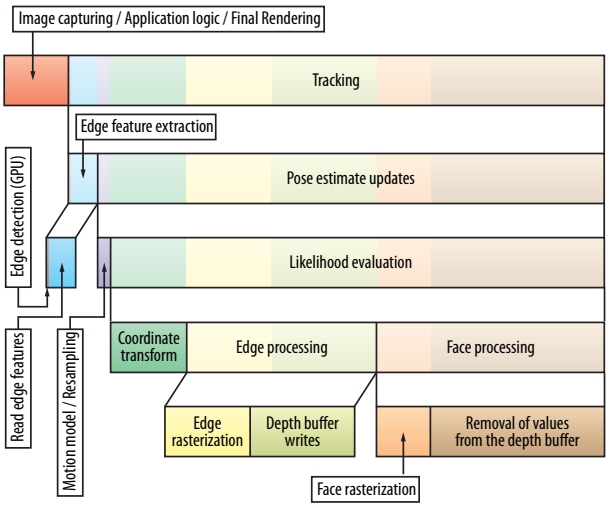


Figure 5: *CPU time distribution over the different functions used in the tracking algorithm.*

allowing a simple model to be used in order to track more complex objects without sacrificing performance. A similar test is performed in a cluttered environment. In most cases, the tracked object's pose is determined quickly, as shown in Figure 1. However, depending on the structure of the extracted edge image, tracking may fail in cluttered environments. When track of the object is lost (for example after inducing fast motion), the recovering process completes often in under 2 seconds.

During the tests, we analyzed the CPU time distribution over the different functions of the algorithm. Only a single CPU core is used during the tests.

After 5 minutes of tracking at a medium distance of around 15 cm, 89.3% of the time is spent in the tracking code and distributed as visualized in Figure 5. 84.4% is used to update the pose estimates. The likelihood evaluation takes up the biggest chunk of 82.3%. 31.6% of the CPU time is spent in edge processing and 38.0% in face processing. This shows that the depth buffer acceleration structure paid off. Without it, much more time would be wasted in face processing as faces consume much more space than edges. The depth buffer structure takes 17.4% for writing operations and 29.0% for the removal process. Only 0.3% make up the edge detection part that runs on the GPU.

The overall distribution over the different parts of the algorithm visualizes, how the algorithm performs with different model complexities. Since most of the time is spent in writing and removing pixels from the depth buffer structure, it should be possible to achieve similar frame rates with higher-complexity models, as long as they have roughly the same size as the tested cube.

Tracking is performed at $60 - 85$ ms per frame, depending on the pixel count of the object's renderings. Since the different likelihood evaluations are independent of each other, multiple threads could be used to process multiple pose estimates in parallel. As an implementation without an optimized depth buffer structure is infeasible, and because the *OpenGL Occlusion Query* extension is not available on current hand-held devices, it is not easily possible to compare directly to other implementations.

5 Conclusion

Augmented reality on mobile devices is in most cases restricted to 2D marker tracking due to the limited processing power. To our knowledge, we describe the first functional edge-based 3D object tracking algorithm, that leaves enough computational power on a mobile device to support advanced applications such as games. Our implementation retargets [Klein and Murray 2006] to consumer devices, as their algorithm provides good results on a desktop computer.

In order to use this approach on a restricted hardware, we introduce multiple extensions. First, we extend the method to incorporate the motion sensors available on the device to detect an accurate approximation for the device's orientation. This leads to the restriction that the tracked object has to be stationary, but allows the algorithm's complexity to be reduced from six degrees of freedom to three degrees of freedom. By including motion sensors into the object itself and transmitting it to the hand-held device, this restriction may be released in the future without increasing its processing power.

Next, we use an efficient rasterization based on sparse depth buffers to cope with the lack of *OpenGL Occlusion Queries* on mobile devices that allow efficient readouts of GPU data back to the CPU. One downside of CPU-based rasterization is, that the size of the tracked object's model is limited. Since the pixels cannot be processed in a highly parallelized manner as on GPUs, run-time increases with the number of rendered pixels. One solution is to reduce the pose estimate count for complex models. However, this also means reducing the tracking accuracy. On the other side, CPU-based approaches allow the use of more flexible algorithms in the likelihood evaluation stage and in the calculation of the motion model, as GPU-based restrictions to programs do not apply.

We prove that current hand-held devices are capable of performing edge-based three-dimensional object tracking. Our solution is also well positioned for the increasing CPU power of upcoming mobile hardware, enabling applications that exploit the key strength of flexible software rasterization methods further.

References

ARMSTRONG, M., AND ZISSERMAN, A. 1995. Robust object tracking. In *Asian Conference on Computer Vision*, vol. I, 58–61.

HARRIS, C. 1993. Tracking with rigid models. In *Active vision*, MIT Press, 59–73.

ISARD, M., AND BLAKE, A. 1998. Condensation - conditional density propagation for visual tracking. *International Journal of Computer Vision 29*, 5–28.

KLEIN, G., AND DRUMMOND, T. 2004. Tightly integrated sensor fusion for robust visual tracking. *Image and Vision Computing 22*, 10 (September), 769–776.

KLEIN, G., AND MURRAY, D. 2006. Full-3d edge tracking with a particle filter. In *Proc. British Machine Vision Conference (BMVC'06)*, vol. 3, BMVA, 1119–1128.

LOWE, D. 1990. Integrated treatment of matching and measurement errors for robust model-based motion tracking. In *Computer Vision, 1990. Proceedings, Third International Conference on*, IEEE, 436–440.

ROSTEN, E., AND DRUMMOND, T. 2005. Fusing points and lines for high performance tracking. In *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*, vol. 2, IEEE, 1508–1515.