The Accumulation Buffer: Hardware Support for High-Quality Rendering

Paul Haeberli and Kurt Akeley

Silicon Graphics Computer Systems

ABSTRACT

This paper describes a system architecture that supports realtime generation of complex images, efficient generation of extremely high-quality images, and a smooth trade-off between the two.

Based on the paradigm of integration, the architecture extends a state-of-the-art rendering system with an additional high-precision image buffer. This additional buffer, called the Accumulation Buffer, is used to integrate images that are rendered into the framebuffer. While originally conceived as a solution to the problem of aliasing, the Accumulation Buffer provides a general solution to the problems of motion blur and depth-of-field as well.

Because the architecture is a direct extension of current workstation rendering technology, we begin by discussing the performance and quality characteristics of that technology. The problem of spatial aliasing is then discussed, and the Accumulation Buffer is shown to be a desirable solution. Finally the generality of the Accumulation Buffer is explored, concentrating on its application to the problems of motion blur, depth-of-field, and soft shadows.

CR Categories and Subject Descriptors: I.3.1 **[Computer Graphics]:** Hardware Architecture - Raster display devices; I.3.3 **[Computer Graphics]:** Picture/Image Generation - display algorithms; I.3.7 **[Computer Graphics]:** Three-Dimensional Graphics and Realism - Color, shading, shadowing and texture.

Additional Key Words and Phrases: Accumulation buffer, antialiasing, motion blur, depth of field, soft shadows, stochastic sampling.

1. Introduction

Traditional 3D graphics workstations include simplistic scan conversion hardware that samples points, lines, and polygons with a single infinitely small sample per pixel. As a result, the renderings of these primitives show aliasing artifacts. While increasing monitor pixel density has reduced the effects of aliasing, motion, the result of increased workstation performance, has increased them. The problem of aliasing remains significant in workstation rendering architectures.

Contemporary workstations have attempted to solve the problem of aliasing using a variety of architectures. Several vendors offer machines that compute proper pixel coverage for points and lines. An early example is the raster-based Evans and Sutherland PS-390 [E&S 87], whose design goal was to duplicate the point and line quality of its calligraphic predecessors. A simpler and less effective line drawing algorithm is implemented by the Silicon Graphics GT system [Akeley 88]. This solution offers a great improvement over aliased lines, but still displays slope and endpoint related anomalies. Both these point and line solutions, and all others known to the authors, rely on the following two observations:

1. Pixel coverages are relatively easy to compute, because the screen geometry of the scan converted primitive is regular and predictable.
2. The quality of intersections is relatively less important than the quality of background-abutting edges, both because background-abutting edges predominate, and because intersections are typically between unrelated points or lines.

Neither of these assumptions is correct for polygons. Because vertexes and narrow areas are neither regular nor predictable, it is difficult to compute correct pixel coverage during polygon scan conversion. Further, polygons frequently share edges with related polygons, and intersect unrelated polygons. Still, there are some examples of workstation-class machines that attempt these calculations. The Pixel Machine [Potmesil 89] takes the brute-force approach of oversampling and convolution with an arbitrary filter. While effective, this approach does not map nicely onto conventional scan-conversion hardware (the Pixel Machine scan conversion system is an array of general purpose processors). The Graphicon 2000 [Star 89] implements an approximation of an A-buffer [Carpenter 84], including hardware that computes a 4x4 coverage mask for each pixel. This implementation suffers from limited (fixed) resolution and errors at polygon vertexes and edges (when the polygon is very thin).

We sought a polygon antialiasing solution with the following properties:

- *Compatibility.* The solution should leverage the capabilities of a contemporary scan conversion system. It should be orthogonal to the features already present, including surface generation, blending, texture mapping, and interactive constructive solid geometry generation.
- *High Quality.* There should be no limit to the quality of the result obtained. Thus, for example, fixed size masks, and algorithms that store finite amounts of depth or color data per pixel, could not be used.
- *Smooth performance/quality tradeoff.* Not only must the quality of the result be allowed to increase without bound, but it also must decrease smoothly toward an acceptable minimum. As quality is decreased, performance must increase toward a maximum that is competitive with other contemporary architectures.

First we describe the performance and quality characteristics of the current generation of workstation graphics systems. Then we describe an architecture that extends these characteristics to include polygon antialiasing. Finally we discuss the additional system features that result from the generality of our solution.

2. Current Architectures

The problem of correctly sampling, and thus antialiasing polygons has been solved many times in many ways. Our concern here is to solve it in a manner that complements the operation of a contemporary high-performance scan conversion system. We must first be familiar with the properties of such a system.

2.1 Polygon Performance

The most obvious trend in high-performance workstation graphics is toward the capability of rendering *lots* of *small* polygons per second. Numbers for previous generation machines reached the 100,000 to 150,000 range [Akeley 88, Apgar 88]. The recently introduced Silicon Graphics 4D VGX raises this number to 750,000 RGB lighted, Gouraud shaded, z-buffered, connected triangles per second, and to 1,000,000 per second when the triangles are flat shaded.

Substantial hardware resources are dedicated toward achieving these impressive polygon rates. We would like to leverage this investment when drawing antialiased polygons.

2.2 Sampling Quality

A second trend is that toward improved sampling quality. Traditional workstation scan conversion systems have taken a less than rigorous attitude toward sampling. Shortcuts in arithmetic processing result in artifacts such as:

1. *Non-subpixel positioning.* After transformation, vertex coordinates are rounded to the nearest pixel location in an integer screen space.
2. *Bresenham sampling.* Pixels are included in the scan conversion of a polygon based on arithmetic appropriate for a line fill, rather than for an area sampling.
3. *Sloppy iteration.* Insufficient accuracy is maintained during edge iteration. Slopes and initial values of parameters are not corrected for the subpixel locations of vertexes, spans or scans.

Early Silicon Graphics machines [SGI 85] and Hewlett Packard graphics systems [Swanson 86] were guilty of all three errors. The Silicon Graphics GT graphics system, first shipped early in 1988, addressed issues 2 and 3, but still forced transformed coordinates to the nearest pixel center [Akeley 88]. More recently shipped machines, including the Silicon Graphics Personal Iris and the Stellar GS1000 [Apgar 88], correctly address all three concerns. The Pixel-Planes system [Fuchs 85] is an early example of an architecture that implements accurate polygon sampling.

2.3 Point Sampling

We refer to a scan conversion algorithm that rigorously selects pixels for inclusion, and rigorously computes parameter values at each pixel, as a Point Sampling algorithm. Such rigor is most easily defined for triangles. The requirements are:

1. The projected vertexes of the triangle must not be perturbed during the scan conversion process.
2. Pixels must be chosen for inclusion in the triangle scan conversion based on whether their infinitely small sample point is inside or outside the exact triangle boundary. A fair test must be established for pixels whose sample point is exactly on the triangle boundary.
3. Parameter values must be assigned to each pixel based on exact calculation at the infinitely small sample point. Such exact calculation is easily defined for triangles as the solution of the plane equation specified by the parameter values at the triangle's three vertexes.

While Point Sampling can require significantly more arithmetic than less rigorous sampling, it has numerous benefits. The Point Sampling pixel inclusion property insures that adjacent polygons neither share nor omit any pixels along their common border. Thus algorithms that count on the number of times a pixel is drawn, such as transparency and constructive solid geometry [Goldfeather 86], operate correctly. Redraws of single-buffered images also are much less "noisy", because pixels change color less often.

The *planar* sampling inherent in Point Sampling allows polygon intersections to be Z-buffered accurately, resulting in smooth transitions from one polygon to the other. Likewise, smooth shaded polygons show no shear artifacts, such as those illustrated at the bottom of Figure 1.

Finally, as we will see in the following section, the accuracy inherent in Point Sampling allows images to be integrated with predictable results.

3. Antialiasing

When polygons are sampled with only one sample per pixel, aliased images, such as the one illustrated in Figure 2, are created. To obtain a properly sampled (antialiased) result, the rendering must take into account the areas of all the polygons that contribute to the shading of each pixel, rather than just a single sample point. This can be accomplished in one of two fundamentally different ways:

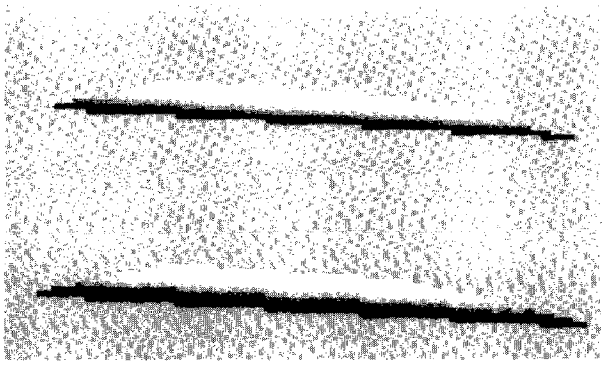
1. *Area Sampling.* The fraction of pixel coverage due to each polygon that intersects the pixel (perhaps multiplied by a filter function) is computed, and these fractions are blended to obtain the final pixel shading.
2. *Multi-Point Sampling.* Many point samples are taken in the region of each pixel, and these samples are integrated (again perhaps with a weighting function) to obtain the final pixel shading.

3.1 Area Sampling

While the area sampling solution has proved useful when antialiasing points and lines, its implementation for the antialiasing of polygons has several problems. These include:

1. Pixel coverage by a polygon is not easy to compute. Unlike points and lines, polygons can become arbitrarily thin, and have vertexes with arbitrary orientation relative to the pixel. Pixel coverage therefore cannot be computed with a simple function such as distance from the center of a point (or line).
2. The choice of parameter values to assign to each coverage fraction is arbitrary and inaccurate. Because parameters typically vary across the pixel, no single sample will yield a correct value. Worse yet, when the fraction-piece does not include the pixel center, a Point Sample algorithm has *no* parameter value to assign. (Any attempt to compute a parameter value outside a Point Sampled polygon risks substantial overflow or underflow.)
3. Correctly blending the pieces into a final pixel value is difficult as well. If the geometric relationship of the pieces is not known, blending will fail either for correlated edges (adjacent polygons) or for uncorrelated edges (intersecting polygons), depending on the blending function chosen. Knowing the geometric relationship requires that some sort of multi-sample operation be done, which violates the spirit of this solution.

While all of these problems can and have been managed well enough for useful area sampling systems to be built, we know of no solution that meets our requirements of *compatibility, high quality,* and *smooth performance/quality tradeoff.* Compatibility is compromised because hardware that is fundamentally designed to do Point Sampling is being used to do area sampling, and because the Z-buffer hardware is no longer useable (the pieces must be sorted, either prior to rendering or in the framebuffer itself). High quality is compromised for all the reasons listed above. Additionally, a smooth tradeoff between performance and quality is unlikely as there is no obvious parameter to vary.



3.2 The Accumulation Buffer

Solution 2, the integration of multiple point samples taken in the region of each pixel, is typically thought to require scan conversion and storage of multiple samples per pixel in a single rendering pass. The availability of hardware that renders roughly 1,000,000 Point Sampled polygons per second, however, allows an alternative implementation to be considered. Specifically, the Point Sampling hardware is used to render multiple images, each with the sample point jittered by a specific amount. These images are then integrated to form the final, antialiased result.

This basic technique for creating antialiased images has been described by others. [Fuchs 85] was the first to propose a successive refinement method that uses accumulation to create antialiased images by rendering the scene repeatedly with sub-pixel offsets. Also, [Deering 88] proposed that sub-pixel offsets could be jittered to further reduce aliasing, while [Mammen 89] described using alpha blending hardware to accumulate a series of images.

The Accumulation Buffer provides 16 bits to store each red, green, blue, and alpha color component, for a total of 64 bits per pixel. The primary operations that may be applied to the Accumulation buffer are:

1. *Clear.* The 16-bit components for red, green, blue and alpha are set to 0 for each pixel.
2. *Add with weight.* Each pixel in the drawing buffer is added to the Accumulation Buffer after being multiplied by a floating-point weight that may be positive or negative.
3. *Return with scale.* The contents of the Accumulation Buffer are returned to the drawing buffer after being scaled by a positive, floating-point constant.

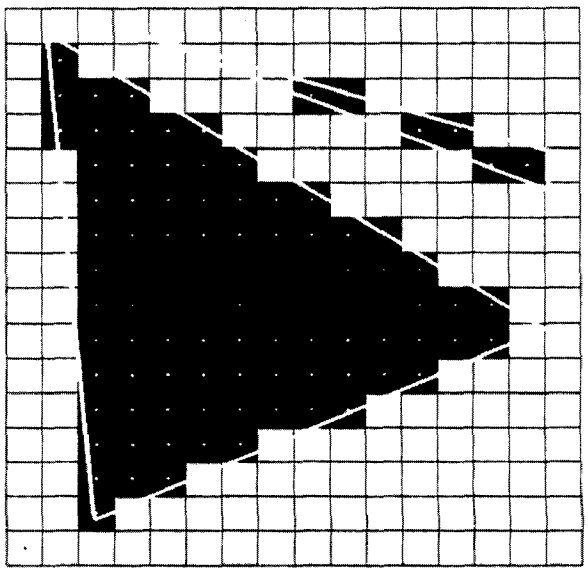


Figure 2. Point sampled polygons.

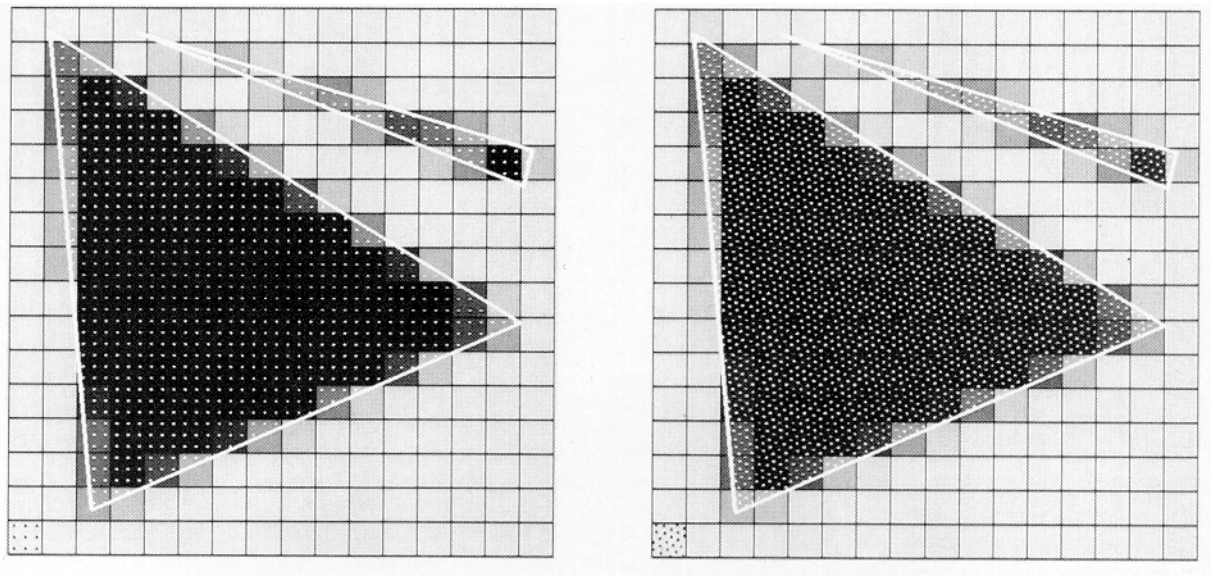


Figure 3. Super sampled polygons.

Each of these operations can be performed for all the pixels in a viewport on the screen at extremely high rates given the parallel architecture of modern framebuffers [Akeley 88].

3.3 Super Sampling

As a first example of the use of the Accumulation Buffer, consider duplicating the result of a single-pass super sample operation. Each pixel is sampled on an *nxm* regular grid. The process is begun by initializing the Accumulation Buffer, using the *Clear* command. The scene is then rendered *nxm* times, each time with a different subpixel offset. After each subpixel rendering is completed, it is added to the Accumulation Buffer using the *Add with weight* function, with *weight* set to one. When the final pass has been completed, the Accumulation Buffer contents are returned to the drawing buffer using the *Return with scale* operation, with *scale* set to *1/(nxm)*. The image is made visible by swapping the drawing and display buffers.

The results of 3 by 3 super-sampling with the Accumulation Buffer are shown on the left side of Figure 3. At the cost a finite amount of hardware (64 bits per pixel) and multiple passes through the data base, the Accumulation Buffer has reproduced the result of a very expensive multi-sample framebuffer. Because the Z-buffer is "reused" on each iteration, each sample is correctly depth buffered as well. From a programming standpoint, all that has been required is multiple passes through a data base, each preceded by a slight modification to the projection matrix [see appendix A for the arithmetic]. No change is made either to the data base itself, or to the process of its traversal. In particular, no sorting of the data is required, and all framebuffer algorithms that were used (blending, constructive solid geometry, etc.) continue to work identically.

There is no reason to be confined to regular *nxm* super-sampling in this architecture. The right side of Figure 3 shows geometry being sampled 23 positions per pixel. A relaxation technique can be used to automatically generate irregular super-sampling patterns for any sample count. To do this, N sample points are randomly distributed inside the area of the pixel. Then repulsion forces were calculated and each point is moved incrementally to respond to this force. By repeating this process, a distribution of sample points is obtained. Figure 4 shows sample patterns with several different numbers of samples per pixel.

3.4 Sampling with a Gaussian

In the limit, the above process will create antialiased images that are box filtered. However, there is no reason to limit samples to the area of a single pixel. Better results can be obtained by using other sampling functions. By distributing point samples in the region surrounding each pixel center, superior antialiasing results

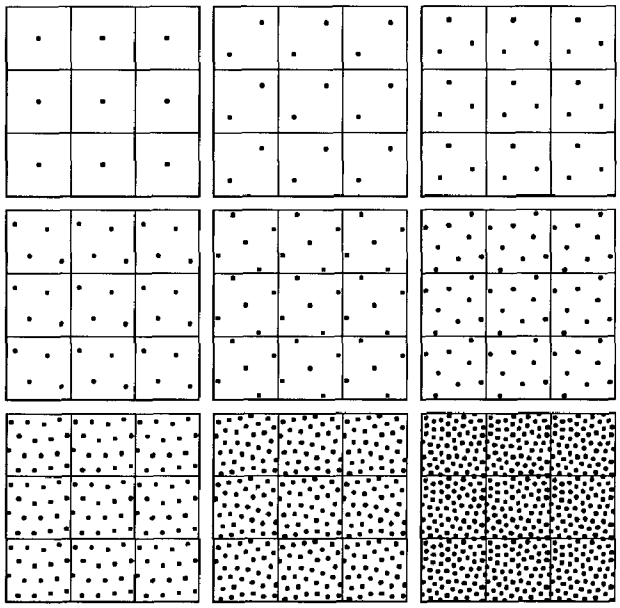


Figure 4. Sample patterns for 1, 2, 3, 4, 5, 8, 16, 33, and 60 samples per pixel.

can be obtained. Geometry can be sampled using a Gaussian (or other sample function) in three distinct ways.

1. The first technique is to distribute samples with an even density around each pixel center, and weight each sample using a Gaussian.
2. Another technique is to distribute samples using a Gaussian distribution and weight each sample equally. This implements *importance* sampling.
3. As an alternative we can use convolution. After each image is drawn into the drawing buffer, a 3 by 3 filter kernel is calculated based on the subpixel offset used to create the image. The drawn image is convolved by the kernel to distribute samples to neighboring pixels. Then this convolved image is added to the Accumulation Buffer.

Figure 5. illustrates the difference between the box and the Gaussian filters. The left part of this figure shows the point samples that contribute to a pixel when simple super-sampling (box sampling) is used, while the right part of this figure shows the sample contributions if Gaussian sampling is used.

To see the effectiveness of this antialiasing solution see Figure 6. This shows an infinite perspective checker-board sampled with 1, 4, 16, and 64 samples per pixel. The top row uses a box filter to sample the geometry, while in the bottom row a Gaussian filter is used.

4. Additional Applications

The integration capabilities of the Accumulation Buffer allow us to handle problems of motion blur, depth of field, and soft shadows as well. Several general solutions to these problems have been discussed in the past [Cook 84, Cook 86, Dippe 85].

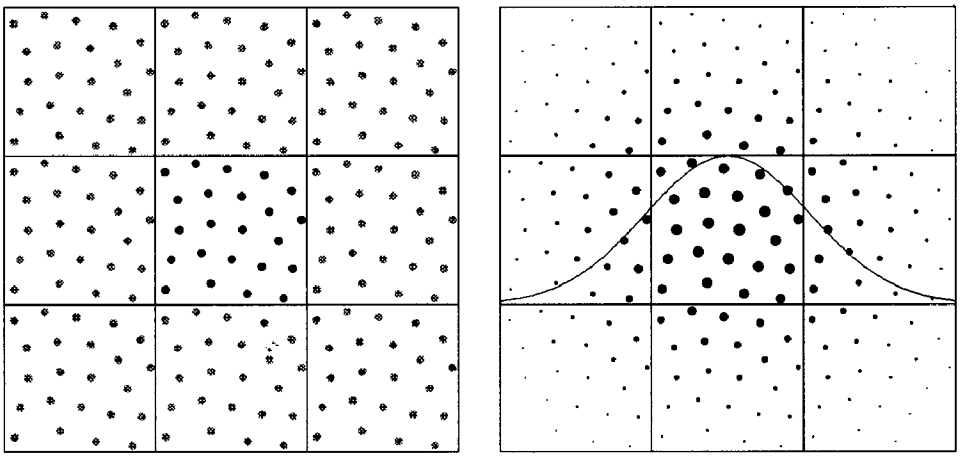


Figure 5. Box and Gaussian sampling geometry.

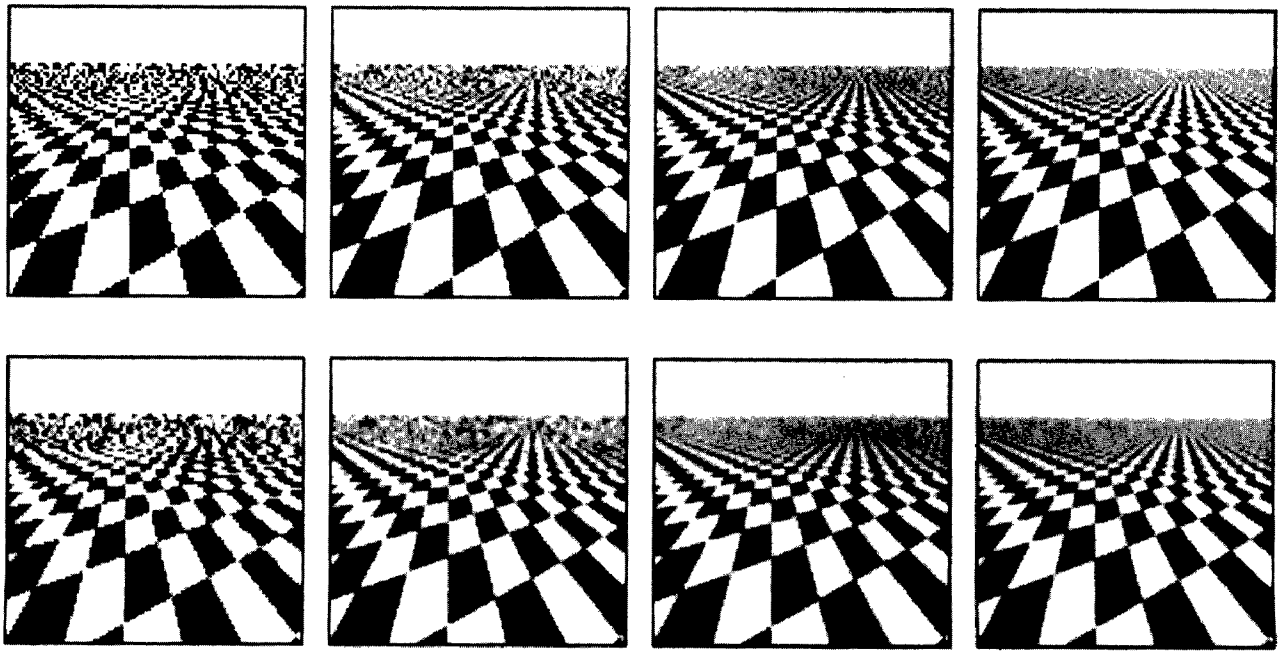


Figure 6. Antialiasing quality. The top row was created using a box filter while the bottom row shows the effect of using a Gaussian. The images were made with 1, 4, 16 and 64 samples per pixel.

These solutions use raytracing to perform Monte Carlo evaluation of the integrals in the rendering equation [Kajiya 86].

As an alternative, the Accumulation Buffer may be used to evaluate these integrals. We have already shown how the Accumulation Buffer can be used to integrate light arriving in the vicinity of each pixel. In the following section we show how it can be used to integrate light reflected by objects that move during the time the camer shutter is open to solve the problem of motion blur. In addition we show how is can be used to integrate light travelling along different paths through the camera lens to introduce depth of field, and also show how soft shadows can be created by integrating illumination from area light sources. These results are natural extensions of the work described in [Cook 84].

4.1 Motion Blur

In order to properly render objects with motion blur, we must integrate light reflected from all objects over the time that the camera shutter is open. Several different approaches to this problem have been explored. As mentioned above, stochastic sampling has been used in raytracers to sample rays over time to create motion blurred images. Another approach uses image processing techniques to provide an approximate solution [Potmesil 85]. Other solutions work well only if graphics primitives are drawn in priority order and do not self intersect [Max 85]. We would like to use a Z-buffering approach to solve this problem.

In order to create motion blurred images with the Accumulation Buffer, we follow almost the same procedure that was used above to perform antialiasing. However, instead of changing the subpixel offset on each pass, the geometry is allowed to move as the image is being accumulated. By accumulating a series of discrete still images, motion blur can easily be created as shown in Figure 7. This image was created by integrating 23 images.

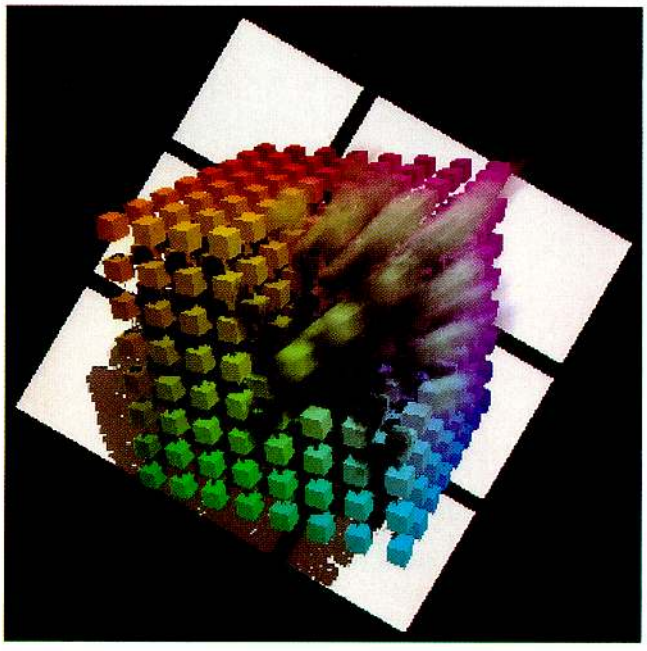


Figure 7. Motion blur image generated by accumulating 23 images.

If N images are integrated, This process will normally create a new image for display once every N frames. However, if higher update rates are desired, and all the frames are weighted equally, filtering by repeated integration [Heckbert 86] can be used to increase performance. To do this, we first accumulate N moving frames, and then display the accumulated image. Now the frame that was drawn N-1 frames ago is drawn, and it is subtracted from the Accumulation Buffer. Finally, the next frame is drawn added to the Accumulation Buffer. The result is a moving box average that is advanced one frame at a time. A new image can be displayed for every two images drawn.

4.2 Depth of Field

To create depth of field in an image, light traveling along different paths through the aperture of the lens must be integrated. Solutions to this problem include stochastic sampling using raytracing as described above, and an approximate solution provided by [Potmesil 82]. In our architecture, the projection matrix is modified as images are accumulated to view the scene from various discrete points across the aperture of the lens. The eye point is perturbed using a distribution of sample points as shown in Figure 8. For each sample point, the projection matrix is loaded, and the scene is drawn [see appendix B for the arithmetic]. After accumulating a number of images, the Accumulation Buffer will hold an image that demonstrates depth of field. Figure 9 shows an image created by accumulating 23 images.

4.3 Soft Shadows

To properly sample light sources requires that we integrate light emitted from the entire area of each light source. Again stochastic raytracing solves this problem as described above. Other solutions include [Brotman 84], where the scene is rendered with shadows while being illuminated from different points on the surface of each light source. And [Reeves 87] where shadow boundaries are blurred to simulate penumbra. To create proper penumbra, again we integrate a series of crude images.

We assume that the rendering hardware can render the scene illuminated by one light source with correct shadows. Given this capability, the solution is to repeatedly render the scene, while accumulating images. Each time the scene is drawn, a light source is chosen and a point on that light source is chosen. A

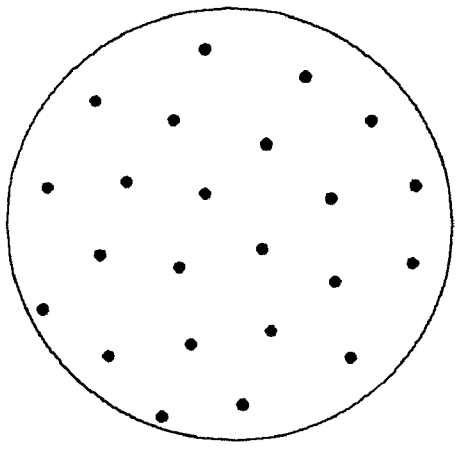


Figure 8. Positions of 23 sample locations used to sample the aperture of the camera lens.

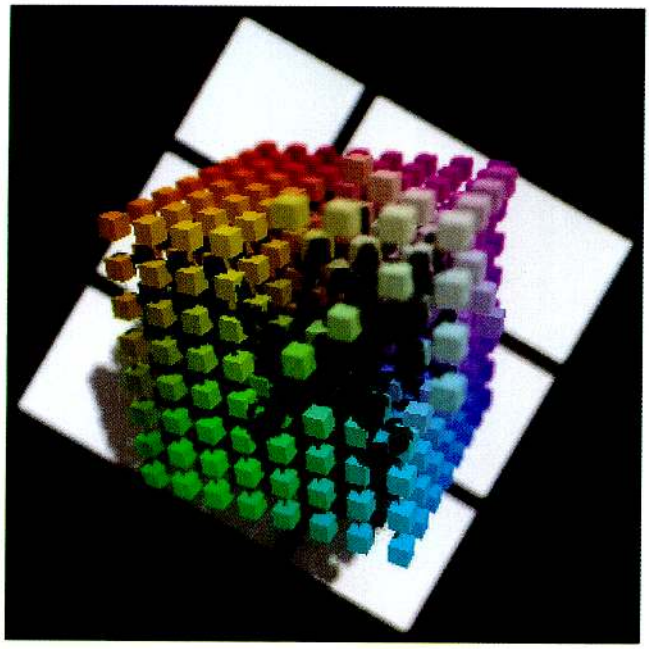


Figure 9. Depth of field image generated by accumulating 23 images.

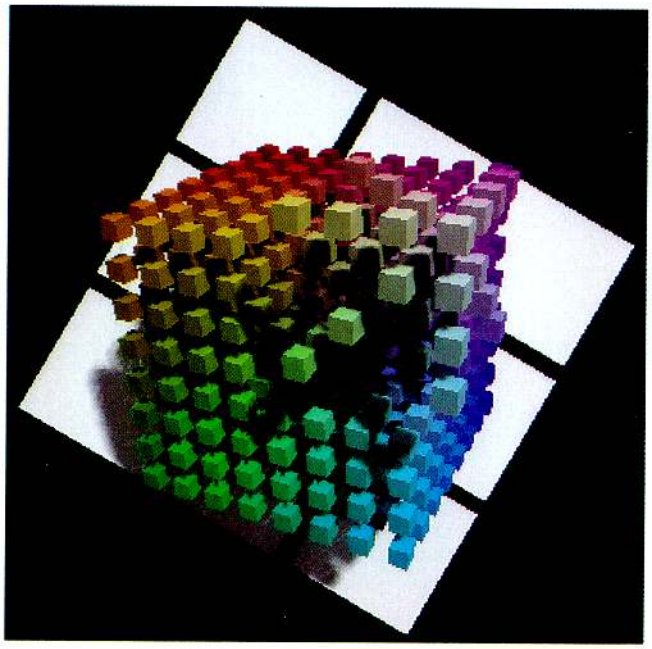


Figure 10. Soft shadows generated by accumulating 23 images.

series of images are integrated to create correct penumbra. Figure 10 shows the result of illuminating a scene with one light, and accumulating 23 images.

4.4 Doing it all

We have shown how the accumulation buffer provides solutions to the problems of spatial aliasing, motion-blur, depth of field, and penumbra. However, in each of the examples above only one problem was solved independently. For example, the motion-blur, depth of field, and penumbra images all demonstrate artifacts of spatial aliasing.

It is possible to solve all four problems at once. Figure 11 shows an image that demonstrates antialiasing, motion blur, depth of field, and soft shadows. It was created by using all the techniques described above simultaneously. To create this image 66 crude images were integrated. As these images were drawn, the subpixel offset was altered as the geometry was moved in equal steps. In addition, the projection matrix used to draw the images was modified to create depth of field, and the light source was sampled at different points to create soft shadows.

5. Discussion

Many other sampling problems are supported by the hardware architecture described here.

The Accumulation Buffer may be used to support anisotropic reflection models [Kajiya 85]. To do this, surface normals are distributed as images are accumulated. In addition, the Accumulation Buffer can be used to filter texture maps and environment maps. To do this, standard mip-mapping [Williams 83] is done, but no interpolation is performed to filter textures in x and y. As images are accumulated, texture maps are filtered along with the scene geometry. In this way, eight texture accesses can be replaced by one texture map access to improve drawing performance.

Alternate low cost implementations of the Accumulation Buffer can be created by reducing the number of bits in the drawing and the Accumulation Buffers. In such a system 4 bits are provided for each R, G, B and Alpha in the drawing buffer, and 8 bits are provided for each R, G, B and alpha in the Accumulation Buffer. High quality images are generated by dithering images into the 4 bit per component frame buffer and accumulating up to 16 images in the 8 bit per component accumulator. By carefully selecting a different 4 by 4 dither matrix on each pass, most dithering artifacts can be removed from the accumulated images.

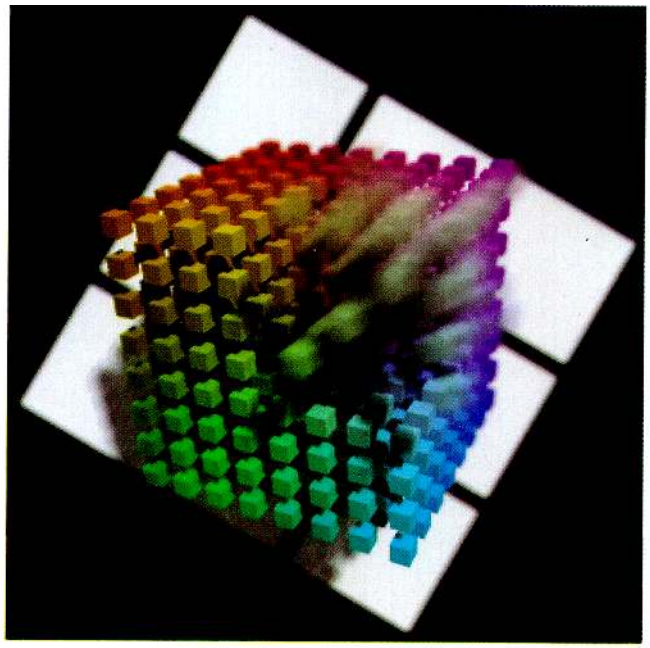


Figure 11. Antialiased image with motion blur, depth of field, and soft shadows generated by accumulating 66 images.

Figure 12. Environment mapped image with motion blur, and depth of field generated by accumulating 24 images.

6. Conclusion

The Accumulation Buffer meets our antialiasing requirements of compatibility, high quality, and a smooth tradeoff between performance and quality:

Figure 12 shows a more complex scene generated using the Accumulation Buffer.

1. *Compatibility.* Both the spectacular polygon performance and the accurate Point Sampling of modern workstation graphics systems are leveraged by the multi-pass Accumulation Buffer algorithm. Use of the Accumulation Buffer is independent of all other framebuffer algorithms - it simply improves the quality of any image that can be generated.
2. *High Quality.* There is no practical limit to the image quality that can be obtained with the accumulation buffer. If more than 256 samples are desired, a *weight* less than 1.0 can be used to avoid overflow. Because each sample is computed exactly, both in terms of parameter interpolation and depth buffering, only the number of samples taken limits image quality.
3. *Smooth performance/quality tradeoff.* Performance and quality trade off smoothly as the number of samples per pixel is increased from 1. Because the Accumulation Buffer hardware is separate from the normal rendering hardware, the performance of that hardware is not compromised.

These attributes, in addition to simplicity, elegance, and general purpose application, make the Accumulation Buffer a desirable architectural enhancement to a workstation graphics system.[1]

7. Acknowedgements

We thank the entire VGX graphics group at Silicon Graphics for supporting this work, and appreciate the comments of the reviewers.

1. Some aspects of this work are patent pending.

8. Appendix A

The following function is used to specify a simple perspective projection in the SGI Graphics Library.

```
window(left,right,bottom,top,near,far)
float left, right, bottom, top, near, far;
{
   float Xdelta, Ydelta, Zdelta;
   float matrix[4][4];

   Xdelta = right - left;
   Ydelta = top - bottom;
   Zdelta = far - near;
   matrix[0][0] = (2.0*near)/Xdelta;
   matrix[0][1] = 0.0;
   matrix[0][2] = 0.0;
   matrix[0][3] = 0.0;
   matrix[1][0] = 0.0;
   matrix[1][1] = (2.0*near)/Ydelta;
   matrix[1][2] = 0.0;
   matrix[1][3] = 0.0;
   matrix[2][0] = (right+left)/Xdelta;
   matrix[2][1] = (top+bottom)/Ydelta;
   matrix[2][2] = -(far+near)/Zdelta;
   matrix[2][3] = -1.0;
   matrix[3][0] = 0.0;
   matrix[3][1] = 0.0;
   matrix[3][2] = -(2.0*far*near)/Zdelta;
   matrix[3][3] = 0.0;
   loadmatrix(matrix);
}
```

The **window** command above creates a projection matrix that specifies the position and size of the rectangular viewing frustum in the near clipping plane, and the location of the far clipping plane. All objects contained within this volume are projected in perspective onto the screen area of the current viewport.

Subpixwindow, below, duplicates the functionality of **window**, and includes parameters that specify a subpixel offset in screen *x* and screen *y*. This function supports subpixel positioning for antialiasing.

```
subpixwindow(left,right,bottom,top,near,far,pixdx,pixdy)
float left, right, bottom, top, near, far, pixdx, pixdy;
{
   short vx1, vx2, vy1, vy2;
   float xwsize, ywsize, dx, dy;
   int xpixels, ypixels;

   getviewport(&vx1,&vx2,&vy1,&vy2);
   xpixels = vx2-vx1+1;
   ypixels = vy2-vy1+1;
   xwsize = right-left;
   ywsize = top-bottom;
   dx = -pixdx*xwsize/xpixels;
   dy = -pixdy*ywsize/ypixels;
   window(left+dx,right+dx,bottom+dy,top+dy,near,far);
}
```

First the pixel size of the viewport is determined. Then delta x and delta y values that incorporate the subpixel offset are calculated. Finally the projection matrix is set using the **window** function.

9. Appendix B

The **genwindow** function below extends the functionality of **subpixwindow** to support depth of field.

```
genwindow(left,right,bottom,top,near,far,pixdx,pixdy,
                                    lensdx,lensdy,focalplane)
float left, right, bottom, top, near, far, pixdx, pixdy;
float lensdx, lensdy, focalplane;
{
   short vx1, vx2, vy1, vy2;
   float xwsize, ywsize, dx, dy;
   int xpixels, ypixels;

   getviewport(&vx1,&vx2,&vy1,&vy2);
   xpixels = vx2-vx1+1;
   ypixels = vy2-vy1+1;
   xwsize = right-left;
   ywsize = top-bottom;
   dx = -(pixdx*xwsize/xpixels + lensdx*near/focalplane);
   dy = -(pixdy*ywsize/ypixels + lensdy*near/focalplane);
   window(left+dx,right+dx,bottom+dy,top+dy,near,far);
   translate(-lensdx,-lensdy,0.0);
}
```

First the pixel size of the viewport is determined. Delta x and delta y values that incorporate the subpixel offset are calculated. The projection is then sheared to change the viewpoint position based on the lens *x* and *y* offsets, and set using the **window** function. Finally the projection is translated to insure that objects in the focal plane remain stationary.

10. References

1. [Akeley 88] Kurt Akeley, and Tom Jermoluk, "High-Performance Polygon Rendering", Computer Graphics, 1988.
2. [Apgar 88] Brian Apgar, et al., "A Display System for the Stellar Graphics Supercomputer Model GS1000", Computer Graphics, 1988.
3. [Brotman 84] Lynne Shapiro Brotman and Norman I. Badler, "Generating Soft Shadows with a Depth Buffer Algorithm", IEEE CG+A October, 1984.
4. [Carpenter 84] Loren Carpenter, "The A-buffer, an Antialiased Hidden Surface Method" Computer Graphics, 1984.
5. [Cook 84] Robert L. Cook et al., "Distributed Ray Tracing", Computer Graphics, 1984.
6. [Cook 86] Robert L. Cook, "Stochastic Sampling in Computer Graphics", ACM Transactions on Graphics, January, 1986
7. [Deering 88] Michael Deering, et al., "The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics", Computer Graphics, 1988.
8. [Dippe 85] Mark A. Z. Dippe' and Erlin Henry World, "Antialiasing Through Stochastic Sampling", Computer Graphics, 1985.
9. [E&S 87] Evans and Sutherland, PS 390 Marketing Brochure, 1987.
10. [Fuchs 85] Henry Fuchs, et al., "Fast Spheres, Shadows, Texture, Transparencies, and Image Enhancements in Pixel-Planes", Computer Graphics, 1985.
11. [Goldfeather 86] Jack Goldfeather, et al., "Fast Constructive-Solid Geometry Display in the Pixel-Powers Graphics System", Computer Graphics, 1986.
12. [Heckbert 86] Paul S. Heckbert, "Filtering by Repeated Integration", Computer Graphics, 1986.
13. [Kajiya 85], James T. Kajiya, "Anisotropic Reflection Models", Computer Graphics, 1985.
14. [Kajiya 86] James T. Kajiya, "The Rendering Equation" Computer Graphics, 1986.
15. [Mammen 89] Abraham Mammen, "Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique", IEEE CG+A, July 1989.
16. [Max 85] Max, Nelson L., and Douglas M. Lerner, "A Two-and-a-Half-D Motion Blur Algorithm", Computer Graphics, 1985.
17. [Potmesil 82] Potmesil, Michael, and Indranil Chakravarty, "Synthetic Image Generation with a Lens and Aperture Camera Model", ACM Transactions on Graphics, April 1982.
18. [Potmesil 83] Potmesil, Michael and Indranil Chakravarty, "Modeling Motion Blur in Computer Generated Images", Computer Graphics, 1983.
19. [Potmesil 89] Michael Potmesil, and Eric M. Hoffert, "The Pixel Machine: A Parallel Image Computer", Computer Graphics, 1989.
20. [Reeves 87] William T. Reeves, et al., "Rendering Anti-Aliased Shadows with Depth Maps", Computer Graphics, 1987.
21. [SGI 85] Silicon Graphics, "Silicon Graphics 3000 Technical Report", 1985.
22. [Star 89] Star Technologies, "Graphicon 2000 Technical Overview", 1989.
23. [Swanson 86] Roger W. Swanson, and Larry J. Thayer, "A Fast Shaded-Polygon Renderer", Computer Graphics, 1986.