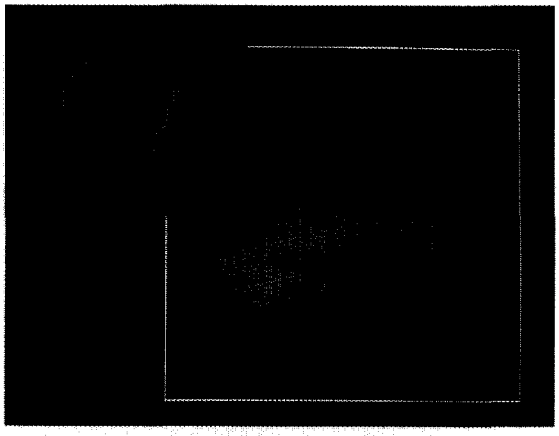
Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique

Abraham Mammen
Stellar Computer

This article describes the benefits of Stellar's Virtual Pixel Maps technique in the support of high-quality rendering operations. A dedicated frame buffer with a fixed number of bits per pixel is inappropriate for implementing high-quality rendering techniques within the framework of a graphics computer system. The Virtual Pixel Maps feature is an elegant abstraction for solving problems that inherently require a large number of bits per pixel. A system implementation is presented to illustrate the types of hardware rendering algorithms that benefit from the concept of Virtual Pixel Maps.



Several high-quality rendering algorithms attempt to present a form of realism typically lacking in most computer-generated graphics displays. Visual cues that portray depth of field, lighting and optical effects, shadows, material properties, physical phenomena, etc., aid tremendously in the overall perception of an image. Unfortunately, rendering systems that are good at displaying shaded geometrical constructs are not well suited for solving the special needs of high-quality rendering effects. This is primarily because these algorithms require a considerable amount of information per pixel and need constructs at the pixel level typically unavailable in most rendering systems. A dedicated frame buffer with a fixed number of bits per pixel is highly restrictive in supporting these algorithms. A system with a virtual frame buffer does not have such restrictions; this article attempts to establish the benefits of such a device.

The application views the Virtual Pixel Map architecture as regular virtual memory; it is a resource that is dynamically available to assign an almost arbitrary number of attributes per pixel. Instead of viewing a pixel just as having color and depth information, the Virtual Pixel Map concept allows the definition of a pixel to be whatever the application requires. With this flexibility, designers of rendering algorithms can use methods that are inherently fast and simple, but require a considerable amount of memory. Rendering techniques that were previously too computationally expensive can now be efficiently managed within the framework of a graphics computer system.[1]

This article is limited to techniques of implementing high-quality antialiased transparency rendering algorithms, although other rendering operations---environment mapping, shadows, image processing, etc.---can be easily implemented using the Virtual

0272-17-16/89/0700-0043$01.00 ©1989 IEEE

Pixel Maps architecture. Several of the techniques presented here are used by the graphics hardware rendering system on the Stellar Graphics Supercomputer Model GS1000.[2,3]

Transparency

Transparency effects are synthesized by linearly combining intensity contributions from the two nearest pixels in z space as

$$I = tI_1 + (1 - t)I_2, 0 \leq t \leq 1$$

where I_1 is the intensity of the pixel closer to the eye point, I_2 is the intensity of the pixel immediately behind it, and t is the transparency factor. If $t = 0$, the pixel is invisible. If $t = 1$, the pixel is opaque.[4]

The transparency factor models the characteristics of the material of the object and is usually specified in one of two ways: either as a constant term for the entire object or in some nonlinear fashion over the surface of the object.[4] For the latter case, one such criterion could be based on the curvature of the object; hence the transparency factor would be a function of the surface normal. From the rendering system's standpoint, this nonlinearity is modeled by computing the transparency factor explicitly at points on the object (on the basis of some physical characteristic being modeled) and then linearly interpolating across the geometry. This is similar to lighting calculations computed at the vertices of polygons and then internally interpolated. In the simple model, the rendering system associates a constant transparency factor for the entire object.

Description of the algorithm

To render transparent objects correctly, it is important to process pixels in a depth-sorted order, so that we can incrementally obtain contributions from all the transparent layers in the scene. It is especially difficult to incorporate transparency in a hidden-surface algorithm that uses z-buffering, because the rendering is performed in no specific order. What makes z-buffering an attractive technique for doing hidden-surface removal becomes a major drawback for algorithms that inherently function on the basis of some form of sorting operation. This is especially true for objects that intersect as well as interpenetrate each other. It is extremely difficult to do object-level sorting at the application level, so that objects are presented to the rendering system in a back-to-front order.

We would like to find a solution that does not force the application to do depth sorting but still uses z-buffering as a hidden-surface removal tool for all the simplicity it provides. The Virtual Pixel Maps technique is an ideal vehicle for solving such pixel-intensive algorithms. We can reduce a difficult problem to a series of simpler problems by performing sorting at the pixel level and accumulating the transparency effect on a multipass basis. For each pass, the objective is to find all the transparent pixels that are closest to the opaque pixels by sorting the pixels in depth order. At this point, we blend the transparent and opaque pixels. Now the farthest transparent pixel becomes the new opaque pixel. This, in effect, is a moving-depth algorithm, where the pixel depth being processed moves toward the eyepoint as transparent layers are resolved.

We assume that the transparent objects in a scene are tagged separately, so that only the opaque objects are initially rendered into the opaque pixel maps. For each pass of the transparent objects, we would like to find the set of transparent pixels closest to (in front of) the corresponding set of opaque pixels (see Figure 1). The sorting operation is performed with two depth pixel maps: the normal opaque depth pixel map and a sort depth pixel map. As each transparent pixel is processed, the current computed depth is compared with the stored opaque depth and the stored sort depth (see Figure 2a). If the current depth is in front of the opaque depth and behind the sort depth, then this transparent pixel is closer to the opaque pixel and hence becomes the new sort pixel (i.e., the current depth, intensity, and alpha values are stored in the respective sort pixel maps). Transparent pixels behind opaque pixels are trivially rejected. After all the transparent pixels have been rendered, the pixels in the sort pixel maps represent those that are closest to the opaque pixels. Now we can blend the opaque and the sort intensity pixel maps and also move the opaque depth closer to the eye position by updating the opaque depth from the stored sort depth value (Figure 2b). This operation continues until all the transparent layers are resolved. At each pixel, we store the following attributes (see Figure 2):

- Opaque depth
- Opaque intensity
- Sort depth
- Sort intensity
- Sort transparency factor (alpha)

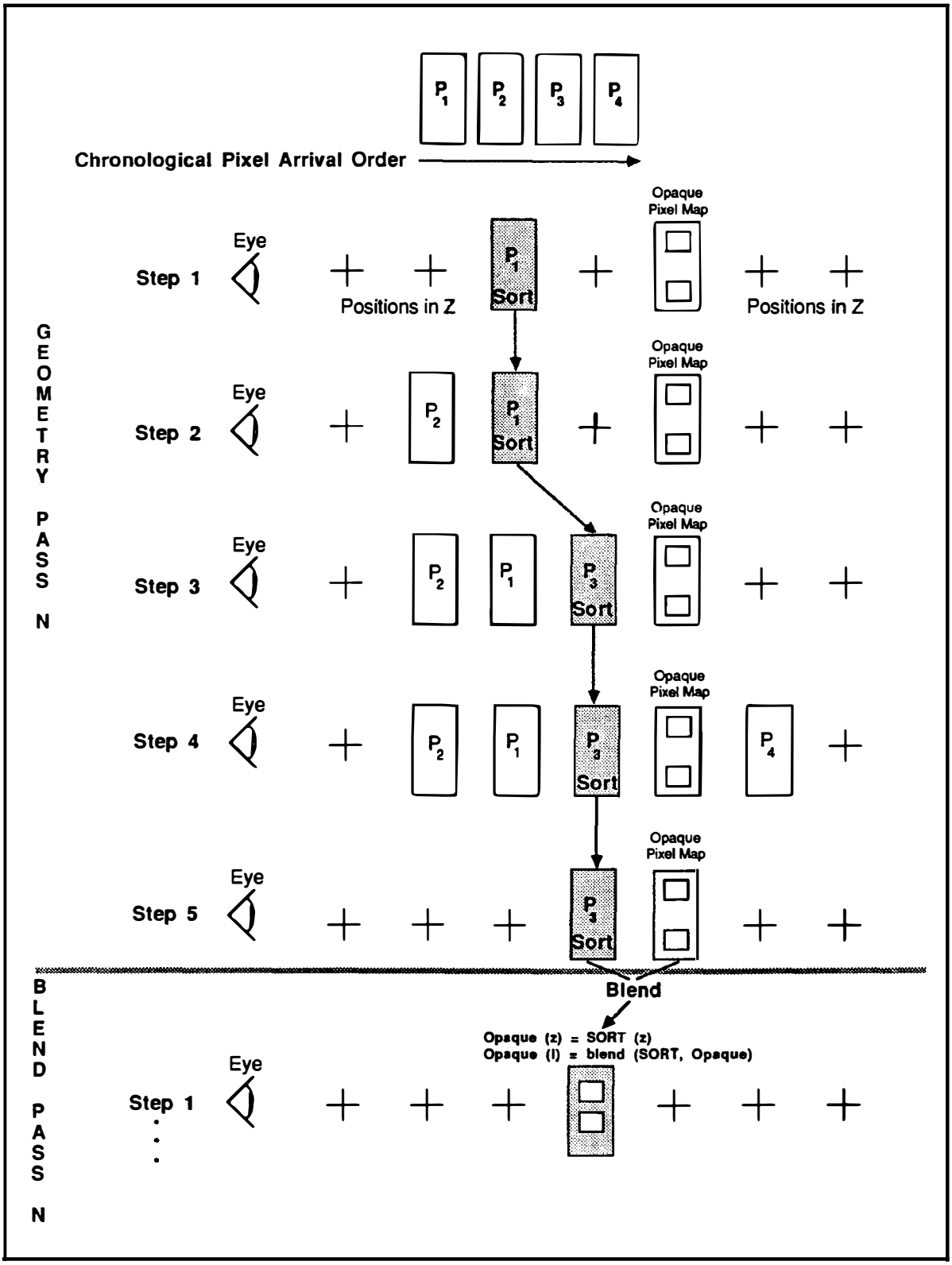


Figure 1. Finding the transparent pixel closest to the opaque pixel.

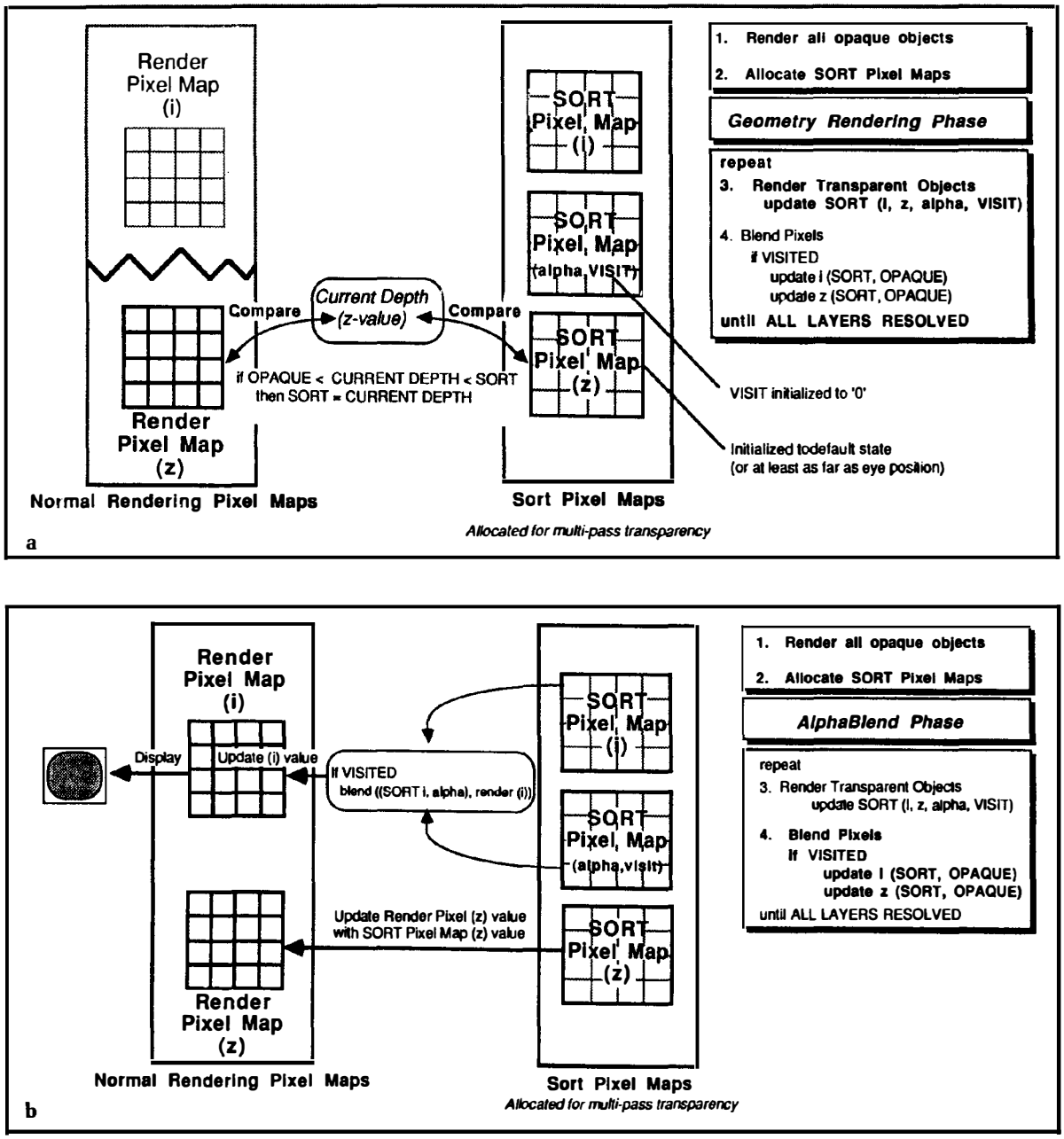


Figure 2. Multipass transparency: (a) geometry rendering phase, (b) alpha blend phase.

The number of passes needed for completion is a function of the maximum number of transparent layers at any pixel. At the end of each pass, the graphics application needs to know whether additional passes are required. The rendering stage of the graphics pipeline supplies a flag, which is queried to determine when to terminate. To keep track of the number of unresolved layers, the sort pixel maps also store a visit flag, which is set whenever the z comparison tourney finds a transparent pixel in front of the opaque pixel. During the pixel-map-blending operation, the rendering stage accumulates the number of pixels that were visited, a state available for the graphics application to query. The simplest method is for the application to continue re-rendering the scene until the total number of pixels having transparent layers reaches some

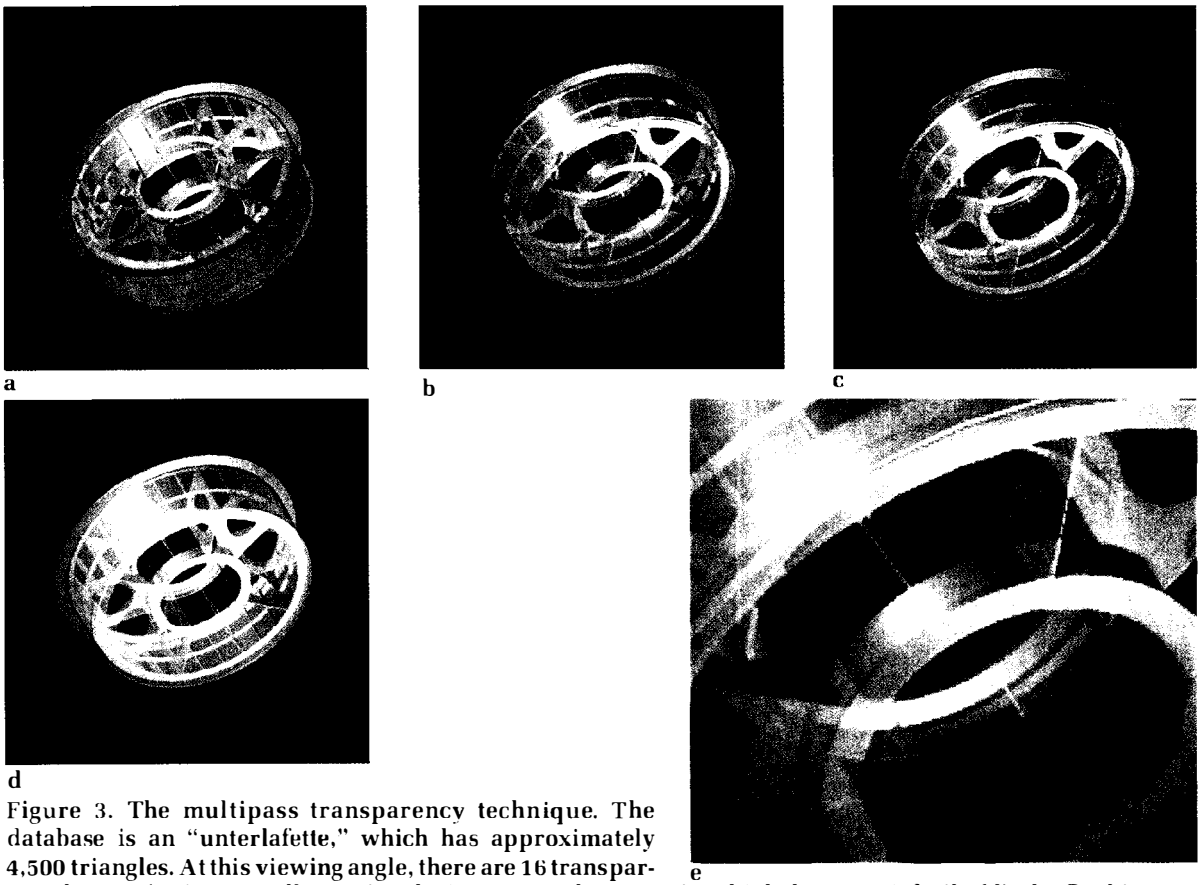


Figure 3. The multipass transparency technique. The database is an "unterlafette," which has approximately 4,500 triangles. At this viewing angle, there are 16 transparency layers: (a-c) Passes illustrating the incremental manner in which the scene is built. (d) The final image after 16 passes. The image was also antialiased by rendering the geometry nine times using a 3 × 3 triangular filter. Thus 144 passes (16 × 9) were needed to generate the image. (e) A magnified region of (d) showing the overall effect of transparency and antialiasing.

threshold level. Figure 3 illustrates how a scene is incrementally synthesized.

Summary of operations

The multipass technique is summarized by the following sample pseudocode:

Definitions

```
Zcomputed:   depth at current pixel
Icomputed:   intensity at current pixel
Acomputed:   alpha at current pixel
Zsort:       depth stored in ZSORT pixel
               map
Isort:       intensity stored in ISORT
               pixel map
Asort:       alpha stored in ASORT pixel
               map
Vsort:       visit flag in VSORT pixel
               map
PixelCount:  total count of outstanding
               transparent layers
Zopaque:     depth stored in ZOPAQUE
               pixel map
Iopaque:     intensity stored in IOPAQUE
               pixel map
```

Typical sequence

```
GetPixelMap (opaque_pix_map) ;
InitPixelMap (opaque_pix_map) ;
RenderOpaqueObjects () ;
GetPixelMap (sort_pix_map) ;
for (;;) {
  InitPixelMap (sort_pix_map) ;
  RenderTransparentObjects () ;
  BlendPixelMap
     (sort_pix_map,opaque_pix_map) ;
  Query (PixelCount) ;
  if (PixelCount < threshold) break ;
}
```

Rendering stage

```
for (transparent_objects)
  for (pixels_in_object)
    if ((Zcomputed > Zopaque)
        && (Zcomputed < Zsort)) {
      Vsort = 1 ;
      Zsort = Zcomputed ;
      Isort = Icomputed ;
      Asort = Acomputed ;
    }
```

Blending Stage

```
PixelCount = 0 ;
for (pixels_in_pixel_map)
  if (Vsort != 0) {
    Zopaque = Zsort ;
    Iopaque += Asort*(Isort - Iopaque) ;
    PixelCount++ ;
  }
```

Optimizations

The simple approach of rendering the entire scene for each transparent pass is not terribly efficient for the geometry and rendering pipeline. To improve efficiency, we dynamically reduce the number of transparent objects that need to be transformed and rendered as we proceed through the various passes. During the rendering phase, we determine if any portion of the object is within the domain of the opaque depth. If the entire object is behind the moving opaque depth space covered by the object, then that object need not participate any further in the transparency operation. If portions of the object are still in front of the opaque depth, then the object cannot be released from consideration and hence needs to be invoked for the next pass.

We assume that each object in the scene is identified by a pointer to its data structure. As part of the normal rendering operation of finding the transparent pixel closest to the corresponding opaque pixel, we can detect whether any portion of the object is in front of the current opaque layer. If all the pixels in the object are behind the corresponding opaque pixels, then the object cannot contribute toward resolving transparency and hence is tagged as *inactive*. If one or more pixels in the object lie in front of the opaque pixels, then the object is tagged as *active*. The state of the object is exported to the application by maintaining an object pointer list; the current object pointer is appended if the object is active; otherwise a null pointer is appended.

After rendering all the transparent objects in the scene, the application invokes the active objects in the list through the geometry and rendering pipeline. As the transparency layers get resolved, each pass requires fewer elements to be processed. For scenes where dense transparent layers exist in sparse areas, such a technique yields a significant performance improvement. Here is a typical sequence:

```
for (transparent_objects) {
  ObjectPointer = NULL ;
  for (pixels_in_object) {
    if ((Zcomputed > Zopaque)
        && (Zcomputed < Zsort)) {
      Vsort = 1 ;
      Zsort = Zcomputed ;
      Isort = Icomputed ;
      Asort = Acomputed ;
    }
    if (Zcomputed > Zopaque)
      ObjectPointer =
      CurrentObjectPointer ;
  }
  *PointerPixMap++ = ObjectPointer ;
}
```

Pixel map initialization

Between passes, the application reinitializes several pixel maps. The Zsort pixel map is initialized to a depth value at least as far as the eye position, so that during the rendering phase, the sorting operation correctly determines which transparent pixel is closest to the opaque pixel. Also, the Vsort pixel map is initialized to 0. The rendering stage marks pixels that have outstanding transparent layers by setting the Vsort value to 1.

Transparent spheres

The multipass technique described also applies to spheres. Since the transparent layers are resolved on a per-pixel basis, it is possible to correctly blend spheres that both intersect and interpenetrate each other. Transparent spheres are synthesized by rendering the front and rear shells, thus associating two distinct transparency layers for every pixel within the sphere. Polygons and spheres can be freely intermingled in the construction of the scene. As with polygons, opaque spheres are rendered before invoking transparent spheres. Figure 4 is an example of this

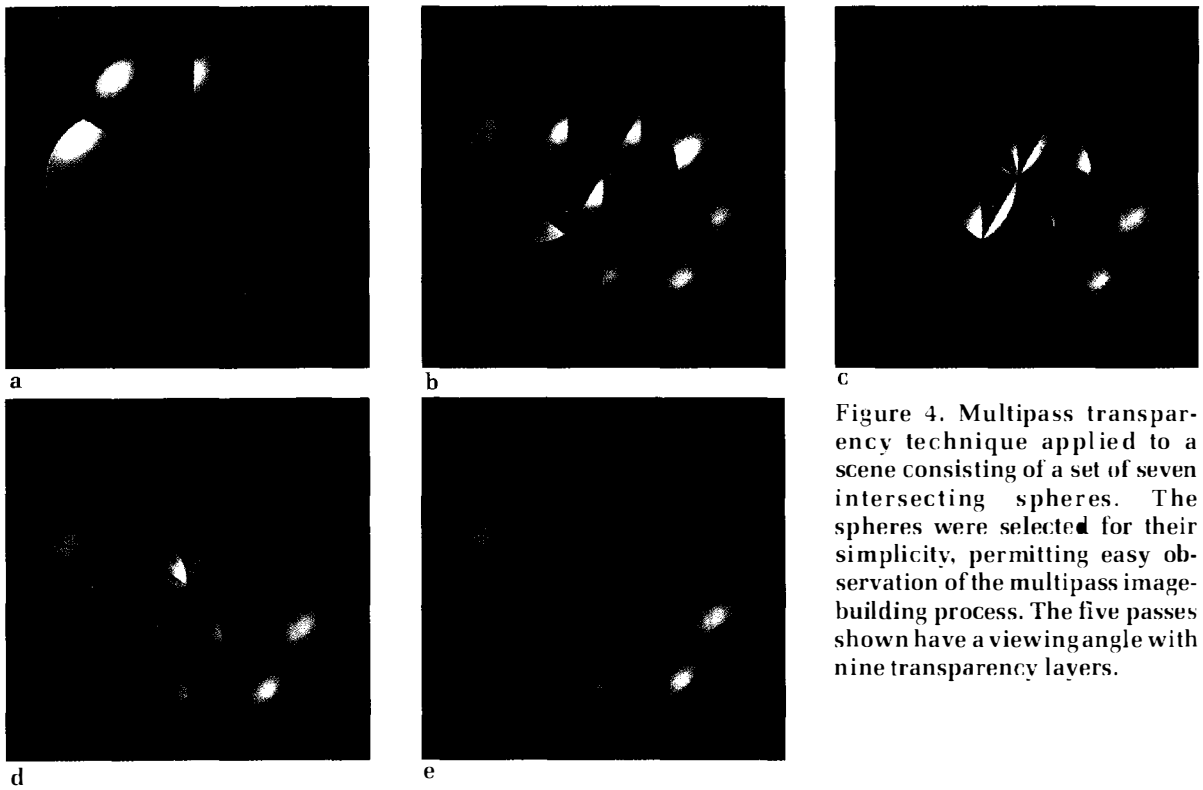


Figure 4. Multipass transparency technique applied to a scene consisting of a set of seven intersecting spheres. The spheres were selected for their simplicity, permitting easy observation of the multipass image-building process. The five passes shown have a viewing angle with nine transparency layers.

technique applied to a display consisting of a cluster of spheres that intersect each other.

Another method of implementing transparency

The above-mentioned transparency algorithm can be slightly modified to require fewer bits per pixel (and hence fewer pixel maps), at the expense of increasing the overall number of passes. In the previous implementation, the sort pixel maps were defined for storing the depth, intensity, alpha, and visit flag. Instead, we now have just one sort pixel map defined for the depth, which is used to determine the pixel closest to the currently defined opaque pixel; the sorting procedure is identical to the previous method. However, no lighting calculations are made while we find the farthest transparent pixel. Through a second pass, we re-render the scene. When the current depth matches the stored transparent depth in the sort pixel map, we know that this pixel is a valid candidate for the blending operation. The lighting model is applied at this stage, and as before we move the opaque depth toward the eye point by updating its value to the current depth.

This method requires two passes to resolve one layer of transparency, while the previous method requires only one pass. However, the former method requires a blending stage at the end of each pass, which operates over the entire space of the pixel map. The current method has instead a second rendering stage. This method performs *z*-buffering twice, and the blending operation is incorporated in the second rendering stage. The major benefit is that only one additional pixel map is needed, significantly improving the overall storage requirements.

There is the potential for somewhat reduced performance with this method, because on the average the second rendering stage takes longer than the pixel-map-blending operation. On the other hand, the first rendering stage is faster than the rendering stage used in the former method because lighting calculations are deferred. It is difficult to estimate which method will be faster from a system standpoint, especially since performance depends on geometry.

Summary of operations

The above-described method of implementing transparency is summarized as follows:

Rendering stage 1

```
for (transparent objects)
   for (pixels_in_object)
```

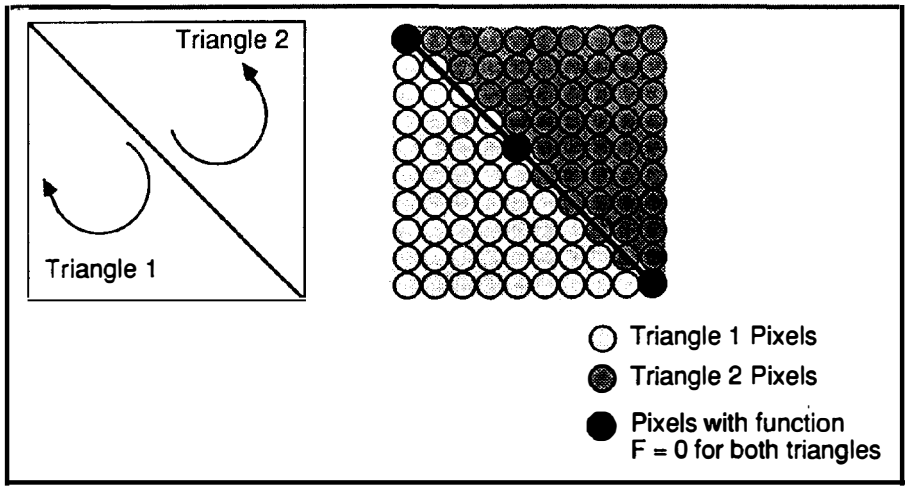


Figure 5. Multiple pixel visit problem.

```
if ((Zcomputed > Zopaque)
   && (Zcomputed < Zsort)) {
 Zsort = Zcomputed ;
}
```

Rendering stage 2

```
for (transparent_objects)
  for (pixels_in_object)
    if ((Zcomputed == Zsort) {
     Zopaque = Zsort ;
     Iopaque += Acomputed
        *(Icomputed-Iopaque) ;
     PixelCount++ ;
    }
```

Multiple pixel visit problem

Most rendering algorithms visit pixels multiple times along the common edge boundaries of polygons. For normal rendering operations, this does not present a serious problem, because the intensity and depth computations are so close for multiple instances of the same pixel that no noticeable artifacts are observed. Occasionally, for databases that have inconsistent surface normals, this results in erratic behavior, especially around silhouette edges. Visiting pixels multiple times for a transparency algorithm has drastic effects, because all such pixels will apparently have more transparent layers and consequently exhibit more intensity saturation than surrounding pixels. This translates to discrete bright spots, which, of course, are immediately noticeable.

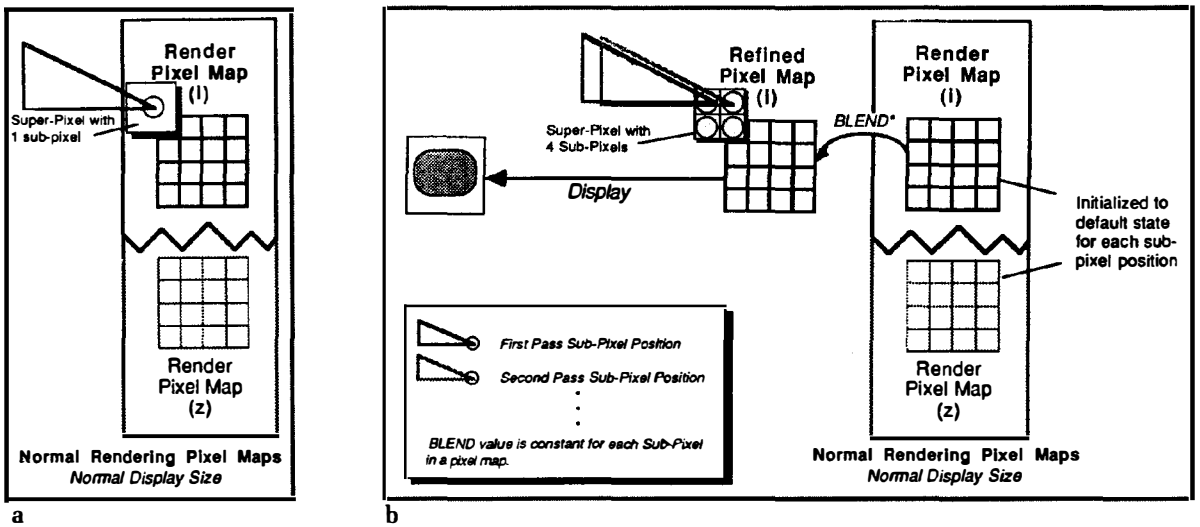
Let us examine why pixels are revisited along polygon boundaries. By constraining our discussion to triangles, we specify the geometry by a triplet of vertices $\{P_0, P_1, P_2\}$. Each side of the triangle is a half plane defined by its endpoints. The intersection of the three half planes specifies the pixels contained within the triangle.[5] The half plane equation is of the form

$$F(x,y) = Ax + By + C$$

where a point in space will lie on the half plane or on either side of it. The sign of the plane equation indicates the side of the plane that the point lies on. By orienting the half planes consistently we force the polarity of the plane equations to have the same sign for the region inside the triangle, and the opposite sign for points outside the triangle. Along a common edge, two adjacent triangles will have opposite orientations for the common edge-plane equation. The real problem is that the *equal-to-zero* condition (the case when a point lies exactly on the edge) exists on either side of the common edge for two adjacent triangles, and, hence, such pixels are visited twice (see Figure 5).

One way of solving this problem is to recognize that the depth computation at the revisited pixel is exactly the same along the common edge for both triangles; therefore we can prevent the pixel from being revisited by ignoring the equal-to-zero case in the *z*-buffering comparison operation. As long as the depth computations are performed accurately enough, we can get the depths to match at the same pixels where the plane equations become exactly equal to zero. This prevents pixels from being revisited along patch boundaries.

Another way of solving this problem is to discriminate on the basis of how plane equations are handled concerning the equal-to-zero condition. The common



a b

Figure 6. Rendering primitives (a) without antialiasing, (b) with antialiasing.

edge treatment for one triangle includes points that fall exactly on the edge, while the adjacent triangle excludes them. This method does not rely on depth computations to be accurate, but it does add another level of complexity to the rendering system.

The method presented here uses the *z*-buffering technique to exclude pixels that were previously visited as part of either the current object or a different object. As a result, additional transparency layers are not created if two or more pixels have the same depths.

Antialiasing

The Virtual Pixel Maps concept also helps to solve the problem of antialiasing. There are several known methods for solving the aliasing problem in computer-generated images.[6-8] One such method is to increase the spatial resolution, causing sample points to occur more frequently (this method is referred to as supersampling). Another method is to represent each sample point with a finite area, rather than an infinitesimally small spot (referred to as area sampling). We describe a multipass method that integrates the area-sampling technique over time.

In the supersampling approach, the scene is rendered with the intensity and depth calculations performed at the enlarged resolution. Then a convolution filter of the desired characteristics is applied to the supersampled display and filtered down to normal display dimensions. By definition, the intensity and depth buffers are larger by a factor equal to the size of the filter kernel.

The incremental approach described here uses one additional pixel map of normal display dimensions. The pixel map is used successively to refine the final image (see Figure 6). Instead of sampling the screen geometry at a higher integer resolution, we append additional bits of fraction at each screen position. We map the filter kernel to a default pixel area in screen space, where each filter coefficient is associated with a subpixel position in the assigned pixel geometry. The integration is achieved by blending contributions from each subpixel position, as each filter kernel coefficient is convolved.

A way of understanding this operation is to view a display pixel as a superpixel consisting of an array of subpixels. Each subpixel position is addressed by the fractional bits of the screen coordinate. Instead of computing a pixel intensity at a fixed position within a pixel (say the pixel center), we allow the geometry to intersect several subpositions within the pixel and accumulate the intensity contribution at each subposition (see Figure 6b). As the integration proceeds, this has the effect of smoothing transitions along edge boundaries (see Figure 7). It is important to realize that the fractional bits allow the geometry to be scanned at a higher resolution, but the pixels that are selected still fall on the normal resolution grid points.

As part of implementing this successive refinement algorithm,[5] the chosen filter coefficients are mapped to a corresponding set of blending coefficients (see the Appendix), which in effect become linear pixel map operators in blending the current and refined pixel maps. The scene is rendered several times, equal to the area of the filter kernel. Each pass is rendered into

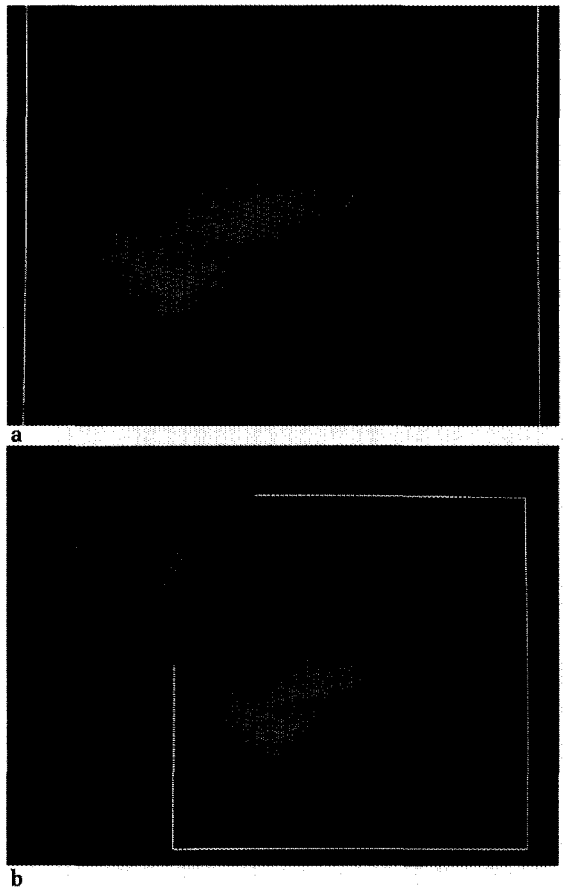


Figure 7. The multipass antialiasing technique. The database shows a Ford 2000X, which has approximately 75,000 triangles, rendered nine times using a 3×3 triangular filter: (a) the final display, (b) final display with rear wheel magnified to show the effects of antialiasing. (Image courtesy of David Godsell, Advanced Vehicle Simulation, Ford Motor Company.)

a scratch pixel map for a specific subpixel position. This operation is followed by the pixel-map operation that refines the final image. Before each rendering phase, the scratch and z pixel maps are initialized to their default state. The method can be summarized as follows:

```
GetPixelMap (scratch_pix_map,
   refine_pix_map) ;
for (i=0; i < kernel_x; i ++) {
  for (j=0; j < kernel_y; j ++) {
    InitPixelMap (scratch_pix_map) ;
    RenderObjects () ;
    RefinePixelMap(scratch_pix_map,
       refine_pix_map,blend_coeff) ;
    JitterInX ;
  }
  JitterInY ;
}
```

The benefits of this approach compared with the supersampling approach are obvious. The total amount of pixel map space is always a constant, regardless of the size of the filter kernel. In addition, the successive refinement technique allows the user to view the results as the image is integrated, without having to wait until the end of the rendering and filtering stages. The final image begins to look antialiased in the first few passes, especially if the subpixel positions are visited in a weighted manner. Because of the incremental approach, the user experiences an apparently faster image presentation than with the supersampling approach, even though the overall rendering time is nearly equivalent.

From an applications standpoint, the only difference between managing an antialiased image and one that is not is that the scene is rendered many times—the interfaces and data structures remain the same. This approach provides a single consistent interface to the rendering stage so that nothing special needs to be done when antialiasing is turned on. Because antialiasing is enabled so easily, applications can choose not to antialias, while, say, animation is in progress, and then enable antialiasing when the image reaches a quiescent state. Because of the incremental approach, the final image quality improves almost immediately. Finally, on most machines, the system implications of acquiring pixel maps that are 9, 16, or 25 times larger than the display buffer are severe. Memory limitations can have a drastic impact on the final perceived rendering performance.

Transparency with antialiasing

Combining antialiasing with transparency is relatively straightforward. Multiple passes of the transparency algorithm yield a pixel associated with a specific subpixel position; accumulation of the image for the other subpixel positions results in the final antialiased image. Algorithmically this can be expressed as follows:

```
GetPixelMap (refine_pix_map,
   opaque_pix_map) ;
for (i=0; i<kernel_x; i ++) {
  for (j=0; j<kernel_y; j ++) {
    InitPixelMap (opaque_pix_map) ;
    RenderOpaqueObjects ;
```

```
    GetPixelMap (sort_pix_map) ;
    for (;;) {
      InitPixelMap (sort_pix_map) ;
      RenderTransparentObjects ;
      BlendPixelMaps (sort_pix_map,
         opaque_pix_map) ;
      Query (PixelCount) ;
      if (PixelCount < threshold)break ;
    }
    ReleasePixelMap (sort_pix_map) ;
    RefinePixelMaps (opaque_pix_map,
       refine_pix_map,blend_coeff) ;
    JitterInX ;
  }
  JitterInY ;
}
```

Incorporating antialiasing in the rendering pipeline

The method presented here renders polygons through the notion of plane equations, where the geometry is defined to be contained within the union of the intersecting planes.[5] The basic operation of including additional fractional bits for each screen position is easily applicable to other scan-converting techniques.

To incorporate antialiasing in the framework of a rendering algorithm, we require that each vertex coordinate be specified with additional bits of fraction, which establish the contributions at a subpixel position. The fractional bits are incorporated in the computation of the plane equations, which by definition now have fractional and integer components. All the decision-making methodology for determining the pixels that are inside the polygon remains the same. If we assume that we need N bits of fraction for each of the x and y coordinates, the plane equations will require an additional $2N$ bits of range. If we assume that the largest pixel map we will consider is 4K × 4K, and we allow 4 bits of fraction for the coordinates, we need a maximum of 32(12 + 12 + 4 + 4) bits assigned to solve the plane equation.

As we jitter the image over the subpixel positions, the rendering algorithm offsets the vertex coordinates by the fractional coordinates of the current subpixel position, which in turn are incorporated into the algebraic equations. At a specific screen position, the plane equations are obtained as

$$F(\text{xint}, \text{yint}) = \text{xint}*A + \text{yint}*B + C - (\text{xfrac}*A + \text{yfrac}*B),$$

where $A = y_2 - y_1$, $B = x_1 - x_2$, $C = y_1*x_2 - x_1*y_2$,
xint = int(x = xvertex + xjitter),
yint = int(y = yvertex + yjitter),
xfrac = frac(x),
yfrac = frac(y).

In addition to adjusting the spatial coordinates to subpixel positions, we also need to adjust correctly the algebraic equations for intensity and depth. Similarly, it is desirable to associate additional bits of fraction for the lighting and depth components. Linearly interpolated (z,i) equations become

$$Z(\text{xint},\text{yint}) = Z(\text{vertex}) - (\text{xfrac}*\frac{dz}{dx} + \text{yfrac}*\frac{dz}{dy})$$

$$I(\text{xint},\text{yint}) = I(\text{vertex}) - (\text{xfrac}*\frac{di}{dx} + \text{yfrac}*\frac{di}{dy})$$

Conclusions

This article has described some of the rendering algorithms that benefit from the power of the Virtual Pixel Maps technique. Two specific algorithms, namely transparency and antialiasing, were picked as examples to illustrate the concept. Many other algorithms could easily be adapted to such an environment, where a large number of attributes are needed per pixel. It is important to recognize that since the rendering algorithms described here are implemented in hardware on a Stellar GS1000, the level of user interaction remains exceptional. Apgar[2] presents a detailed description of a hardware system (Stellar GS1000) that embodies the concepts described here.

Future work

The Virtual Pixel Maps feature can be used to produce shadow effects. Williams[9] presented a method of generating shadows in two passes: In the first pass, the scene is rendered from the light source's point of view such that the depth values closest to the light source are stored in a light depth pixel map. In the second pass, the scene is rendered with respect to the camera's position. A point on the object is mapped to a point in the light source space. The transformed depth is compared with the object depth closest to the light source, as stored in the light depth pixel map. The point is considered to be in shadow if the transformed depth is behind the stored depth. An intensity attenuation factor[10] can be derived that indicates the proportion of the surface in shadow, which is then used in the shading calculations. For scenes rendered

with multiple light sources, this process is repeated with separate light depth pixel maps assigned to each light source.

Similarly, the Virtual Pixel Maps concept can be used for such environment mapping functions as texture mapping and reflection mapping. Pixel maps store texture or reflection information. The rendering interface is provided with mapping parameters, specified on object vertices that map points in environment space to points in object space. As the mapping function is interpolated across the object, the environment colors are accessed from the pixel maps, which are then incorporated in the lighting and shading equations. ■

Acknowledgments

I would like to thank Brian Apgar for suggesting that I write this article. I also thank all the people in the graphics hardware group at Stellar for their comments and ideas while I was writing the article, as well as for their input while I was implementing these techniques on the Stellar GS1000. Special thanks go to George Mitsuoka for providing valuable assistance during the architectural and implementation stages. I would also like to thank Clare Campbell for her help in preparing this document.

References

1. A. Fournier and D. Fussell. "On the Power of the Frame Buffer," *ACM Trans. Graphics*, Vol. 7, No. 2, Apr. 1988, pp. 102-128.
2. B. Apgar, B. Bersack, and A. Mammen. "A Display System for the Stellar Graphics Supercomputer Model GS1000," *Computer Graphics* (Proc. SIGGRAPH), Vol. 22, No. 4, Aug. 1988, pp. 255-262.
3. M. Sporer, F. Moss, and C. Mathias, "An Introduction to the Architecture of the Stellar Graphics Supercomputer," *Digest of Papers Compcon 88*, CS Press, Los Alamitos, Calif., 1988, pp. 464-467.
4. D.S. Kay and D. Greenberg, "Transparency for Computer Synthesized Images," *Computer Graphics* (Proc. SIGGRAPH), Vol. 13, No. 2, Aug. 1979, pp. 158-164.
5. H. Fuchs et al., "Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes," *Computer Graphics* (Proc. SIGGRAPH), Vol. 19, No. 3. July 1985, pp. 111-120.
6. L. Carpenter, "The A-buffer, an Antialiased Hidden Surface Method," *Computer Graphics* (Proc. SIGGRAPH), Vol. 18, No. 3, July 1984, pp. 103-108.
7. F. Crow, "The Aliasing Problem in Computer Generated Shaded Images," *CACM*, Vol. 20, No. 11, Nov. 1977, pp. 799-805.
8. F.C. Crow, "A Comparison of Antialiasing Techniques," *CG&A*, Vol. 1, No. 1, Jan. 1981, pp. 40-48.
9. L. Williams, "Casting Curved Shadows on Curved Surfaces," *Computer Graphics* (Proc. SIGGRAPH), Vol. 12, No. 3, Aug. 1978, pp. 270-274.
10. W. Reeves, D. Salesin, and R. Cook. "Rendering Antialiased Shadows with Depth Maps," *Computer Graphics* (Proc. SIGGRAPH), Vol. 21, No. 4, July 1987, pp. 283-291.

Appendix: Blending coefficients

As discussed earlier, we need to calculate a set of blending coefficients that maps to the chosen set of filter coefficients. The blending function operates between two image pixel maps in the following manner:

$$I_n = \alpha R + \beta I_{n-1}$$

where R is the currently rendered pixel map, I is the destination pixel map, which is the pixel map that is being successively refined, and $\{\alpha, \beta\}$ are the blending coefficients.

The final pixel intensity is determined as

$$I = \sum_{n=0}^{N} F_n I_n$$

where the filter coefficient F_n is applied to I_n, the intensity at pixel position P_n.

It is desirable to maintain the final value of I to be in the same numeric range as I_n, and hence the filter coefficients $\{F_n\}$ should be normalized such that

$$\sum_{n=0}^{N} F_n = 1$$

For each pass of the filter kernel, we want to determine a pair of $\{\alpha_n, \beta_n\}$ that successively blends the

image. We need to map a set of $\{F_n\}$ to a set of $\{\alpha_n, \beta_n\}$ for a set of subpixel positions $\{P_n\}$:

$$I_0 = \alpha_0 R_0, \text{ with } \beta_0 = 0$$
$$I_1 = \alpha_1 R_1 + \beta_1 I_0 = \alpha_1 R_1 + \beta_1 \alpha_0 R_0$$
$$I_2 = \alpha_2 R_2 + \beta_2 I_1 = \alpha_2 R_2 + \beta_2 \alpha_1 R_1 + \beta_2 \beta_1 \alpha_0 R_0$$
$$\vdots$$

Generalizing, we get

$$I_N = \alpha_N R_N + \beta_N \alpha_{N-1} R_{N-1} + \beta_N \beta_{N-1} \alpha_{N-2} R_{N-2} + \cdots +$$
$$\vdots$$
$$\beta_N \beta_{N-1} \beta_{N-2} \cdots \beta_1 \alpha_0 r_0$$

As we successively refine the image, it is desirable to maintain normalized intensity ranges across each pass. This implies that

$$\alpha_n + \beta_n = 1, \text{ with } \beta_0 = 0$$

After N passes, the final refined image is the same as the image produced by applying the convolution filter to the supersampled image. Hence

$$I_N = \sum_{n=0}^{N} F_n I_n$$

We can now map the filter coefficients $\{F_n\}$ to $\{\alpha_n, \beta_n\}$:

$$F_N = \alpha_N$$
$$F_{N-1} = \beta_N \alpha_{N-1}$$
$$F_{N-2} = \beta_N \beta_{N-1} \alpha_{N-2}$$
$$\vdots$$
$$F_0 = \beta_N \beta_{N-1} \beta_{N-2} \cdots \beta_0 \alpha_0$$

Given that $\alpha_n + \beta_n = 1$, we have

$$\alpha_N = F_N, \ \beta_N = 1 - F_N$$
$$\alpha_{N-1} = \frac{F_{N-1}}{\beta_N} = \frac{F_{N-1}}{1 - F_N}$$
$$\alpha_{N-2} = \frac{F_{N-2}}{\beta_N \beta_{N-1}} = \frac{F_{N-2}}{1 - F_N - F_{N-1}}$$

Generalizing,

$$\alpha_n = \frac{F_n}{1 - F_N - F_{N-1} \cdots F_{n+1}}$$

Since $\sum_{n=0}^{N} F_n = 1$

$$\alpha_n = \frac{F_n}{F_N + F_{N-1} + \cdots + F_0 - F_N - \cdots - F_{n-1}}$$
$$= \frac{F_n}{F_0 + F_1 + \cdots + F_n}$$
$$\beta_n = 1 - \alpha_n = 1 - \frac{F_n}{F_0 + F_1 + \cdots + F_n}$$
$$= \frac{F_0 + F_1 + \cdots + F_{n-1}}{F_0 + F_1 + \cdots + F_n}$$

We now have a means of calculating the blending coefficients from a set of filter kernel coefficients.

Abraham Mammen is a member of the graphics hardware group at Stellar Computer, where he is responsible for algorithm and microcode development for the rendering system on the Stellar GS1000. Before joining Stellar in 1986, he worked at Computervision in high-end geometry and rendering accelerators, and earlier at General Instrument Corporation as a systems architect responsible for microprocessor development for digital signal processing and visual entertainment applications. His interests focus on graphics architectures and algorithms for scientific and engineering visualization applications.

Mammen received his BE in electrical engineering in 1978 from Birla Institute of Technology and Science, Pilani, India, and his MS in electrical engineering and computer science in 1980 from the State University of New York at Stony Brook. He is a member of ACM and IEEE.

Mammen can be contacted at Stellar Computer, 85 Wells Ave., Newton, MA 02159.