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

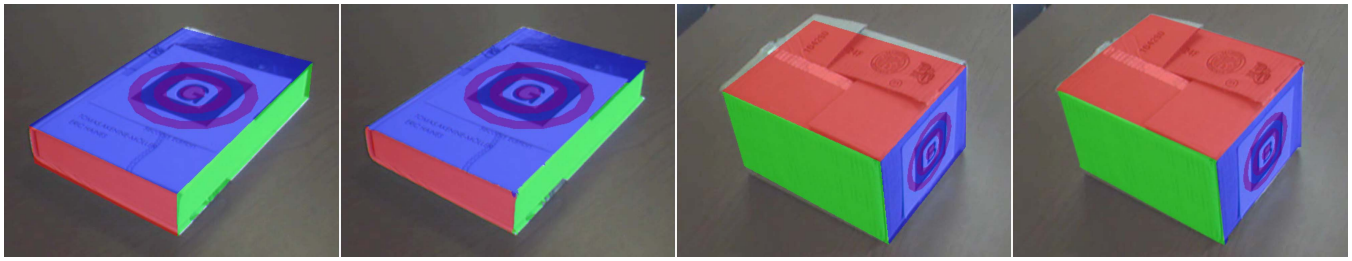

Figure 1: *Left pair:* An input set of polygons, and the corrected result. Note the modeling corrections around the perimeter of the model. *Right pair:* Another input set of polygons, and the corrected result. Because of the weaker intensity edges, smoothing is enabled.

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 oversimplified modeling. 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.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtuality Reality I.4.9 [Image Processing]: Applications

Keywords: image-based techniques, hardware acceleration, 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, registration remains a major problem in most AR applications, seriously hindering 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 problematic when virtual and physical features should coincide, such as when virtual geometry is drawn directly on top of a physical object 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, halo glow around objects, re-texturing of physical objects, and re-lighting, which requires alpha blending of additive or subtractive light contributions directly onto the physical geometry.

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 are reduced (see Figure 1). 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.

It is 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 easier integration than would otherwise be possible.

2 Related Work

Computer vision based tracking is an actively researched area, with a wide variety of different techniques. Common algorithms are based on tracking simple image features, such as edges, corners, or textures [9, 4, 3], or more complex information such as fiducial markers [7] and reference images [13]. 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 [11]), making integration easier since a video of the scene is already acquired.

Computer vision is also frequently used in hybrid tracking systems, combined with other techniques such as inertial tracking [15]. There is even work on modular systems which dynamically adapt to new tracking inputs [12]. Our technique is also a modular approach, but we apply the correction separately, as a post-processing step. This is more general, as it does not need to be specifically integrated with other tracking systems.

Most similar to our paper is Klein's approach [8], using image

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

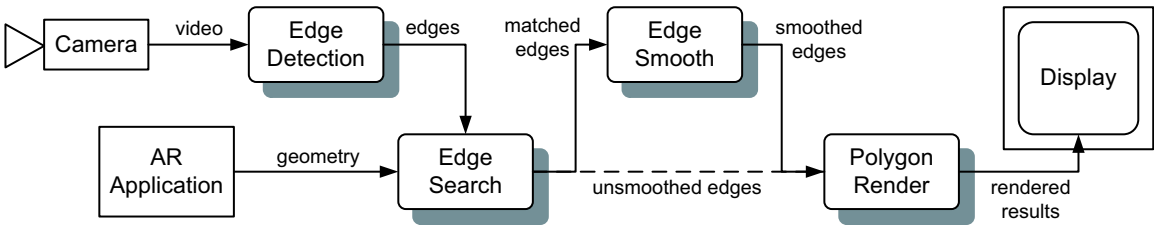

Figure 2: Flow diagram of algorithm. The inputs are a scene image and virtual geometry. Edges are extracted from the image and matched with the geometry's edges. The matched results are optionally smoothed, and the geometry is then rendered, clipped to the detected edges.

edges to correct an inertial tracking result and refine local occlusion boundaries. Our algorithm 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, easing the burden on the CPU for other application tasks.

Currently, graphics hardware is being used for many image processing and computer vision techniques. Image segmentation [14] and feature (edge and corner) detection [5] can both 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 feature tracking and pose estimation steps to draw virtual objects in the physical environment [6]. However, 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 only final processing in hardware, minimizing communication.

3 Technique

Our technique acts as a per-frame post-processing step, after the tracking and animation components of an AR application have finished. We take the final geometry for each frame and render it with our algorithm to create a better matched image. 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, and 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. Tracker errors can be propagated through a series of transformations to provide the 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 vertex may exist [2, 10]. These errors are used to determine the search regions for step 2. 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 in the form of a texture map. Third, the geometry to be rendered must be provided as a list of polygons with associated modelview and projection matrices (currently, quads are required, but an extension to handle triangles 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 image edges actually correspond to the edges the user sees.

4 Implementation

We take advantage of programmable graphics hardware by writing each algorithm step as a rendering pass with appropriate geometry, textures and shaders. By using shaders for all passes and keeping temporary data in texture memory, we remove the need to pass back data from the GPU to the CPU, 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 default because of its speed, but Sobel filtering suffers from the lack of any edge continuity enforcement, so edges can become fragmented easily.

An alternate choice is a CPU implementation of the Canny edge detection algorithm [1] (a GPU implementation is available [5], but we were unable to integrate it with our system). However, Canny results are very sensitive to the appropriate threshold values, which require tuning from scene to scene and possibly even within a scene. 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. 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). Adjacent vertices 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 the 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. Search regions are defined in texture coordinates. The search shader walks perpendicular to the edge, or near corners, radially from the internal region's corner. It samples the edge image

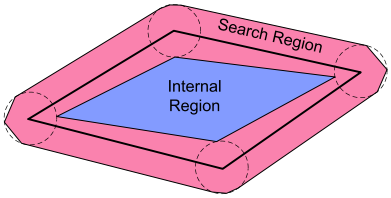

Figure 3: An input polygon is drawn in thick black. The dotted circles around each vertex shows the tracking error estimates. The blue region is guaranteed to be part of the polygon, while the red region is uncertain.

along this path, retrieving that pixel's edge magnitude and orientation. A weighting function is applied to these samples, and the position of the maximum weighted sample is encoded as an offset vector in the red and green channels of the output color.

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

Sample weights are the product of the edge strength s, a distance term, and an orientation term (Equation 1). 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 are often quite noisy – that is, there are frequent discontinuous jumps between adjacent offsets, which create a "frayed" appearance (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 a separate edge smoothing step. This is faster than more robust edge searching and allows the smoothing filter to be modified or removed depending on application 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 takes regular samples of the detected edge offset. 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. Once the walk is complete, the noisiness measure is examined – if it exceeds a user-specified threshold, the edge is too noisy and each offset value is replaced by the mean offset. Otherwise, the original offsets are kept. The mean value is used instead of a more appropriate measure such as the median due to the lack of efficient n-element median algorithms for graphics hardware.

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. The mean of the absolute values of the second derivatives is the 'noisiness' of the detected edge.

We implement the smoothing filter in a vertex shader, which is 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.

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 and compares its position to the sample – if it is outside (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 unchanged. Near the detected edge, alpha values drop off linearly for a smooth polygon border. Since each rendered pixel does only one texture sample, this pass is very fast.

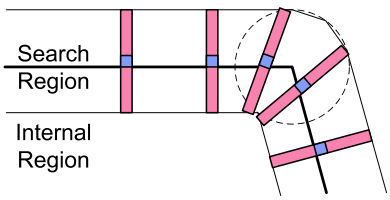

Figure 4: For each pixel along the input polygon edge (shown in blue), a perpendicular line of pixels (shown in red) is searched for edges in the image. At the corner, these search lines extend radially from the internal corner.

Alternately, this final pass can be slightly altered to accommodate wireframe rendering. The polygons' internal regions are omitted, and the clipping shader sets pixels' alpha values based on the distance from the detected edge, for an antialiased line.

Some filtering of the detected edge image can also be done. Rather than sampling one edge offset, its neighbors can be sampled as well 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 rendering performance and visual quality of registration correction, as well as the limitations of our current implementation.

5.1 Performance

For testing, we integrated our technique with a simple AR application that uses the ARToolKit [11] to overlay a virtual model of a box on top of a physical, tracked box. The ARToolKit is prone to small errors in orientation that 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 detected edge, and the smoothed edge. The original edge clearly shows registration errors – because of tracking 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 a straight line. 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.

The quality of the detected edge depends heavily on the edge detection result. If the edge detection step finds a clear, strong edge

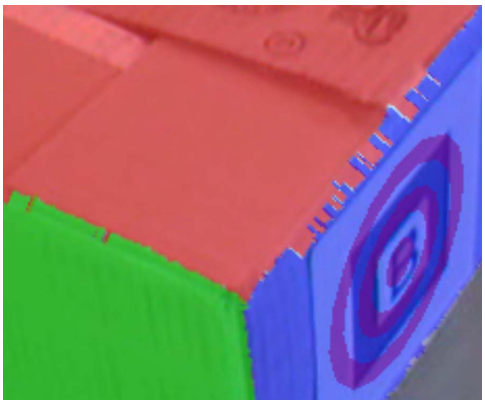

Figure 5: An example of the detected edge results, before smoothing. The red-blue edge shows severe fraying due to the low contrast image edge – global smoothing is necessary. The red-green edge has much more isolated noise and would benefit from localized smoothing. The blue-green edge is an ideal result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

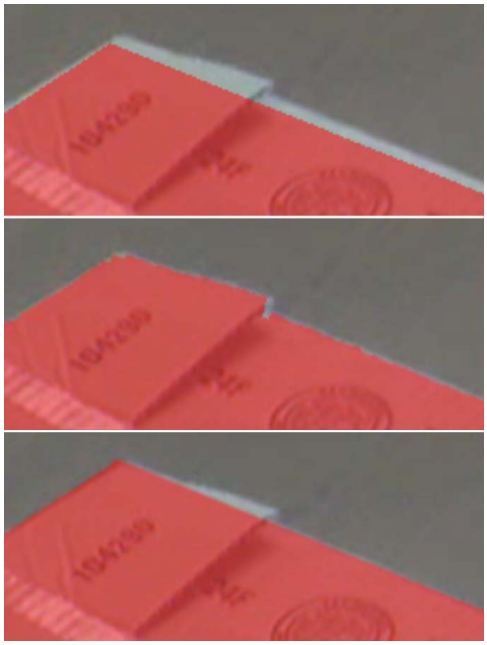

Figure 6: Comparison of results. *Top to bottom:* The original polygon edge, our detected edge, and the smoothed detected edge.

within the search region, then the detected edge will 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 detected edge will be noisy and will jitter from frame to frame, so the smoothed edge will be more appropriate. While the smoothed edge may not match the physical edge as closely, it will jitter less between frames and match more closely than the original polygon edge.

5.2 Discussion

The limitations of this technique are important to examine. Foremost is the fact that we do not modify the tracking result, just the final rendering. This means we cannot correct cumulative drift from other tracking technologies. It also means that polygons are clipped, rather than moved, so the borders of textures will be clipped as well. On the other hand, effects such as per-pixel lighting will not suffer, as they are calculated uniquely for each pixel.

We must 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, 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. If a noisy edge is replaced by the mean offset edge, any corrections of modeling error are lost. It would be preferable to smooth the edge in a feature-preserving way. We partially address this issue by allowing the noisiness threshold to be user-specified per-edge. This allows the user to judge the modeling error for an edge and determine how much noise should be tolerated.

Finally, in cases where polygons are viewed from a shallow angle, or when the tracking error is very large, our algorithm will fail because the search regions for opposite edges will overlap. This can confuse detected edges and 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.

6 Conclusions

We have presented a general post-processing technique for reducing the visual effects of registration and modeling error. The algorithm is appropriate for a class of AR applications which modify the appearance of physical objects, 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 high contrast environments, our algorithm will considerably improve registration results, within a few pixels. Even in low contrast scenes, our fallback options yield improved stability. By taking advantage of graphics hardware, we achieve these results with minimal impact to application performance. Future work includes exploring more robust and domain-specific edge detection techniques, and local, feature-preserving edge smoothing.

References

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

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

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.