Image-space correction of AR registration errors using graphics hardware

Stephen DiVerdi* Tobias Höllerer†

Department of Computer Science
University of California, Santa Barbara, CA 93106

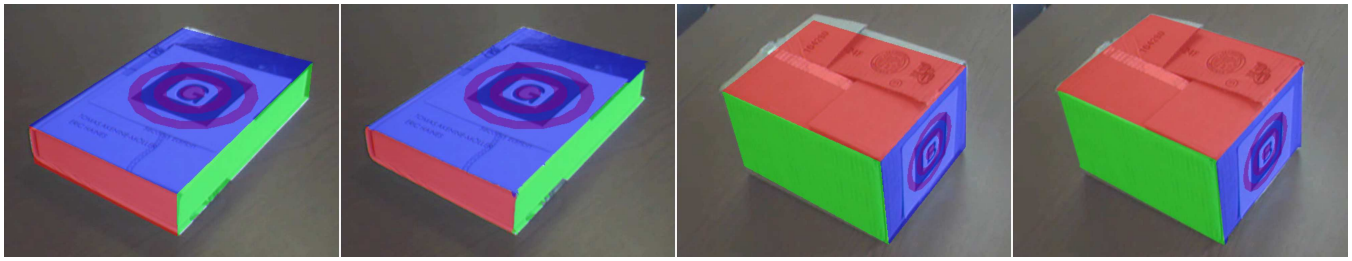

Figure 1: *Left pair:* An input set of polygons, and the corrected result. Note the modeling corrections around the perimeter of the model. *Right pair:* Another input set of polygons, and the corrected result. Because of the weaker intensity edges, smoothing is enabled.

ABSTRACT

Many Mixed Reality applications rely on drawing virtual imagery directly on top of physical objects in a video scene. Registration accuracy is a serious problem in these cases since any imprecisions are immediately apparent as virtual and physical edges and features coincide.

We present a hardware-accelerated image-based post-processing technique that adjusts rendering of virtual geometry to better match edges present in images of a physical scene, reducing the visual effect of registration errors from both inaccurate tracking and over-simplified modeling. We detect intensity edges in an image of the scene captured by a camera, and search for these edges around the boundary of projected polygons. These detected edges are used to clip the boundaries of the rendered polygons, making virtual geometry edges match strong image features.

Our algorithm is easily integrable with existing AR applications, having no dependency on the underlying tracking technique. We use the advanced programmable capabilities of modern graphics hardware to achieve high performance without burdening the CPU.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Display Algorithms I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtuality Reality I.4.6 [Image Processing]: Segmentation—Edge and Feature Detection

Keywords: image-based technique, hardware accelerated, edge detection, registration

1 INTRODUCTION

The progress of computer technology in recent years has been very helpful to the AR community, making numerous improvements in geometric registration possible. However, even with these improved techniques, registration is still a major problem in most AR applications, seriously hurting the sense of integration between virtual and physical worlds. These errors continue to exist because accurate tracking of motion and accurate modeling of real geometry are two difficult problems that require laborious calibration and are prone to errors. Registration errors are especially noticeable and problematic when virtual and physical features should coincide, such as when virtual geometry is drawn directly on top of physical objects to affect its appearance. Applications that make use of such overlaid virtual geometry include, but are not limited to, highlighting of an object using wireframe outlines or colored polygons [16], halo glow around objects [11], re-texturing of physical objects, and re-lighting [25, 8], which requires alpha blending of additive or subtractive light contributions directly onto the physical geometry.

In order to correct registration errors to pixel accuracy, augmented reality tracking often makes use of image analysis techniques. One serious difficulty with such an approach is the integration of the corrections with the specific tracking technology used. Ideally, in AR applications, tracking should be kept separate from the application content, and should be modularized, so that one tracking technology (e.g. marker-based tracking) may be easily substituted with another one (e.g. ultrasound tracking). Image-based corrections do not depend heavily on the used tracking technology, and therefore should be applied separately, so that the we can avoid the necessity for developing a new hybrid tracker every time a new tracking technology is introduced.

In this paper, we present a technique that can be applied to AR applications, regardless of tracking technology, to consistently reduce visual registration errors of overlaid virtual geometry. The basic assumption of our technique is that geometric edges in an AR scene model correspond with edges in the physical world. We then detect the edges in an acquired video of the scene. For each edge in our virtual model, we search a region determined by a per-vertex tracking-error estimate (provided by the application) for strong nearby edges, and then smooth the detected edges. Finally, the original model polygons are rendered, clipped against the detected edges so they approximate the video features. The result is virtual objects that more closely match strong features in the physical scene, and experience less jitter in their positions. Both tracking errors and modeling errors will be reduced (see Figure 1).

Our results depend on the quality of the edge detection that is possible. High contrast environments will yield the best improve-

*e-mail: sdiverdi@cs.ucsb.edu
†e-mail: holl@cs.ucsb.edu

ments, but even when contrast is low and video features are weak, we still achieve comparable or better visual quality. It is also important to note that we do not modify the tracking result, or perform a rigid transformation of any sort. Our technique is an image space warp that is performed as a post-processing step, which is fundamentally different from techniques which use image edges as part of the tracking computation. This way, it is generally applicable to a wide variety of AR tracking technologies, with much easier integration than would otherwise be possible. The use of vertex and pixel shaders of modern computer graphics cards ensures that the AR applications still run at similar speeds as the uncorrected versions.

The rest of the paper is organized as follows. We discuss related work in Section 2. Our basic algorithm is presented in Section 3, and the details of our implementation are described in Section 4. The results and limitations are presented in Section 5, and we conclude in Section 6.

2 Related Work

Computer vision based tracking is an actively researched area in the augmented reality community. A wide variety of different techniques have been used to determine position and orientation information from available image data. Common algorithms are based around tracking simple image features, such as edges, corners, or textures [13, 15, 5, 4]. These features can be combined for improved tracking [27], or more complex information can be used such as scale invariant features [23], fiducial markers [9], reference images [24], or even terrain data [1]. Some of these techniques require a model of the tracked scene as input, while others can recover a scene model during the tracking, or are posed entirely as 2D correspondence problems with no knowledge of the scene's 3D geometry. All of these algorithms use image analysis as the sole source of tracking information, while we use image information to improve an already computed tracking result. In fact, our algorithm could be applied on top of another vision based tracker (as we have done, using the ARToolKit [20]), making integration easier since a video of the scene is already acquired.

Relying on any one tracking technology is often undesirable, so computer vision is also frequently used in hybrid tracking systems to support or be supported by other trackers. It is common to use inertial tracking, combined with some form of vision tracking, such as tracked texture features or markers [30], or reference image matching [14]. There is even work on modular hybrid systems which dynamically adapt to new tracking inputs at runtime, such as OpenTracker [22] or Ubitrack [19]. Our technique is also a modular approach, but rather than integrate the image edge data with other tracking technologies, we apply the correction as a post-proessing step. This is more general, as it does not need to be specifically integrated into other static or dynamic hybrid systems.

Most similar to our paper is the static hybrid approach of using image edge information to correct an inertial tracking result [12], and even to refine local occlusion boundaries, with very good results. Ours differs in a number of ways. By correcting polygon boundaries per pixel rather than adjusting the pose estimate for the entire polygon, we can adapt to modeling errors that arise from representing a complex physical edge with a straight polygon edge (see Figure 6). Additionally, Klein's technique is designed for a static physical scene, whereas we can handle many independently moving physical objects at no additional cost. Finally, we take advantage of hardware acceleration to avoid relying on a software renderer, easing the burden on the CPU for other application tasks.

Currently, graphics hardware is being used more and more for non-3D graphics computations, including a large variety of image processing and computer vision techniques. Discrete wavelet transformations [28], FFTs [18], image segmentation [29], feature (edge and corner) detection [6], and stereo matching [26] can all be performed with hardware acceleration. These results have even been applied to the field of augmented reality, using the GPU to directly perform all the necessary feature tracking and pose estimation steps to draw virtual objects situated in the physical environment [7]. Again, we differ in that our result is not a tracking improvement, but a post-processing filter that adjusts rendering to improve visual quality. GPU processing is not particularly well suited to tasks that involve significant communication between GPU and CPU, as most computer vision algorithms do. Our approach, on the other hand, is tailored to the particular capabilities of the GPU, doing final processing in hardware without readback.

Finally, MacIntyre and colleagues have worked to address the effects of registration errors from inaccurate tracking by propagating these errors on to geometry, and then modifying the presented visual and interface appropriately [3, 17]. These results are important for our technique to provide the per-vertex tracking error estimate we require to determine edge search regions.

3 Technique

Our technique acts as a per-frame post-processing step, after the tracking and animation components of an AR application have finished computing each frame. We take the final geometry for each frame and render it with our algorithm to create the better matched image. The algorithm itself is a series of discrete steps, each of which processes the video and geometry towards the final goal. This modular design allows easy adaptation and modification of the algorithm by changing isolated components individually.

The inputs to our algorithm are: a list of quad polygons, a list of per-vertex position error estimates, and an image of the physical scene. The following steps are each computed, and the output is a visual of the virtual scene with improved geometric registration (see Figure 2).

1. Perform edge detection on scene image.
2. Search edge image within polygon error regions for strong, similar edges.
3. (optional) Smooth individual detected edges.
4. Render original polygons, clipped to detected edges.

A few requirements must be met for our algorithm to be applied to an AR application. First, per-vertex estimates of tracking error must be provided. Every tracker has some error associated with its measurements, in both position and orientation. These errors can be propagated through a series of transformations to provide a position and orientation error of a local coordinate system, which can then be applied to individual vertices to determine a region of the screen where a pixel may exist [3, 17]. These errors are used to determine the search regions for step 2. However, the underlying tracking technique is unimportant – all that is needed is an estimate of its error.

Second, an image of the scene must be made available for step 1, in the form of a texture map. This means the AR application must acquire a video and load each frame to the graphics card's texture memory before our algorithm can execute. Finally, the geometry to be rendered currently must be provided as a list of quad polygons with associated modelview and projection matrices. An extension of the algorithm to handle triangle lists is straightforward.

Since the output consists of only the rendered virtual geometry, our technique will work for both video and optical see-through AR applications. However, optical see-through will require careful calibration of the acquired video with the user's field of view, so the edges from step 1 actually correspond to the edges the user sees.

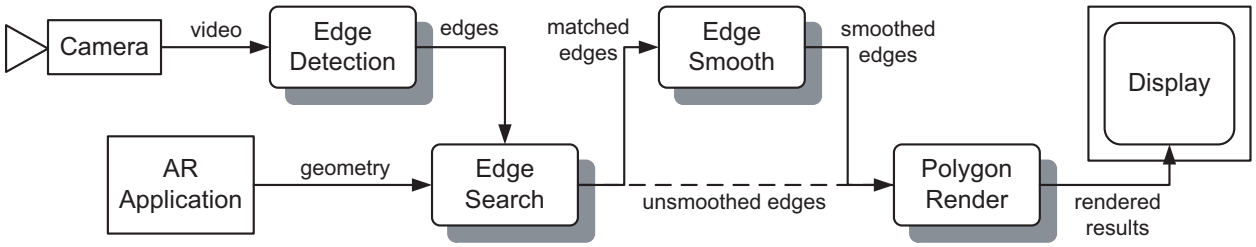

Figure 2: Flow diagram of algorithm. The inputs are an image of the scene and the virtual geometry. Edges are extracted from the image, and matched with the geometry's edges. The matched results are optionally smoothed, and the original geometry is then rendered, clipped to the detected edges.

4 Implementation

Each step of our technique raises particular implementation questions that are important to examine. In general, we take advantage of new programmable graphics hardware by writing each step as a rendering pass with appropriate geometry, textures and shaders. By using shaders for all passes and keeping image data in texture memory, we remove the need to ever pass back image data from graphics memory to main memory, which is a common speed bottleneck of GPU programming. It is also important to note that each step is applied to all the polygons before continuing to the next step, so the number of passes needed is the same regardless of the amount of geometry.

4.1 Edge Detection

We rely on established edge detection algorithms for the first step of our technique. Our default algorithm is a GPU implementation of a 3x3 Sobel filter – a fragment shader samples the texture in the kernel and outputs a color encoding the gradient direction and magnitude. This is our standard choice for reasons of speed, but Sobel filtering suffers from the lack of any edge continuity enforcement, so edges can become fragmented easily.

An alternate choice is OpenCV's Canny function [10], which is a standard CPU implementation of the algorithm [2] (GPU implementations are now available as well [6], but we were unable to integrate them with our system). While Canny edge detection does do a better job in general of finding exact edges by enforcing an edge continuity constraint, the output is only a binary value, lacking any information about edge strength or orientation. The results are also very sensitive to the appropriate threshold values, which require tuning from scene to scene, and possibly even within a scene, if there is a significant change in lighting. Theoretically, it should be possible to obtain better results with Canny than Sobel given proper tuning, but we have been unable to achieve such results.

4.2 Edge Searching

The second step is to search the edge image for strong edges near each polygon edge. First, back-facing polygons are culled, as they should not be visible in the scene image. Per polygon, we project the vertices to screen coordinates and use the vertex error estimate to determine a region of the screen where the vertex may lie (the dotted circles in Figure 3). Vertices connected by an edge then define a search region (roughly the convex hull of the two vertex regions, shown in red in Figure 3) in which the edge can be found. The interior of these border search regions is guaranteed to be part of the polygon regardless of the vertices' actual positions within the error estimates (the blue region in Figure 3).

Once the search regions have been determined, each edge is rendered as a line, with our *search shader* activated and the edge image as a texture input. Each line has its search region defined in texture coordinates. For each pixel on the line, the search shader walks along a perpendicular line from the inside to the outside of the search region (see Figure 4). Near corners, these lines extend radially from the internal region's corner. At each step, it samples the edge image, retrieving that pixel's edge magnitude and orientation. A weighting function is applied to these samples, and the maximum weighted sample encountered is tracked. Once the outside boundary of the search region is reached, the position of the maximum weight sample is encoded as an offset vector in the red and green channels of the pixel color.

$$w = s * \frac{d}{d_{max}} * |v_g \cdot v_s| \tag{1}$$

The weighting function, equation (1), takes into account all the information that is known about a pixel in the edge image. Sample weights are the product of the edge strength s, a distance term, and an orientation term. The distance term linearly weights the sample based on distance from the input polygon edge. The orientation term is the absolute value of the dot product of v_g, the gradient vector, and v_s, the search direction vector – if the two vectors are parallel or antiparallel, then the detected edge and the polygon edge are similarly oriented. The final result is a weight value between 0 and 1, representing the likelihood of the particular sample.

4.3 Edge Smoothing

The detected edges output from step 2 of our algorithm are often quite noisy – that is, there are frequent discontinuous jumps between adjacent offsets, which create a "frayed" appearance of the detected edge (see Figure 5). This noise flickers rapidly among frames, creating a displeasing visual artifact. One possible way to address this problem would be to improve the edge searching algorithm. Instead, we chose to implement an edge smoothing filter as a separate step. This is faster than more robust edge searching, and

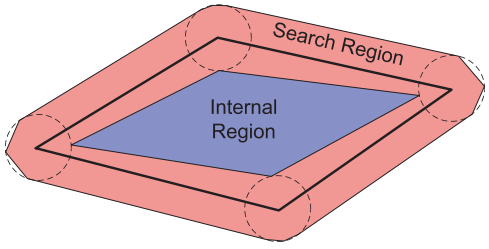

Figure 3: An input polygon is drawn in thick black. The dotted circles around each vertex shows the tracking error estimates. The blue region is guaranteed to be part of the polygon, while the red region is uncertain.

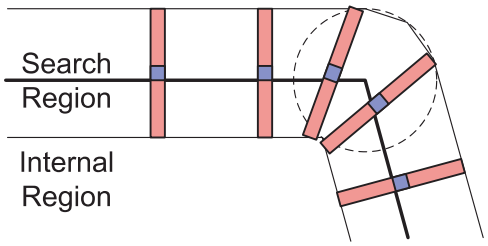

Figure 4: For each pixel along the input polygon edge (shown in blue), a perpendicular line of pixels (shown in red) is searched for edges in the image. At the corner, these search lines extend radially from the internal corner.

allows the smoothing filter to be replaced with different algorithms, or removed entirely depending on the application's needs.

The smoothing step is applied by rendering the polygon edges as lines again, with the *smoothing shader* activated and the detected edge offsets from step 2 as a texture input. For each polygon edge, the shader walks the entire edge, taking regular samples of the detected edge offset information. A running sum of offsets is kept, to determine the mean offset at the end of the pass. Additionally, a measure of the 'noisiness' of the edge is calculated for each sample and accumulated over the edge. Once the walk is complete, the aggregate noisiness measure is examined – if it exceeds a user-specified threshold, the edge is determined to be too noisy, and each offset value in the detected edge is replaced by the mean offset for the entire edge. Otherwise, if the edge is not too noisy, the original offsets are maintained. The mean value is used instead of a more appropriate measure such as the median, due to the difficulty of implementing an efficient n-element median algorithm on a stream architecture (the graphics hardware) with limited temporary storage capabilities.

The noisiness measure is based on the second derivative of the detected edge offsets. At each sample, a 1x3 Laplace filter ($[-1, 2, -1]$) is applied to the sample and its neighbors, to compute the second derivative of the detected edge at that point. The mean of the absolute values of the second derivatives is the 'noisiness' of the detected edge.

If the smoothing operator were implemented in a fragment shader, for each fragment, it would need to sample the entire edge and recalculate the mean and noisiness. For an edge of 100 pixels, this means 10,000 textures samples, which is clearly a waste of processor cycles. We avoid this level of redundancy by implementing the smoothing filter in a vertex shader, which is then executed per-vertex, or twice per edge (since vertex shader outputs are interpolated across the line, the same result must be computed at both vertices). The results are passed to the fragment shader via the vertices' output colors, which are shared among the fragments. This is made possible by using the latest programmable shader capabilities made available in NVIDIA's 6000 series GPUs - specifically, we need support for vertex texture sampling. In absence of these graphics hardware capabilities, the smoothing step can be omitted, or it can be done more slowly in a fragment shader as discussed.

4.4 Rendering

After the previous three steps, we have a smoothed detected edge image, which encodes the per-pixel offset of the detected edge from each polygon's original edges. To render the newly deformed polygon, first, the internal region is rendered normally. Then, the unknown border regions are rendered with the *clipping shader* and the detected edge texture as input. In this final pass, each fragment samples the detected edge image to determine where the polygon's real edge is. Then the pixel compares its position to the detected edge – if it is outside the detected edge (determined by the inside and outside boundaries of the search region), its alpha value is set to zero, but if it is inside, its alpha is kept at its original value. Near the detected edge, alpha values are dropped off linearly for a smooth polygon border. Since each rendered pixel does only one texture sample, this pass is very fast.

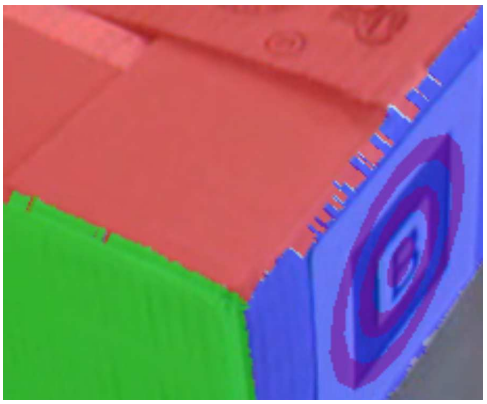

Figure 5: An example of the detected edge results, before smoothing. The red-blue edge shows severe fraying due to the low contrast image edge – global smoothing is necessary. The red-green edge has much more isolated noise and would benefit from localized smoothing. The blue-green edge is an ideal result of the detection.

Alternately, if a wireframe rendering is desired rather than full polygons, this final pass can be altered very slightly to accommodate. The internal regions of each polygon are left out, but the unknown border regions are still drawn the same as before. However, in this case the shader will set the alpha value of pixels farther than a set width from the detected edge to zero. Pixels near the detected edge are kept, and alpha values are smoothly dropped off at the boundary for an antialiased line.

Some filtering of the detected edge image can also be done in this step. Rather than sampling one pixel for the edge offset, its neighbors can be sampled as well, and the multiple values are combined to compute the final offset. We implemented support for 1x3, 1x5 and 1x7 block filters, as well as a 1x3 median filter. As graphics hardware is designed for this type of filtering, it does not significantly affect performance.

5 Results

We present the achieved results of our algorithm, including rendering performance and visual quality of registration correction. We also discuss the limitations of our current implementation.

5.1 Performance

For testing, we have integrated our technique with a simple AR application that uses the ARToolKit [20] for marker-based tracking, to overlay a virtual model of a box on top of a physical, tracked box. The ARToolKit is prone to small errors in orientation estimates, which propagate to large translational errors for vertices that are far from the marker's center. With an NVIDIA GeForce FX 6800GT graphics card, we experience a modest 8.2% drop in framerate, from 61 to 56 frames per second, when processing a pre-recorded 640x480 video stream, using all four steps of our technique, with a 1x3 block filter in the fourth step.

Figure 6 shows a comparison of the original polygon edge, the unsmoothed detected edge, and the smoothed detected edge. The original edge clearly shows registration errors – because of tracking

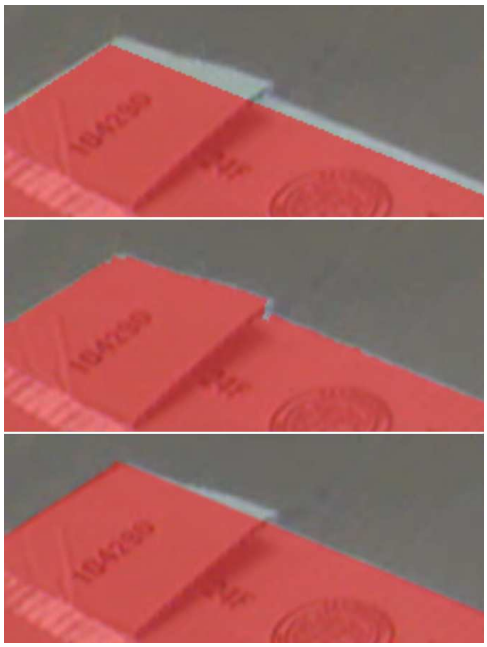

Figure 6: Comparison of results. *Top to bottom:* The original polygon edge, our detected edge, and the smoothed detected edge.

error, the polygon edge is moved away from the physical edge, and because of modeling error, the complex shape of the physical edge is represented by the straight edge of a polygon. Since this is a high contrast image region, the detected edge matches the physical edge nearly perfectly, correcting both the tracking and modeling error. The smoothed edge loses the ability to correct the modeling error, but is still a marked improvement over the original edge, due to the tracking correction.

The quality of the unsmoothed edge depends heavily on the edge detection result. If the edge detection step finds a clear, strong edge within the search region of the polygon, then the detected edge will be match the image's edge very closely, with very little noise. Unfortunately, in real environments with complex lighting and low dynamic range video acquisition, low contrast images are commonplace, making good edge detection difficult. In these cases, the unsmoothed edge result will be noisy and will jitter from frame to frame, so the smoothed edge result will be more appropriate. While the smoothed edge may not match the physical edge as closely, it will be jitter less between frames and be a closer match than the original polygon edge, improving the visual quality.

5.2 Discussion

The nature of the design choices we made to implement this technique leads to some limitations which are important to examine. Foremost is the fact that we do not modify the tracking result, just the rendering of the polygons. This means that we are not using a hybrid vision-based tracking system, and we cannot correct drift from other tracking technologies. It also means that polygons are clipped, rather than moved, so decal textures will not be shifted with the polygon's position. If text is texture mapped onto a polygon, the edge of the text will be clipped off, rather than all of the text being warped slightly. On the other hand, effects such as per-pixel lighting will not suffer from this effect, as they are calculated uniquely for each pixel.

Currently, our per-vertex tracking error estimation is a single float value, which represents a screen pixel radius. This is a simplistic assumption, as errors are more likely to be ellipses in screen space. The downside of our simplification is that our search regions may be too wide, decreasing performance unnecessarily and possibly getting distracted by edges that are farther away (though that error is minimized by penalizing distant detected edges in the search shader's weighting function). Of course, we also assume the reliability of the tracking error estimate. In the event that the error is over-estimated, time will be wasted searching larger regions, and edges may become distracted more easily (though weighting detected edges by distance reduces this effect). Conversely, if the error is under-estimated, as the edge search step may not find an edge in the search region, in which case the original input polygon edge is used. This way we make sure our result is always at least as good as the input.

The global nature of our detected edge smoothing can be limiting as well. In the case that a detected edge is determined to be too noisy and is replaced by the mean offset edge, the result is a straight line. This means any per-pixel corrections of modeling error will be lost to avoid the noise. Clearly, it would be preferable to locally smooth the edge in a feature-preserving way, to keep the important detected edge features, but lose the distracting noise. One way we have addressed this issue in our current implementation is by allowing the noisiness threshold to be specified per-edge by the user. This way the user can judge the modeling error for an edge and determine about how much noise should be tolerated before resorting to a straight-line approximation. We find that in our test cases, the edge noise is a distracting enough artifact that we always set the noisiness threshold to zero, forcing smoothing on every edge.

Finally, in cases where polygons are viewed from a shallow angle, or when the tracking error is very large, our algorithm will fail because there will be no known internal region of the polygon, and the search regions for opposite edges will overlap. This can cause confusion with detected edges and can result in non-convex polygons. To avoid this problem, when the internal polygon has negative area (by calculating area assuming counter-clockwise vertex ordering), our technique bails out and the original, unmodified polygon is drawn instead. This avoids visual artifacts and assures that our result is as least as good as the unmodified input.

6 Conclusions

We have presented a general post-processing technique for reducing the visual effects of registration error, both from inaccurate tracking and oversimplified modeling. The algorithm is applicable for a class of AR applications which modify the appearance of physical objects by overlaying corresponding virtual geometry on top of them, such as physical object selection or re-lighting. Our algorithm does not depend on the underlying tracking technology and is easily integrated with most AR applications. In environments with strong edges in the physical scene, our algorithm will considerably improve the matching of polygons to video edges, within a few pixels. Even in scenes with low contrast and poor edges, our fallback options yield improved stability and closer matching to video edges. We take advantage of advanced graphics hardware to provide these results with minimal impact to application performance.

Our approach also suggests many interesting avenues for future work. Depending on the particular domain of an AR application, the edge detection step of our algorithm could be improved with domain-specific edge detection techniques. For example, structured light, infrared spectrum, or stereo vision could all provide useful edge information. A real-time version of the non-photorealistic camera [21] could provide the necessary edge information reliably as well. We wouldd also like to develop better detected edge smoothing techniques such as a sliding window for local smoothing, to reduce noise while still preserving edges, allowing for better, more general correction of modeling errors. Finally, we are inves-

tigating ways to read back the resulting polygon deformation to the CPU, so that the AR application can make use of our results as feedback to the tracking (or even model) information.

REFERENCES

[1] R. Behringer. Registration for outdoor augmented reality applications using computer vision techniques and hybrid sensors. In *Proceedings of IEEE Virtual Reality*, pages 244–251, March 1999.

[2] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, 1986.

[3] E. Coelho, B. MacIntyre, and S. Julier. OSGAR: A scene graph with uncertain transformations. In *International Symposium on Mixed and Augmented Reality*, November 2004.

[4] A. Comport, E. Marchand, and F. Chaumette. A real-time tracker for markerless augmented reality. In *Proceedings of the International Symposium on Mixed and Augmented Reality*, pages 36–45, October 2003.

[5] A. Davison. Real-time simultaneous localisation and mapping with a single camera. In *Proceedings of the International Conference on Computer Vision*, October 2003.

[6] J. Fung and S. Mann. Using multiple graphics cards as a general purpose parallel computer : Applications to computer vision. volume 1, pages 805–808, 2004.

[7] J. Fung, F. Tang, and S. Mann. Mediated reality using computer graphics hardware for computer vision. In *Proceedings of the International Symposium on Wearable Computing 2002*, pages 83–89, October 2002.

[8] S. Gibson, J. Cook, T. Howard, and R. Hubbold. Rapid shadow generation in real-world lighting environments. In *Proceedings of the 14th Eurographics workshop on Rendering*, pages 219–229, 2003.

[9] W. Hoff, T. Lyon, and K. Nguyen. Computer vision-based registration techniques for augmented reality. In *The Proceedings of Intelligent Robots and Control Systems XV, Intelligent Control Systems and Advanced Manufacturing*, volume 2904, pages 538–548, November 1996.

[10] Intel Corporation. *Open Source Computer Vision Library Reference Manual*. December 2000.

[11] G. James and J. O'Rorke. Real-time glow. In *GPU Gems*, 2004.

[12] G. Klein and T. Drummond. Sensor fusion and occlusion refinement for tablet-based ar. In *Proceedings of the International Symposium on Mixed and Augmented Reality*, pages 38–47, October 2004.

[13] D. Koller, G. Klinker, E. Rose, D. Breen, R. Whitaker, and M. Tuceryan. Real-time Vision-Based camera tracking for augmented reality applications. In *ACM Symposium on Virtual Reality Software and Technology*, September 1997.

[14] M. Kourogi and T. Kurata. Personal positioning based on walking locomotion analysis with self-contained sensors and a wearable camera. In *Proceedings of the International Symposium on Mixed and Augmented Reality*, pages 103–112, October 2003.

[15] J. Lee, S. You, and U. Neumann. Tracking with omni-directional vision for outdoor AR systems. In *Proceedings of the International Symposium on Mixed and Augmented Reality*, pages 47–56, September 2002.

[16] M. Livingston, J. Swan II, J. Gabbard, T. Höllerer, D. Hix, S. Julier, Y. Baillot, and D. Brown. Resolving multiple occluded layers in augmented reality. In *Proceedings of the International Symposium on Mixed and Augmented Reality*, pages 56–65, October 2003.

[17] B. MacIntyre and E. Coelho. Adapting to dynamic registration errors using level of error (LOE) filtering. In *International Symposium on Augmented Reality*, October 2000.

[18] K. Moreland and E. Angel. The FFT on a GPU. In *Proceedings of the ACM conference on Graphics hardware*, pages 112–119, 2003.

[19] J. Newman, M. Wagner, M. Bauer, A. MacWilliams, T. Pintaric, D. Beyer, D. Pustka, F. Strasser, D. Schmalstieg, and G. Klinker. Ubiquitous tracking for augmented reality. In *Proceedings of the International Symposium on Mixed and Augmented Reality*, pages 192–201, 2004.

[20] I. Poupyrev, D. Tan, M. Billinghurst, H. Kato, H. Regenbrecht, and N. Tetsutani. Developing a generic augmented-reality interface. *Computer*, 35(3):44–50, March 2002.

[21] R. Raskar, K. Tan, R. Feris, J. Yu, and M. Turk. A non-photorealistic camera: depth edge detection and stylized rendering using multi-flash imaging. In *Proceedings of ACM SIGGRAPH*, August 2004.

[22] G. Reitmayr and D. Schmalstieg. Opentracker-an open software architecture for reconfigurable tracking based on XML. In *Proceedings of IEEE Virtual Reality*, pages 285–286, 2001.

[23] I. Skrypnyk and D. Lowe. Scene modelling, recognition and tracking with invariant image features. In *Proceedings of the International Symposium on Mixed and Augmented Reality*, pages 110–119, November 2004.

[24] D. Stricker and T. Kettenbach. Real-time and markerless vision-based tracking for outdoor augmented reality applications. In *Proceedings of the International Symposium on Augmented Reality*, pages 189–190, October 2001.

[25] N. Sugano, H. Kato, and K. Tachibana. The effects of shadow representation of virtual objects in augmented reality. In *Proceedings of the International Symposium on Mixed and Augmented Reality*, pages 76–83, October 2003.

[26] K. Sugita, T. Naemura, and H. Harashima. Performance evaluation of programmable graphics hardware for image filtering and stereo matching. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 176–183, 2003.

[27] L. Vacchetti, V. Lepetit, and P. Fua. Combining edge and texture information for real-time accurate 3d camera tracking. In *Proceedings of the International Symposium on Mixed and Augmented Reality*, pages 48–57, November 2004.

[28] J. Wang, T. Wong, P. Heng, and C. Leung. Discrete wavelet transform on gpu. In *Proceedings of ACM Workshop on General Purpose Computing on Graphics Processors*, pages C–41, August 2004.

[29] R. Yang and G. Welch. Fast image segmentation and smoothing using commodity graphics hardware. *Journal of Graphics Tools*, 7(4):91–100, 2002.

[30] S. You, U. Neumann, and R. Azuma. Hybrid inertial and vision tracking for augmented reality registration. In *Proceedings of IEEE Virtual Reality*, pages 260–267, 1999.