Parctically Frameless Rendering

Mathias M. Wloka Robert C. Zeleznik and Timothy Miller

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-95-06
March 1995

Practically Frameless Rendering

Matthias M. Wloka, Robert C. Zeleznik, and Timothy Miller
Science and Technology Center for Computer Graphics and Scientific Visualization,
Brown University Site*

Abstract

We describe practically frameless rendering: a new technique that applies wherever a frame-buffer is in use, for example, to 3D z-buffered rendering or to 2D user interfaces. Its advantages include higher frame-rate (our software implementation almost doubles the frame-rate and future hardware implementations potentially quadruple it), less lag, and generation of motion blur without additional overhead. However, practically frameless rendering also degrades image quality. Thus, practically frameless rendering allows the programmer to trade faster image generation for reduced image quality. Depending on the application, such a trade-off might be desirable. We explore some of the possible variations.

Keywords: frameless rendering, adaptive refinement, motion blur, scan-conversion, real-time, frame-rate, lag, spatial coherency

1 Introduction

We introduce a new technique that almost doubles and potentially quadruples the frame-rate of today's high-performance z-buffer renderers. In addition, the technique produces an approximate motion blur at no additional cost. However, it also affects image quality (see Figure 1).

We call this new technique *practically frameless rendering*. It applies ideas from frameless rendering [2] to scan-conversion renderers, thus making frameless rendering practical. Nonetheless, our new technique maintains discrete frames, i.e., strictly speaking it is not frameless.

1.1 Benefits and Drawbacks

Practically frameless rendering has many advantages. Most notably, it sharply increases frame-rate (see Section 4). The increased frame-rate in turn reduces end-to-end lag [14], thus making applications more usable [7] [10]. More important, increasing frame-rate in head-mounted displays beyond a certain threshold (typically around 10 frames per second) boosts user performance dramatically for certain tasks [10].

*Box 1910, Department of Computer Science, Brown University, Providence, RI 02912. Phone: (401) 863-7600, email: {mmw|bcz|tsm}@cs.brown.edu.

Show-Off Photo

(Room with flying hamburger and rotating chair.)

Figure 1: Practically frameless rendering almost doubles and potentially quadruples the frame-rate of today's high-performance renderers. However, it also affects image quality.

However, since our technique and its variations only recompute a quarter of the pixels per frame (as compared to traditional double-buffered rendering), comparisons between our update rate and traditional frame-rate are unfortunately equivocal. Ultimately, user studies must ensure that we are not comparing apples to oranges.

Our new rendering technique also introduces visual artifacts (see Figure 1) that approximate motion blur. While adding motion blur to scenes is in general desirable [15] [3], it used to be reserved for non-interactive tasks because motion blur algorithms historically incur high overhead. Real-time algorithms for use in interactive 3D applications are only a recent development [15] [11]. While those motion blur algorithms induce only a small overhead, rendering without blur is nonetheless faster. In contrast, practically frameless rendering generates motion blur while boosting performance at the same time.

On the other hand, the generated motion blur is only approximate: it is roughly equivalent to rendering the object semi-transparently at four different times and displaying the result in the same frame. Thus, fast moving objects generate visible artifacts (see Figure 2).

Furthermore, the blur always applies to the whole scene. Thus, a fast head rotation in an immersive setup generates an indistinguishable blur. We do not know how or if the user is affected. We nonetheless provide alternatives designed to avoid possible problems in Section 5.

As a secondary effect of the motion-blur-like artifact, objects never pop into or out of view. Instead, they fade in or out of existence. In general, all abrupt and discontinuous changes in the image stream are automatically smoothed to gradual fades — a particularly beneficial feature for objects with changing levels of detail. Previous techniques must simulate such a fade either by morphing object shapes or by simultaneously rendering the object semi-transparently both at the lower level of detail as well as at the higher one [12]; thus, previous techniques always incur extra overhead when switching levels of detail whereas practically frameless rendering does not.

Since practically frameless rendering applies equally to 2D rendering, 2D user interfaces in particular benefit from the produced motion blur and image fades [3]. While previous work already implements these ideas [3], practically frameless rendering provides these effects for free, i.e., the programmer need not explicitly specify them.

1.2 Overview

The remainder of this paper is structured as follows. Section 2 relates practically frameless rendering to previous work in frameless rendering, adaptive refinement, and motion blur. The basic mechanism underlying practically frameless rendering is described in Section 3. Then, in Section 4 we demonstrate how to implement practically frameless rendering in software and measure the resulting performance. Section 5 explores several variations of the new algorithm. Finally, in Section 6 we discuss our contribution as well as possible future work.

2 Related Work

2.1 Frameless Rendering

Frameless rendering [2] proposes to shorten overall lag by banishing double-buffering. To avoid display artifacts such as image tearing, the authors propose a randomized sampling scheme to update individual pixels. Such a scheme integrates well with ray-tracers [13], but makes frameless rendering inapplicable to current polygon renderers, i.e., useless for today's high-performance rendering architectures.

In contrast, practically frameless rendering takes advantage of spatial coherency and thus specifically applies to scan-converting z-buffer engines. As a result, frame-rate increases so much that software implementations are feasible and useful (see Section 4).

However, to exploit spatial coherence we revert to maintaining discrete frames. We believe the performance gain achieved through exploiting spatial coherence more than justifies the delay introduced by discrete frames. In addition, some display technologies, particularly CRT designs, refresh their pixels sequentially — in effect, they update pixels a frame at a time, therefore disabling truly frameless rendering anyhow.

2.2 Adaptive Refinement

Practically frameless rendering is also an adaptive refinement [1] technique. As such, it is limited because for static images it converges on the final rendering in only four iterations (although Section 6 shows how to extend it to possibly use nine or sixteen iterations). However, it is also an efficient refinement technique if implemented in hardware: rendering the final image directly is only slightly faster (see Section 4).

Single Object Artifact Photo

"Hamburger in Space"

Figure 2: The motion blur practically frameless rendering generates is visually inferior to other known techniques, in particular for fast moving objects. Nonetheless, its low computation overhead makes it an attractive alternative for highly interactive applications.

2.3 Motion Blur

Since practically frameless rendering generates visual artifacts that approximate motion blur, we also evaluate it as a motion blurring technique. Motion blurring techniques roughly fall into three classes: non-real-time, 2D, and real-time 3D solutions.

First, non-real-time solutions [4] [5] create beautiful images of motion blur. However, their excessive use of compute time disqualifies their use in interactive applications. Since practically frameless rendering strictly applies to highly interactive applications, the respective visual results are incomparable.

Second, while 2D solutions [9] [11] are more efficient than the non-real-time solutions, they usually rely on direct access to the 2D image bitmap. Thus, their implementation usually cannot take advantage of today's high-performance rendering architectures and therefore, they are also inapplicable in highly interactive settings. Again, we thus feel that the visual results are incomparable.

Third and last, real-time 3D solutions [15] [6] are fast enough for the highly interactive domain where practically frameless rendering applies. The quality of the blur generated by practically frameless rendering is inferior to those techniques (in particular for fast moving objects, see Figure 2). On the other hand, practical frameless rendering is much faster: if we view traditional double-buffered rendering as a motion blur technique that produces low quality blurs (i.e. none), practically frameless rendering produces better blurs and does it faster than double-buffered rendering.

3 Basic Method

Practically frameless rendering remaps how pixels in the frame-buffer display on the screen. Instead of using an identity mapping which displays pixel (x, y) at display position (x, y), we map all pixels (x, y) in the top-left quadrant of the frame-buffer to display positions $(2x, 2y)$. The other three quadrants map similarly as illustrated in Figure 3.

We then render a scene at quarter resolution into one of the frame-buffer quadrants. The next image renders into a different quadrant.

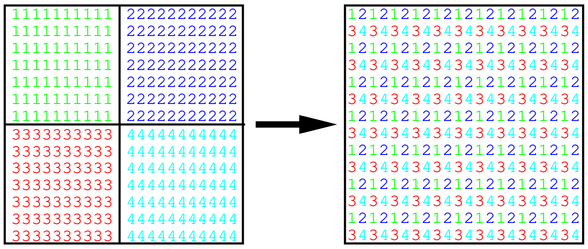

Figure 3: Practically frameless rendering remaps how frame-buffer pixels display. The top-left quadrant of the frame-buffer maps to every fourth pixel on the display, thus spreading it over the whole display as shown. The other three quadrants map similarly, also shown. The mapping is implemented either by copying pixels between buffers or, without overhead, by address remapping.

We select which quadrant to render into by successively cycling through all four. The effect on the display is shown in Figures 4 and 5.

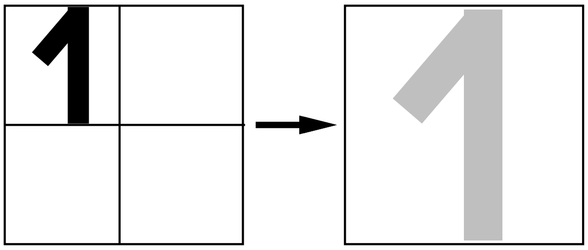

Figure 4: Images are rendered at quarter resolution into a quarter of the frame-buffer as shown at left. These image pixels then map to every fourth pixel in the display as explained in Figure 3.

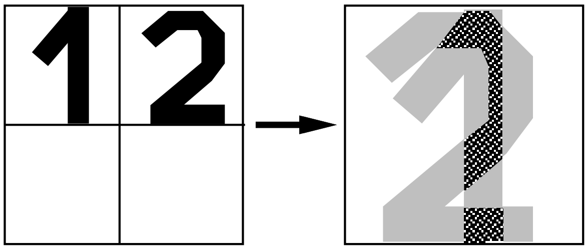

Figure 5: The next quarter-resolution image maps to a disjunct set of pixels, thus displaying the current and the three previous quarter-resolution images simultaneously.

Due to our frame-buffer-to-display mapping all four frame-buffer quadrants are displayed at all times. A static object appears solid after four iterations, since its image in all four quadrants is identical. The pixels of a dynamic object, however, are interspersed with the background pixels, since the image of the dynamic object in the four quadrants changes over time. This pixel interspersion approximates transparency and thus produces a motion-blur-like effect for dynamic scenes (see Figures 1 and 2).

For a static scene the above described method renders the same image in each quadrant. Thus, the four pixels in each two-by-two pixel group in the display are all the same. Effectively, we thus display a quarter-resolution image expanded both horizontally and vertically by a factor of two via pixel replication.

To avoid this image degradation, practically frameless rendering transforms the camera slightly before it renders into a particular frame-buffer quadrant. For example, if we render into the top left quadrant, we translate the original camera film-plane half a frame-buffer-pixel up and half a pixel to the left. The original camera eye-point remains unchanged. Similarly, the top-right, bottom-left, and bottom-right quadrants cause the camera film-plane to move half a frame-buffer-pixel up, down, or down, respectively and half a pixel to the right, left, or right, respectively.

These transformations guarantee that after rendering four static quarter-resolution images, the displayed image is full resolution, if the renderer supports subpixel resolution. In other words, for static scenes four iterations of practically frameless rendering produce an image equivalent to a conventional full-resolution rendering.

If the renderer does not support subpixel resolution, e.g., the implementation described in Section 4.1, then shifting the film-plane as described above does not necessarily produce a full-resolution rendering. Because vertex coordinates are rounded to the nearest pixel, each of the four renderings produce a triangle that is slightly different from the desired full-resolution rendering. These rounding errors thus cause the union of the four rendered triangles to be different from from the full-resolution triangle (see Figure 6).

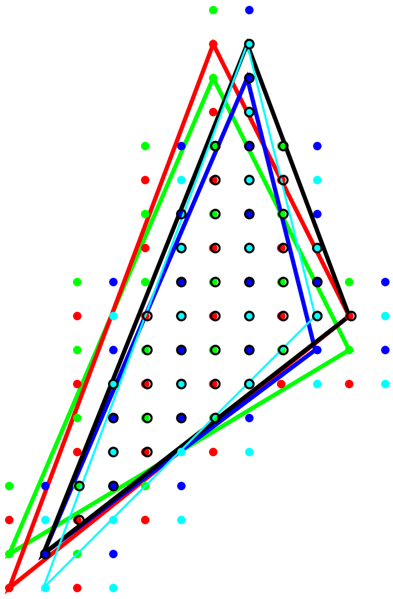

Figure 6: The black lines (and all the pixels outlined in black) represent the full-resolution rendering of a triangle. The green, blue, red, and cyan lines and the respectively enclosed green, blue, red, and cyan pixels represent the quarter-resolution renderings of the triangle. The union of the quarter-resolution renderings is not equivalent to the full-resolution rendering due to rounding errors.

Finally, practically frameless rendering is faster than traditional rendering because for each update it only creates a quarter-resolution image instead of a full-resolution one. If we are assuming that today's renderers are pixel-bound, i.e., their speed is limited by how fast they scan-convert, rendering quarter-resolution images should be as much as four times faster than rendering full-resolution images. This assumption is not unreasonable, given the current trend towards constructing scenes with less, yet larger and texture-mapped polygons. In addition, our performance tests described in Section 4 confirm these numbers.

4 Software Implementations and Performance

Since we cannot directly modify our hardware renderers, we implement the frame-buffer-to-display mapping shown in Figure 3 in software. We describe two such implementations. The first uses the rectcopy and stenciling features [8] of the SGI RealityEngine2. The second one alters a high-performance software z-buffer to produce images at different resolutions.

4.1 RealityEngine2 Implementation

The following three steps implement practically frameless rendering on our SGI Onyx with a RealityEngine2.

First, we select which frame-buffer quadrant to render into. (We select quadrants in the following order: top-left, bottom-right, top-right, bottom-left, repeat.) We center the camera with respect to that quadrant and also slightly move the image-plane position as described in Section 3. Then, we change the view-port so that all subsequent operations only affect that selected quadrant of the back-buffer.

Second, we clear the color- and z-planes of that back-buffer quadrant. Then, we render the scene into the cleared quadrant.

Third and last, we copy the pixels from the back-buffer quadrant to the front-buffer. We use horizontal and vertical pixel replication (i.e. `rectzoom(2.0, 2.0)`) to fill the entire front-buffer with the new data. To avoid overwriting data from the other three quadrants already stored in the front-buffer, we also enable stenciling before executing the pixel transfer. The stencil mask used depends on which quadrant we are transferring; for example, if we are transferring the top-left quadrant, all but the top-left pixel for every two-by-two pixel group are blocked, i.e., only the top-left pixel for every two-by-two group actually overwrites a front-buffer pixel.

Thus, we implement the mapping depicted in Figure 3 by copying pixels from the back-buffer to the front-buffer (i.e., using `rectcopy()`). The copy operation is modified to replicate and stencil (i.e., `rectzoom()` and `stencil()`) pixels to achieve the desired effect.

The front- and back-buffers in this implementation have different functions, i.e., the back-buffer generates quarter-resolution renderings and the front-buffer retains the four most recent renderings in a displayable format. Therefore, they are no longer interchangeable: we never switch front- and back-buffers. It also follows that this implementation is suspectable to artifacts usually associated with single-buffer displays. In particular, because the pixel transfer from the back-buffer to the displayed front-buffer takes 13 ms, i.e., longer than the vertical retrace of the monitor, image tearing is possible. In practice, since only every fourth pixel exhibits the tear, it is not as noticeable as in conventional single-buffer displays.

Table 1 shows how this implementation performs on an SGI Onyx with a RealityEngine2. All examples render to a 800 by 800 pixel display. The actual speed-up indicates how much faster this implementation of practically frameless rendering is compared to conventional double-buffered rendering. The potential speed-up discounts the overheads particular to the implementation of the rendering method. For example, when double-buffering the scenes "Abstracts," "Room," and "Campus," we wait approximately 16 ms, 8 ms, and 8 ms, respectively, to swap buffers. The pixel transfer in practically frameless rendering from back- to front-buffer costs a constant 13 ms per frame. The potential speed-up thus reflects the speed-up possible if practically frameless rendering were implemented in hardware by remapping addresses. (The scenes "Abstracts," "Room," and "Campus" are shown in Figures 10, 11, and 12, respectively.)

Note that the speed-up achieved by practically frameless rendering substantially decreases the more polygons we render. This decrease indicates that we are no longer pixel-bound as the number of polygons increases while their size decreases.

	Abstracts	Room	Campus
Number of Triangles	656	5,000	10,000
Double-Buffered	33 ms (30 fps)	34 ms (30 fps)	74 ms (13 fps)
Practically Frameless	18 ms (57 Hz)	26 ms (39 Hz)	69 ms (15 Hz)
Actual Speed-Up	1.9	1.3	1.1
Potential Speed-Up	3.8	2.0	1.2

Table 1: Performance results of our implementation using an SGI Onyx with a RealityEngine2. The rows "double-buffered" and "practically frameless" show how long it takes to render the scene using double-buffered rendering and practically frameless rendering, respectively. The actual speed-up is the speed-up achieved. The potential speed-up discounts the overhead inherent to the implementation, i.e., waiting for the vertical retrace in double-buffered rendering and transferring pixels from the back- to the front-buffer in practically frameless rendering.

4.2 Software Z-Buffer Implementation

In the previous section, we produce quarter-resolution images by rendering into a frame-buffer that is only one fourth the size of the original frame-buffer. Another way to generate quarter-resolution images is to change the step-size of the scan-converter: instead of computing every pixel in a full-resolution image, it only computes every other pixel of every other scan-line. The software z-buffer implementation described here uses this second approach.

Our software z-buffer uses the following three steps to implement practically frameless rendering. First, it clears every other pixel on every other scan-line in an off-screen buffer. Second, because it uses an appropriately adjusted step-size, it then scan-converts and z-buffers polygons into these cleared pixels only. Third and finally, we bit-blit the off-screen buffer to the screen. The bit-blit copies all the pixels from the off-screen buffer, i.e., not just the modified ones, and thus induces overhead.

As we repeat these three steps, we adjust the horizontal and vertical step-size offsets so that after four iterations all the pixels in the off-screen buffer are modified.

Because this algorithm operates in full-resolution and only skips certain pixels, it always rounds the vertex coordinates to the correct pixels. Therefore, it automatically renders a static scene in full-resolution after four passes (without having to modify any camera parameters in-between passes). In particular, it does not share the rounding error problems described in Section 3 and exhibited in the implementation of Section 4.1.

Table 2 shows how this pure software implementation performs. All timing measurements are for 512 by 512, 8 bit raster images rendered on an HP 9000/735 with a CRX24Z graphics card. Double-buffering is achieved via the same implementation, except that the step-sizes are adjusted to render full-resolution images for each iteration. Neither the practically frameless rendering nor the double-buffered rendering wait for the vertical retrace. (The scenes "Cube," "Txplane," and "Teapot" are shown in Figures 13, 14, and 15, respectively.)

	Cube	Txplane	Teapot
Number of Triangles	12	700	7000
Double-Buffered	38 ms (26 Hz)	42 ms (24 Hz)	110 ms (9 Hz)
Practically Frameless	24 ms (41 Hz)	32 ms (31 Hz)	105 ms (10 Hz)
Speed-Up	1.6	1.3	1.1

Table 2: Performance results of our implementation using a modified software z-buffer running on an HP 9000/735 with CRX24Z graphics card. The rows "double-buffered" and "practically frameless" show how long it takes to render the scene using double-buffered rendering and practically frameless rendering, respectively. The actual speed-up is the speed-up achieved. The scenes "Cube" and "Txplane" are texture-mapped; the scene "Teapot" is not. None of the objects are shaded.

5 Variations

In the following sections we vary the basic frame-buffer-to-display mapping introduced in Figure 3. While we always maintain the characteristics of practically frameless rendering as described in Section 3, these variations differ slightly in the images they produce — each one trades off different image enhancements and degradations.

5.1 Locally Randomized Grid

The original mapping always maps pixels from the top-left frame-buffer quadrant to the top-left pixel in each two-by-two pixel group in the display (see Figure 3). Similarly, pixels from the top-right, bottom-left, and bottom-right quadrant respectively map to the top-right, bottom-left, and bottom-right pixel in each two-by-two group. Thus, we say that the top-left, etc. quadrant always maps to the top-left, etc. pixel.

Since there are four quadrants that have to choose one and only one pixel (in every two-by-two group), there is a total of $4! = 24$ possible quadrant-to-pixel maps. Note that the original method picks one of these quadrant-to-pixel maps and applies it to every two-by-two pixel group in the display.

In this variation, we randomly pick one of the 24 different quadrant-to-pixel maps for every two-by-two pixel group. Since we thus perturb the overall frame-buffer-to-display map only locally (i.e., within every two-by-two pixel group), we call this variation *locally randomized grid*.

Because the locally randomized grid is not as regular as the original mapping, two advantages result. First, the increased irregularity diffuses the motion blur artifacts, thus making the motion blur higher quality (see Figure 7). Second, the increased irregularity also reduces image tearing artifacts.

On the other hand, the locally randomized mapping makes it impossible to adjust camera parameters to automatically generate full-resolution images for static scenes (see Section 3). Thus, one either accepts quarter resolution, or switches to double-buffered rendering whenever the scene becomes static (or whenever the viewpoint becomes static).

Random Grid Blur Picture

Figure 7: The locally randomized grid variation locally perturbs pixel mappings. Thus, it diffuses the motion blur artifacts, heightening the image quality.

5.2 Semi-Transparent Grid

Practically frameless rendering, as a motion blur technique, always simultaneously displays four distinct temporal samples of the scene (see Section 3). Since each sample each uses one fourth of the display's pixels, all samples have the same weight. Therefore, we emulate a temporal box-filter [5]. Other temporal filters, for example, ramp filters, generally produce visually more pleasing results.

To experiment with different temporal filters, we change the original implementation of practically frameless rendering. Instead of overwriting exactly one pixel in every two-by-two pixel group when copying a frame-buffer quadrant to the display, we instead alter all four pixels in the two-by-two group. For instance, when copying the top-left quadrant, we overwrite the top-left pixel in every two-by-two group. In addition, we assign the top-right pixel a value halfway in-between its old value and the new value from the top-left quadrant. Similarly, the bottom-left pixel combines three fourths of its old value with one fourth the new value. The bottom-right pixel remains unchanged.

We call this technique *semi-transparent grid* because it writes pixels from the frame-buffer to the display through a semi-transparent grid. Like the locally randomized grid, it also prohibits automatic full-resolution rendering of static scenes. However, it does improve the generated motion blur (see Figure 8).

5.3 Grid with Centroid Bleeding Hole

As mentioned in Section 1, because practically frameless rendering always blurs the whole scene, users cannot focus anywhere during rapid camera movements, thus possibly causing problems in immersive setups. To avoid these possible problems we create a circular inset in the middle of the display that never blurs. We implement this inset by overwriting all the pixels within the inset every time we copy a frame-buffer quadrant to the display. In other words, whereas outside the inset each frame-buffer quadrant overwrites exactly one pixel in every two-by-two group (as before), within the inset each frame-buffer quadrant always overwrites all four pixels in every two-by-two group.

Semi-Transparent Grid Blur Picture

Figure 8: The semi-transparent grid variation writes pixels from the frame-buffer to the display through a semi-transparent grid. It thus emulates different temporal filters, improving the generated motion blur.

Bleeding Hole Picture

Figure 9: The grid with centroid bleeding hole variation always updates all the pixels within a circular center-inset. Thus, objects within the inset never blur, enabling the user to focus during rapid camera movements.

Since the human perceptual system easily recognizes discontinuities, we must diffuse the border between the inside (where all the pixels update) and the outside (where only one fourth of the pixels update) of the inset. We do so by modifying the mapping for pixels outside the inset: the closer a frame-buffer-quadrant pixel maps to the inset border, the more likely that it overwrites not only the display pixel it maps to, but also one or several of the other pixels in the same two-by-two group. Figure 9 shows an image generated by this technique.

While this technique might be superior for immersive setups, it introduces other problems. The image part within the inset is always quarter resolution. Slightly shifting the film-plane as described in Section 3 (for example, to at least maintain the automatic full-resolution for the image parts outside the inset), causes object edges within the inset to flicker — after all, within the inset we are displaying four slightly different images in rapid succession. In addition, image tearing becomes obvious within the inset.

6 Conclusion and Future Work

Practically frameless rendering is a new technique for use in interactive computer graphics. It increases frame-rate, decreases lag, and generates motion blur without overhead. However, it also updates only a quarter of the pixels at a time, and only approximates motion blur.

We think it is a valuable tool for graphics programmers because it allows them to explore various trade-offs between frame-rate and lag versus image quality. As shown in Section 5, practically frameless rendering influences quality attributes such as resolution, interlace, or blur. Hopefully, practically frameless rendering inspires other techniques that allow similar trade-offs.

Furthermore, practically frameless rendering applies broadly. While this paper concentrates on its use for 3D rendering, in particular in conjunction with high-performance renderers, it is not limited to this domain. Practically frameless rendering applies wherever a frame-buffer is in use. Thus, for example, 2D user interfaces benefit equally. Practically frameless rendering's simplicity further aids a possibly wide application, especially if implemented in hardware.

6.1 Future Work

Since practically frameless rendering degrades image quality to various degrees, user studies need to be performed to evaluate the effects on user performance. Such studies would compare those effects to the benefits achieved through increased frame-rate and decreased lag.

Furthermore, the variations introduced in Section 5 cause the loss of automatic full-resolution rendering of static scenes. Therefore, an interesting extension of practically frameless rendering would switch between practically frameless rendering for dynamic scenes and double-buffering for static scenes. Such a combination technique possibly eliminates the disadvantages associated with using either one of the techniques in isolation. Deciding when a scene is sufficiently static to perform the switch is one of the associated problems.

Finally, practically frameless rendering generalizes from rendering images at quarter resolution to rendering images at $\frac{1}{9}$th, $\frac{1}{16}$th, ... the original resolution. While we would not expect a comparable speed-up for those other resolutions, the speed-ups generated might nonetheless be useful.

Acknowledgements

We thank Ellen Scher Zagier for discussing her work on frameless rendering with us and our advisors Andy van Dam and John Hughes for their support.

We gratefully acknowledge the support of this work in part by NSF/ARPA Science and Technology Center for Computer Graphics and Scientific Visualization, Sun Microsystems, Autodesk, Taco Inc., Microsoft, NASA, NCR, and ONR Grant N00014-91-J-4052, ARPA Order 8225.

References

[1] L Bergman, H. Fuchs, E. Grant, and S. Spach. Image rendering by adaptive refinement. *Computer Graphics (SIGGRAPH '86 Proceedings)*, 20(4):29–37, August 1986.

[2] Gary Bishop, Henry Fuchs, Leonard McMillan, and Ellen J. Scher Zagier. Frameless rendering: Double buffering considered harmful. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 175–176. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[3] Bay-Wei Chang and David Ungar. Animation: From cartoons to the user interface. *UIST Proceedings*, pages 45–55, November 1993.

[4] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *Computer Graphics (SIGGRAPH '84 Proceedings)*, 18(3):137–145, July 1984.

[5] Mark A. Z. Dippe and Erling Henry Wold. Antialiasing through stochastic sampling. *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):69–78, July 1985.

[6] Paul Haeberli and Kurt Akeley. The accumulation buffer: Hardware support for high-quality rendering. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):309–318, August 1990.

[7] Richard Held and Nathaniel Durlach. Telepresence, time delay and adaptation. In Stephen R. Ellis, editor, *Pictorial Communication in Virtual and Real Environments*, chapter 14. Taylor and Francis, 1991.

[8] Steven W. Hiatt, editor. *Graphics Library Programming Guide – Document Number 007-1210-060*. Silicon Graphics, Inc., Mountain View, CA, 1992.

[9] J. Korein and N. Badler. Temporal anti-aliasing in computer generated animation. *Computer Graphics (SIGGRAPH '83 Proceedings)*, 17(3):377–388, July 1983.

[10] Andrew Liu, Gregory Tharp, Lloyd French, Stephen Lai, and Lawrence Stark. Some of what one needs to know about using head-mounted displays to improve teleoperator performance. *IEEE Transactions on Robotics and Automation*, 9(5):638–648, 1993.

[11] Nelson Max. Polygon-based post-process motion blur. *The Visual Computer*, 6(6):308–314, December 1990.

[12] John Rohlf and James Helman. IRIS performer: A high performance multiprocessing toolkit for real-time 3d graphics. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 381–394. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[13] Andrei State, Jonathan McAllister, Ulrich Neumann, Hong Chen, Tim J. Cullip, David T. Chen, and Henry Fuchs. Interactive volume visualization on a heterogeneous message-passing multicomputer. In *1995 Symposium on Interactive 3D Graphics*, 1995. To be published.

[14] Matthias M. Wloka. Lag in multiprocessor virtual reality. *Presence*, 4(1), 1994.

[15] Matthias M. Wloka and Robert C. Zeleznik. Interactive real-time motion blur. In Michael Gigante, editor, *Insight through Computer Graphics: Proceedings of CGI'94*, page 83. World Scientific, 1994.

Abstracts Picture

Figure 10: The scene "Abstracts" used for performance evaluation.

Room Picture

Figure 11: The scene "Room" used for performance evaluation.

Campus Picture

Figure 12: The scene “Campus” used for performance evaluation.

Cube Picture

Figure 13: The scene “Cube” used for performance evaluation.

Txplane Picture

Figure 14: The scene “Txplane” used for performance evaluation.

Teapot Picture

Figure 15: The scene “Teapot” used for performance evaluation.