The ZZ-Buffer:
A Simple and Efficient Rendering Algorithm with Reliable Antialiasing

David Salesin*
Stanford University

Jorge Stolfi†
DEC Systems Research Center

Abstract

The ZZ-buffer is a new rendering algorithm that is simple, efficient, and produces high-quality images. The algorithm correctly renders transparent surfaces, shadows with real penumbrae, and depth of field effects. The ZZ-buffer algorithm is substantially faster than ray tracing and nearly as versatile. While the ZZ-buffer is somewhat slower than the Z-buffer or A-buffer, it avoids the aliasing and other artifacts of these algorithms. The algorithm's efficiency comes from a screen-space object indexing scheme, and from the use of lazy evaluation for visibility tests. It achieves reliable antialiasing by employing an adaptive form of stochastic supersampling. The algorithm is simple enough that we give most of its code here in this paper. The algorithm has been implemented as part of a commercial production system and has proved robust over a large variety of images.

*Address: Stanford University, Computer Science Dept., Stanford, CA 94305, phone (415) 723-4102. Part of the work was done while this author was consulting at Sogitec Audiovisuel in Paris.

†Address: DEC Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, phone (415) 853-2226.

1 Introduction

Existing rendering algorithms offer only the unpleasant choice between low quality and high computational cost. One must either use an algorithm that makes unreliable approximations, such as the Z-buffer [5, 12] or A-buffer [4], and put up with aliasing and other artifacts, or fall back on a general-purpose ray tracing algorithm [7, 16, 30], and be prepared to pay the price.

We present here the ZZ-buffer algorithm, a new rendering method that avoids aliasing and most other objectionable artifacts, and that is considerably more efficient than ray tracing. In addition to providing reliable antialiasing, the ZZ-buffer algorithm can handle transparent surfaces, shadows with real penumbrae, and depth of field effects. It can also handle a wide variety of geometric primitives directly, without converting them first to polygons or some other intermediate form.

The ZZ-buffer algorithm superficially resembles the A-buffer and Z-buffer algorithms in that it uses an image-space buffer to aid in determining visibility. However, the ZZ-buffer algorithm avoids the aliasing and quantization artifacts that are inherent in the Z-buffer by using stochastic sampling [6, 9, 18, 19] and performing precise visibility tests at each sample point. Note that such artifacts can also appear with an A-buffer whenever the scene contains features smaller than the resolution of the A-buffer's pixel mask [4] or whenever surfaces become too close within a pixel.

The ZZ-buffer improves on the performance of ray tracing by three major strategies. First, it uses an efficient indexing scheme, similar to the "item buffer" of Weghorst *et. al.* [29], for determining the objects intersected by primary rays and by rays to the light sources. Second, it uses the same indexing scheme to detect those pixels that contain no small features and that can therefore be sampled by a single ray. Finally, it uses rough approximations to the depths of objects at each pixel in order to eliminate invisible objects without ever computing their exact depths. (The ZZ-buffer does not optimize the tracing of reflected or refracted rays, although these features can be incorporated into the ZZ-buffer algorithm at the usual cost.)

Although our algorithm does more work per pixel than the A-buffer to determine visibility, the ZZ-buffer only shades surfaces that are actually visible, whereas the A-buffer must shade them all.

The paper is organized as follows. Section 2 introduces the ZZ-buffer in its general form. Section 3 shows how the ZZ-buffer is used in a rendering algorithm. Section 4 gives a more detailed analysis of the ZZ-buffer algorithm's capabilities and running time and compares it to the A-buffer and to some typical acceleration schemes for general ray tracing. Finally, Section 5 comments on our implementations of the ZZ-buffer, and Section 6 proposes some areas for further research.

2 The ZZ-buffer

The ZZ-buffer is a two-dimensional array $ZZ[i, j]$, each of whose entries is associated with a rectangular *cell* in the screen plane. Each cell covers a fixed-sized rectangular block of pixels (or possibly a single pixel) of the final image. See figure 1.

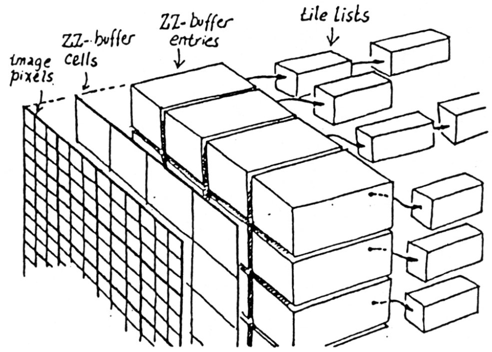

Figure 1. The ZZ-buffer.

Each entry of the ZZ-buffer contains the following fields:

```
type Entry = record
  tilelist:      pointer to TileList
  zmin, zmax:    Coordinate
  opaque:        boolean
end record
```

The *tilelist* field points to a linked list of objects that may be visible within the ZZ-buffer cell. The *opaque* flag tells whether this list contains at least one object that is opaque and covers the cell entirely.

The *zmin* and *zmax* fields (from which the ZZ-buffer derives its name) provide an estimate of the range of z-coordinates (depths) spanned by the visible objects in the cell. The range $[zmin, zmax]$ stored in the ZZ-buffer need not be accurate, but must include the z-coordinates of every visible point of the model within the cell. See figure 2.

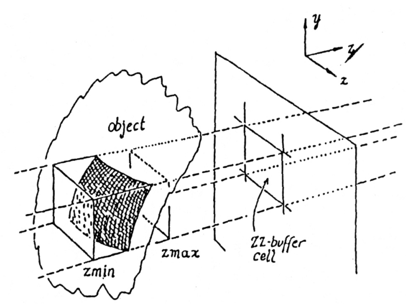

Figure 2. Tiles.

Each element of the *tilelist* is called a *tile* and describes an object implicitly clipped to a ZZ-buffer cell. The contents of a tile are similar to those of a ZZ-buffer entry:

```
type Tile = record
  obj:           pointer to Object
  zmin, zmax:    Coordinate
  opaque:        boolean
end record
```

```
type TileList = record
  first:   Tile
  rest:    pointer to TileList
end record
```

However, in a *Tile* record, the *opaque*, *zmin*, and *zmax* fields refer to properties of a single object (restricted to the cell in question), whereas the fields of the ZZ-buffer entry summarize the properties of the entire list of objects.

Each ZZ-buffer entry is constructed by repeated calls to the *AddTile* procedure below. The procedure performs a quick visibility test, based on the *zmin*, *zmax*, and *opaque* attributes of the new tile and of the ZZ-buffer entry. The new tile is thrown away if it is completely obscured by those in the list. Conversely, the tiles already in the list are thrown away if they are completely obscured by the new one. If neither of these two conditions holds, then the new object is appended to the list of potentially visible objects, and the fields *zmin*, *zmax*, and *opaque* of the ZZ-buffer entry are updated.

```
procedure AddTile takes
  e:    var Entry
  new:  Tile
begin
  if not e.opaque or new.zmin ≤ e.zmax then
    { The new object may be visible: }
    if opaque and e.zmin > new.zmax then
      { The new object blocks all the old ones: }
      reclaim(e.tilelist)
      e.tilelist ← alloc TileList[new, nil]
      e.zmin ← new.zmin
      e.zmax ← new.zmax
      e.opaque ← new.opaque
    else
      { Add object to list and update entry: }
      e.tilelist ← alloc TileList[new, e.tilelist]
      e.zmin ← min(e.zmin, new.zmin)
      if e.opaque and new.opaque then
        e.zmax ← min(e.zmax, new.zmax)
      elseif not e.opaque and new.opaque then
        e.zmax ← new.zmax
      elseif not e.opaque then
        e.zmax ← max(e.zmax, new.zmax)
      endif
      e.opaque ← e.opaque or new.opaque
    endif
  endif
end procedure
```

Note that this operation takes only a constant amount of time per object and per cell, because whenever the z interval of the new tile overlaps that of the entire list, we just append the new tile to the list. We do not attempt to throw away the individual tiles that are obscured by the new tile because the time spent in this search is likely to be wasted when a subsequent tile obscures them all.

3 ZZ-buffer rendering

The ZZ-buffer rendering algorithm can be divided into two phases. In the first phase, called *scan conversion*, we take each object, compute a rough $[zmin, zmax]$ interval for every cell that may contain some part of the object, and store the resulting tiles into the ZZ-buffer using the *AddTile* operation. In the second phase, called *rendering*, the visibility of objects within each pixel is further refined (if it has not already been resolved by the ZZ-buffer), visible objects are shaded, and the shades are blended to produce the final pixel colors.

In practice, memory limitations may dictate that these two phases be interleaved on a scanline-by-scanline basis. This approach entails transforming all geometric primitives to screen coordinates at the outset and sorting them in y, but then allows us to keep in memory only those objects that intersect the current scanline.

In the algorithms below, we assume that each object O is represented internally by some record $O.data$, containing geometric parameters (already transformed to screen coordinates) and other data about the object, along with a suite of procedures $O.methods$ that implement the basic rendering and visibility functions that our algorithm requires. We will write $O.M(x, \ldots)$ as a shorthand for $O.methods.M(O.data, x, \ldots)$.

3.1 Scan conversion

In the scan-conversion phase, we estimate the set of cells that each object covers, either totally or partially. We also have to estimate the range of screen depths $[zmin, zmax]$ of the visible parts of the object within each cell, and whether the object is "opaque," that is, whether it is made of an opaque material and completely covers the cell.

Both the $[zmin, zmax]$ interval and the *opaque* flag can be rough approximations to the truth, as long as they are conservative. Thus, the $[zmin, zmax]$ interval must completely contain the actual z depths of the visible portions of the object, but need not represent the smallest such interval. Similarly, the *opaque* flag should be `true` only if the object is indeed opaque and guaranteed to completely cover the cell, but it may always be set to `false` in case of doubt.

The implementor of the scan-conversion process, therefore, has great latitude in performing these estimates. At one extreme, the programmer could simply call *AddTile* for every cell within the xy-bounding box of the object, with $opaque =$ `false` and $[zmin, zmax]$ equal to the object's overall z-extent. This "sloppy" approach would be perfectly correct. However, it will generally result in longer tile lists and make the rendering phase more expensive.

At the other extreme, the programmer might code a very precise scan-conversion procedure that computes the tightest z range possible for each tile, and sets the *opaque* flag to `true` every time the time the object is opaque and completely covers the cell. However, this approach may also be inefficient since it means wasting a lot of computation on objects that may become obscured later on, and since in many cases visibility can be determined by a much coarser estimate.

The best approach to writing scan-conversion routines is to strike a balance between these two extremes. To illustrate these tradeoffs, we present scan-conversion routines for two popular geometric primitives: polygons and quadric surfaces.

Polygons

We classify polygons into two categories: small and large. We consider a polygon small if its *xy*-bounding box does not span more than a few cells (say, 5 or 6) in each dimension. For a polygon this small, there is little difference between the cells touched by the polygon and by its bounding box, so we can use the sloppy approach described above.

If the polygon is larger than a few cells, we can treat it with an ordinary scan-conversion algorithm [22], except for one small change: as the scan conversion proceeds, we keep track of two positions along each active edge instead of just one. See figure 3. For each edge of the polygon, imagine two "bugs" that trace Bresenham paths along the grid lines bracketing that edge. These paths divide the cells into three main sets, characterized by how much each cell is covered by the polygon: the *outside cells*, disjoint from the polygon; the *inside cells*, completely covered by it; and the *boundary cells*, covered only in part.

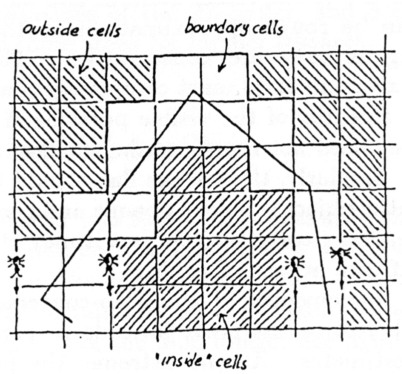

Figure 3. Scan-converting a polygon.

We update only the ZZ-buffer entries corresponding to inside or boundary cells. For boundary cells, the object inserted into the ZZ-buffer is the polygon itself, with *opaque* = `false`. For interior cells, we store instead a pointer to a simplified object that consists only of the polygon's supporting plane, without its edges. This substitution is made so that the rendering phase will not look at the polygon's edges if it needs to test whether a sample ray within this cell intersects the polygon. The *opaque* flag is `true` or `false` depending on whether the polygon is made of an opaque or transparent material. The *zmin* and *zmax* along each scanline can be obtained incrementally by the Bresenham algorithm.

If a polygon is large and also has many edges, it may be worth extending the optimization above even further, by remembering for each cell touched by the polygon the subset of edges intersecting the current scanline. These edges are all we need for testing whether a ray passing through the cell intersects the polygon.

Quadric surfaces

For quadric surfaces, such as ellipsoids, paraboloids, cylinders, and cones, we do scan conversion as follows. First, we map the quadric to screen space, so that its equation has the form

$$ax^2+bxy+cxz+dy^2+eyz+fz^2+gx+hy+iz+j=0,$$

where x, y, and z are screen coordinates. Now consider a particular ZZ-buffer cell that corresponds to the region $[x-\varepsilon, x+\varepsilon] \times [y-\varepsilon, y+\varepsilon]$ of the screen plane. A ray from the viewpoint through this cell consists of the points $(x+\delta_x, y+\delta_y, z)$ for all z. Substituting this expression into the formula above and collecting powers of z gives the quadratic equation $Az^2+Bz+C=0$, where

$$\begin{aligned}
A &= f \\
B &= cx+ey+i+c\delta_x+e\delta_y \\
C &= ax^2+bxy+dy^2+gx+hy+j \\
&\quad + (2ax+by+g)\delta_x+(bx+2dy+h)\delta_y \\
&\quad + a\delta_x^2+d\delta_y^2.
\end{aligned}$$

We then compute intervals $\bar{A}$, $\bar{B}$, and $\bar{C}$ that are guaranteed to contain the possible values of A, B, and C for this particular cell:

$$\begin{aligned}
\bar{A} &= [f,f] \\
\bar{B} &= cx+ey+i \pm \varepsilon * (|c|+|e|) \\
\bar{C} &= ax^2+bxy+dy^2+gx+hy+j \\
&\quad \pm \varepsilon * (|2ax+by+g|+|bx+2dy+h|) \\
&\quad + a*[0,\varepsilon^2]+b*[0,\varepsilon^2]
\end{aligned}$$

In these equations, $u \pm v$ denotes the interval $[u-v, u+v]$, and the intervals are added as in interval arithmetic [21].

The result is a "fuzzy" second-degree equation $\bar{A}z^2+\bar{B}z+\bar{C}=0$ whose coefficients are intervals. We solve this equation using standard interval arithmetic methods, obtaining a pair of intervals $[z0.lo, z0.hi]$ and $[z1.lo, z1.hi]$ that are guaranteed to contain the z-coordinates of the first and second intersections of any ray with the quadric surface within this cell. We then store into the ZZ-buffer either the first interval $[z0.lo, z0.hi]$, or else the union of the two intervals, depending on whether the surface is opaque or transparent.

3.2 Rendering

Once all the objects intersecting a given scanline have been placed into the ZZ-buffer by the scan-conversion phase, the rendering phase takes over. We begin this phase by sorting the tile list in each cell of the ZZ-buffer in order of increasing *zmin*. (At this point, we can throw away any tile whose *zmin* is greater than the cell's *zmax*.)

Once the tile lists are sorted, we render each pixel, using either a single sampling ray or a bundle of $S \times S$ jittered sampling rays, depending on the complexity of the scene inside the pixel. We determine visibility by using first the z ranges and *opaque* flags stored in the tiles, and computing exact z values only when this information is not enough. Finally, we perform shading only for those surfaces that are found to be visible.

Top-level rendering routines

In the rendering phase, the top-level routine *RenderPixel* is called once for each pixel; its task is to compute the pixel's color based on the tile list of the cell covering the pixel. The routine first attempts to compute an overall shade for the entire pixel by going through the tile list from front to back and evalutating the shade of each tile at the pixel center. This process continues until an opaque tile is reached, or until the situation becomes too complex to be rendered accurately with a single sample. In the latter case, the routine resorts to jittered supersampling: it shoots $S \times S$ rays within the pixel, for some constant S, and tests each ray against the remainder of the list.

In the following pseudocode, (xc, yc) are the coordinates of the center of the pixel being rendered, and a *Tint* is a record t consisting of a color triple $t.color = [r, g, b]$ and the opacity coefficients for each channel $t.alpha = [\alpha_r, \alpha_g, \alpha_b]$ [23].

```
procedure RenderPixel takes
  tilelist: pointer to TileList
  xc, yc:   Coordinate
begin
  (tint, tilelist) ← RenderArea(tilelist, xc, yc)
  if tilelist = nil then
    PutPixel(xc, yc, tint)
  else
    fgd ← tint;
    for i ← 1 to S do
      for j ← 1 to S do
        (xs, ys) ← Jitter(xc, yc, i, j)
        tint ← RenderPoint(tilelist, xs, ys, fgd)
        PutSample(xs, ys, tint)
      end for
    end for
  endif
end procedure
```

The subroutine *PutSample* simply stores the given tint into an array of sampled tints, with every block of $S \times S$ elements in the array corresponding to a single pixel. The subroutine *PutPixel* replicates the given tint into all $S \times S$ sample positions covered by the pixel. The array of samples will ultimately be filtered to produce the final image, as described in Section 3.4.

The function $Jitter(xc, yc, i, j)$ considers the pixel centered at (xc, yc) to be an $S \times S$ array of subpixels, and it computes a uniformly distributed random point within subpixel $[i, j]$.

The subroutine *RenderArea* is the part of *RenderPixel* that takes care of tiles that can be rendered with a single sample per pixel. It repeatedly extracts the nearest tile from the tile list, renders it, and blends the resulting color and opacity with those of the previously rendered tiles. If the routine cannot determine which is the closest tile (because of overlapping z ranges), or if the closest tile is too complex to be rendered with a single sample, then the routine gives up, returning the tint that it has computed so far, along with the list of tiles that remain to be processed.

The test to see whether the tile can be shaded with a single sample is handled by the method *TooComplex*, associated with the tile's object. This method must return `true` if the object does not cover the pixel entirely, or if the object's shade varies too much within the pixel—e.g., because the object is textured or contains a highlight within the pixel. In case of doubt, it is always safer to return `true` and force supersampling.

```
procedure RenderArea takes
  tilelist: pointer to TileList
  xc, yc:   Coordinate
returns
  tint:       Tint
  remainder:  pointer to TileList
begin
  tint.color ← [0, 0, 0]
  tint.alpha ← [0, 0, 0]
  l ← tilelist
  while l ≠ nil
  and not Saturated(tint.alpha)
  and (l.rest = nil or l.zmax < l.rest.first.zmin)
  and not l.first.obj.TooComplex(xc, yc) do
    zc ← l.first.obj.ComputeZs(xc, yc);
    objtint ← ShadePoint(l.first.obj, xc, yc, zc)
    tint ← ComposeTints(tint, objtint)
    l ← l.rest
  end while
  return (tint, l)
end procedure
```

The *ComposeTints* function combines the colors and transparencies of a foreground object f and a background object b into a single tint t, according to the standard compositing formulas [23]:

$$t.color.\gamma = f.color.\gamma + (1 - f.alpha.\gamma) * b.color.\gamma$$
$$t.alpha.\gamma = f.alpha.\gamma + (1 - f.alpha.\gamma) * b.alpha.\gamma,$$

where γ is either r, g, or b.

Once *RenderPixel* decides to use jittered supersampling, it calls the subroutine *RenderPoint* for each sample ray. Note that if the first few objects in the ZZ-buffer tile list are transparent, they may have already been shaded by *RenderArea* before *RenderPoint* is called. *RenderPoint* therefore takes as a parameter the color of the preceding layers, along with the list of tiles not yet rendered, and the jittered sample position.

The *RenderPoint* routine works by computing shades for the objects that intersect the ray at (xs, ys) in increasing z order until the *alpha* for the sample is saturated. *RenderPoint* calls *NextHit* to enumerate intersections in correct z order. Note that this task is not trivial, since the tiles are sorted in order of increasing *zmin*, which may not correspond to the order of the objects' actual z coordinate at (xs, ys). Note also that each tile may represent more than a single intersection: for example, a tile for a transparent sphere represents two intersections, one for the front surface and one for the back.

The *RenderPoint* routine makes calls to a routine *ShadePoint* that returns the object's shade at a sample point. The *ShadePoint* routine does not have to do any color averaging or antialiasing itself, since (as observed by Cook, Porter, and Carpenter [7]) the stochastic sampling that is already being performed by *RenderPixel* will avoid aliasing in the shading as well.

```
procedure RenderPoint takes
  tilelist:     pointer to TileList
  xs, ys:       Coordinate
  foreground:   Tint
returns
  tint:  Tint
begin
  tint ← foreground
  hits ← nil
  tiles ← tilelist
  (obj, z, tiles) ← NextHit(tiles, hits, xs, ys)
  while obj ≠ nil
  and not Saturated(tint.alpha) do
    objtint ← ShadePoint(obj, xs, ys, z)
    tint ← ComposeTints(tint, objtint)
    (obj, z) ← NexHit(hitqueue)
  end while
  return tint
end procedure
```

Visibility Testing

When testing for visibility, instead of computing all ray-object intersections and sorting them at the outset, we determine each subsequent intersection only as it is needed. In most cases, we will only need to compute the first intersection.

The *NextHit* routine implements this idea. It scans the given tile list and tests each tile against the ray through (xs, ys) using the *HitTest* object method. Whenever *NextHit* finds a tile that intersects the ray, it calls upon the *ComputeZs* method to get the exact z values of the intersection points, and merges those intersections into the sorted list *hits*. This process continues until *NextHit* is sure it has located the nearest intersection in the list—that is, until the *zmin* of the next tile lies beyond the smallest z in *hits*. At this point, *NextHit* suspends its search, removes the first element of *hits*, and returns it as the next closest hit, along with the list of tiles not yet examined.

```
type HitList = record
  obj:   pointer to Object
  z:     Coordinate
  rest:  pointer to HitList
end record

procedure NextHit takes
  tilelist:  pointer to Tilelist
  hits:      pointer to HitList
  xs, ys:    Coordinate
returns
  obj:       Object
  z:         Coordinate
  tilelist:  pointer to Tilelist
begin
  zbest ← +∞
  if hits ≠ nil then zbest ← hits.z endif
  while tiles ≠ nil and zbest > tiles.first.zmin do
    tile ← tiles.first; tiles ← tiles.rest
    if tile.opaq or tile.obj.HitTest(xs, ys) then
      zlist ← tile.obj.ComputeZs(xs, ys)
      for each z in zlist do
        if z < zbest then zbest ← z endif
        InsertHits(tile.obj,z,hits)
      end for
    endif
  end while

  if hits = nil then
    return (nil, +∞)
  else
    obj ← hits.obj
    z ← hits.z
    hits ← hits.rest
    return (obj, z)
  endif
end procedure
```

Shading

Once we have determined a surface that is visible along a sample ray, we compute its color and transparency at the point of intersection p by calling the routine *ShadePoint*. This routine calls the object method *SurfaceProperties* to determine the surface's material and normal vector at the point p. Then, for each light source, *ShadePoint* calls a subroutine *Illuminate* to determine the amount of light reaching p from that source. It then computes the effect of the light on the surface, using the method *GetColor* associated with the surface's material.

```
procedure ShadePoint takes
  obj:  pointer to ProjectedObject
  ps:   ScreenCoordinates
returns
  tint: Tint
begin
  p ← Map(xs, ys, zs, camera.screen-to-world)
  (norm, material) ← obj.SurfaceProperties(p)
  viewdir ← Normalize(camera.viewpoint − p)
  tint.alpha ← material.GetAlpha(norm, viewdir)
  tint.color ← [0, 0, 0]
  for each source in source-list do
    (lightcolor, lightdir) ← Illuminate(source, p)
    objcolor ← material.GetColor
      (lightcolor, lightdir, norm, viewdir)
    tint.color ← tint.color + objcolor
  end for
  return tint
end procedure
```

Shadows

The ZZ-buffer can also be used to compute shadows, in a manner similar to the "light buffer" of Haines and Greenberg [14]. We will first consider using the ZZ-buffer to cast shadows from a point source—i.e., casting shadows without penumbrae. We will then see how the algorithm can be enhanced to compute accurate penumbrae in the following section.

To implement shadows, we modify the ZZ-buffer algorithm in two ways. During scan conversion, we create an additional ZZ-buffer for each light source, and we scan convert every object into each of these ZZ-buffers from the point of view of the corresponding light. This scan conversion is identical to the ordinary scan conversion already described. The resolution of the light ZZ-buffers are independent of each other and of the resolution of the camera ZZ-buffer (and are often considerably smaller).

During rendering, we determine visibility as usual, but when computing the illumination from each light source we use the light's ZZ-buffer to detect any obscuring objects. To do this, we transform the visible surface point p that is being shaded to the screen coordinate system corresponding to the light's point of view. We then extract the list of tiles from the corresponding cell of the light's ZZ-buffer and check each tile's object against the ray from the light source to the point p, using the object's *HitTest* and *ComputeZs* methods. If we find an opaque object that intersects the ray, then p is in shadow with respect to the light. If we find transparent objects, then the light is filtered by the objects' transmission coefficients.

In order to compute which objects are in front of the point being shaded, it is not necessary to sort the obscuring objects by their distance from the light, since their combined transmission coefficients are independent of this order. Instead, we can just go through the list of tiles, multiplying the transmission coefficients of every object that intersects the ray, until we find a tile whose *zmin* is greater than the z of the point being shaded.

Penumbrae

Penumbrae (soft shadow edges) are among the most important ingredients of realistic image synthesis, as illustrated by figure 4.

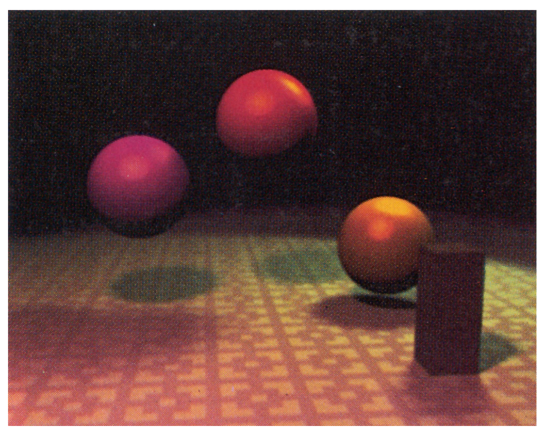

Figure 4. A scene with real penumbrae.

To produce penumbrae, we imagine that each light source is a disk of some fixed size, oriented parallel to the projection plane of its ZZ-buffer. See figure 5. The visibility computation is performed as before, but shadowing is done, as in distributed ray tracing [7], by casting a *perturbed ray* from a jittered point on this disk to the point being shaded. In screen coordinates, these rays will be lines of the form $(x_p + m_x z, y_p + m_y z, z, 1)$, where (x_p, y_p) is the point where the ray hits the screen plane, and m_x, m_y are numbers that describe the deviation of the ray from the z-axis. See figure 5. Note that in the light's screen coordinates, the light source itself is a disk at infinity centered at the point $(0,0,1,0)$, so $\sqrt{m_x^2+m_y^2}<\tan\theta$, where θ is the angular radius of the disk. Note that this bound on m_x and m_y is independent of the screen coordinates x_p and y_p.

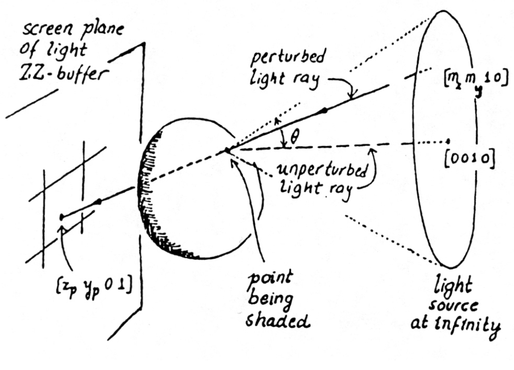

Figure 5. Ray tracing penumbrae.

In order to find the objects that might be intersected by this perturbed ray, we have to modify the scan conversion somewhat. We place into the ZZ-buffer list for each cell every object that is intersected by the double truncated "pyramid" shown in figure 6. This pyramid is the union of all perturbed rays that hit the ZZ-buffer cell.

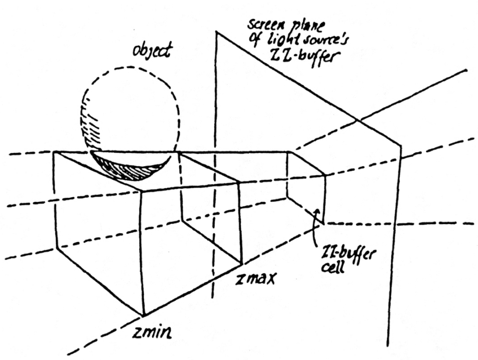

Figure 6. Scan converting for penumbrae.

Ideally, we would like to determine exactly which pyramids intersect the primitive, compute tight bounds for the z ranges of such intersections, and determine which cells have their pyramids completely blocked by the primitive (which would allow us to set the corresponding *opaque* flags to `true`). Unfortunately, these computation are difficult, except perhaps for very simple primitives such as planes or quadrics. A practical alternative is to scan convert an enlarged three-dimensional bounding box of the object in screen space. To compute the enlarged bounding box, note that a perturbed ray intersects an object if and only if the corresponding unperturbed ray (with the same screen intercept (x_p, y_p)) intersects a copy of the object sheared by the map $(x,y,z) \to (x-m_xz, y-m_yz, z)$. Therefore, if the bounding box is of the form $R \times [zmin, zmax]$, we insert the object into every cell of the ZZ-buffer that overlaps the enlarged rectangle R' obtained by displacing the edges of R outward by the distance $(\tan\theta)\max\{|zmin|, |zmax|\}$.

Note that we must set the *opaque* flag to `false` in every cell covered by R'. Although in this way we lose the ability to remove covered objects from the ZZ-buffer in the *AddTile* routine, the ZZ-buffer still provides an indexing capability, and its z ranges still optimize the visibility computation.

The following code implements the *Illuminate* subroutine for shadows with penumbrae:

```
procedure Illuminate takes
  source: LightSource
  pt:     WorldCoordinates
returns
  lightcolor: Color
  lightdir:   Direction
begin
  (x_s, y_s, z_s) ← Map(pt, source.world-to-screen)
  (m_x, m_y) ← LightJitter(source.size)
  lightpos ← (m_x, m_y, 1, 0)
  lightpos ← Map(lightpos, source.screen-to-world)
  if source.at-infinity then
    distsq ← 1
  else
    distsq ← |lightpos − point|^2
  endif
  lightdir ← Direction(lightpos − pt)
  (x_p, y_p) ← (x_s − m_x * z_s, y_s − m_y * z_s)
  list ← source.zzbuffer[x_p, y_p]
  atten ← source.BeamIntensity(light.dir)/distsq
  lightcolor ← atten * source.color
  while list ≠ nil
  and not TooDark(lightcolor) do
    tile ← list.first; obj ← tile.obj
    if tile.zmin < z_s − ε
    and (tile.opaque or obj.HitTest(x_p, y_p, m_x, m_y))
    then
      zlist ← obj.ComputeZs(x_p, y_p, m_x, m_y)
      for each z'_s in zlist do
        if z'_s < z_s − ε then
          pt' ← (x_p + m_x * z'_s, y_p + m_y * z'_s.z'_s)
          pt' ← Map(pt', source.screen-to-world)
          alpha ← obj.GetAlpha(pt', lightdir)
          lightcolor ← lightcolor * ((1,1,1) − alpha)
        endif
      end for
    endif
  end while
end procedure
```

Note that this code requires slightly generalized versions of *HitTest* and *ComputeZs* that take a perturbed ray (with slope (m_x, m_y)) instead of a ray parallel to the screen z-axis. If penumbrae are not required, then *source.size* is 0, m_x and m_y are 0, $x_p = x_s$, $y_p = y_s$, and the code can be simplified considerably.

3.3 Depth of field effects

The same mechanism that we use to produce penumbrae can be used to create images that simulate the effects of limited depth of field, as shown in figure 7. When rendering each sample point, we generate a random point on the camera's "lens" (which is a disk on the camera plane centered at the nominal viewpoint) and fire a ray from this jittered viewpoint toward the sample point on the screen plane. In order to locate all objects that might intersect this perturbed ray, we must scan-convert objects into the camera's ZZ-buffer by the same mechanism that we used for extended light sources.

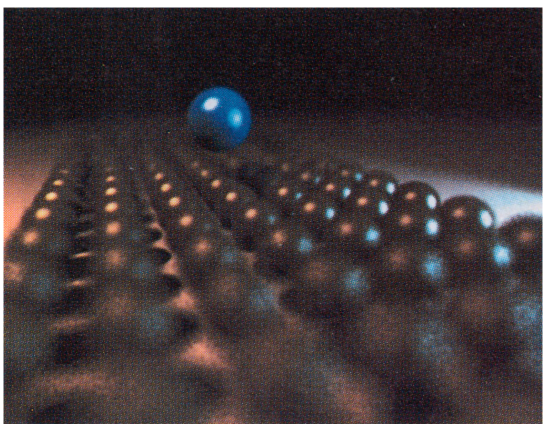

Figure 7. Limited depth of field.

3.4 Filtering

Recall that the output of the rendering phase is an array of samples, with $S \times S$ samples for each pixel of the final image. After each scanline of pixels is produced, we take the corresponding S rows of the sample array and compress them into a single row of pixel values, using standard convolution filtering [20, 24]. In this filtering, each sample contributes to a 3×3 square of pixels in the final image, with weights that depend on the distance between the unjittered sample position and the centers of those pixels.

4 Comparison to other methods

We now compare the ZZ-buffer algorithm against several other rendering algorithms that resemble it in capability or speed. In particular, we will consider the classical Z-buffer and A-buffer algorithms, and some variants of ray tracing.

4.1 Capabilities

The ZZ-buffer algorithm approaches the versatility of distributed ray tracing in that it can render a wide variety of objects and produce many of the same effects, such as shadows and penumbrae, depth of field, and transparent surfaces. The ZZ-buffer algorithm is also free of many of the approximation errors that plague the Z-buffer and A-buffer algorithms, and that can result in aliasing and other image artifacts; the images produced by the ZZ-buffer, in fact, are as good as those of distributed ray tracing.

In the rest of this section, we will discuss these points in more detail.

Antialiasing

The stochastic supersampling that the ZZ-buffer employs replaces aliasing by noise of the correct average intensity. Although this noise is somewhat objectionable, it arises in cases where the Z-buffer or A-buffer would produce much more objectionable aliasing effects, such as jaggies or Moiré patterns.

The Z-buffer has the problem that it cannot keep track of more than a single object per pixel, so it cannot accurately render small features that may be present in the model, or that may arise as the result of occlusions among larger surfaces. Indeed, users of the Z-buffer algorithm typically generate images at much higher resolution and then filter them down to final size. The A-buffer can be seen as a modification of a supersampling Z-buffer in which depth and color information are kept at the original image resolution, while coverage information is represented at the higher resolution.

Both the supersampling Z-buffer and the A-buffer still alias if the feature size is sufficiently small. Figure 8 shows an example, due to R. Cook [6], in which the A-buffer algorithm produces severe aliasing. The subject is a "comb" with 200 triangular teeth, each tooth 100 pixels high and 1.01 pixels wide at the base. The upper left and right images show the comb rendered with 4 and 36 samples per pixel, using uniform sampling, while the lower two images show the comb rendered with equal numbers of samples, and jittered sampling. Note that the A-buffer is typically run with 32 samples per pixel, while the ZZ-buffer produces perfectly acceptable results with just 4.

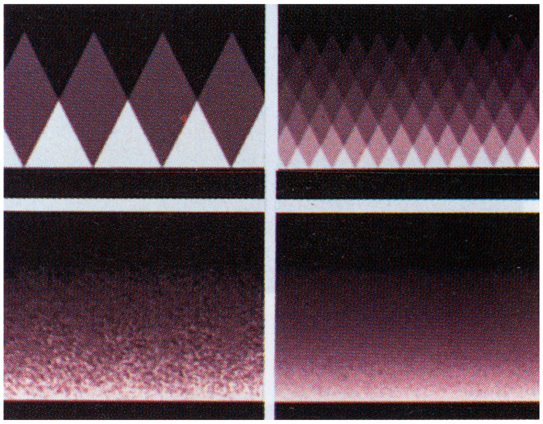

Figure 8. Aliasing: A-buffer vs. ZZ-buffer.

Figure 9 shows an infinite checkerboard in perspective, rendered with 4 and 36 samples per pixel, both unjittered (above) and jittered (below). Notice how jittered sampling blends the small detail to give a slightly grainy color of the correct average intensity.

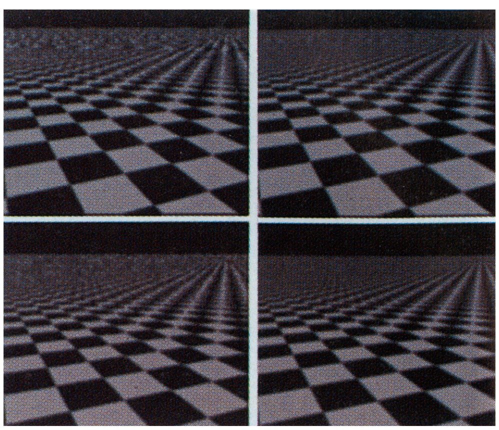

Figure 9. Aliasing on a checkerboard.

Correct visibility

Since the A-buffer represents depth and color information only at pixel resolution, it must use approximate formulas to blend the colors of two surfaces that intersect within a pixel. These approximations can easily go astray. Problems arise for example when too many surfaces intersect within the same pixel. Moreover, when two surfaces are close but non-intersecting, the A-buffer is unable to tell that one surface is completely in front of the other and instead blends their colors as if they were intersecting.

The ZZ-buffer avoids this problem since it stores pointers back to original objects and always determines exact visibility for each sample point. (Note that the approximate z ranges computed in the scan conversion phase are used only for preliminary culling of invisible surfaces.)

Figure 10 shows two spheres of radius 50, whose centers differ by only 0.001 in the x and y directions. Note that the ZZ-buffer correctly determines the relative visibility of the two spheres, even though their z ranges overlap in every pixel. By contrast, the A-buffer algorithm would blend the colors of the two spheres, producing shades of yellow throughout.

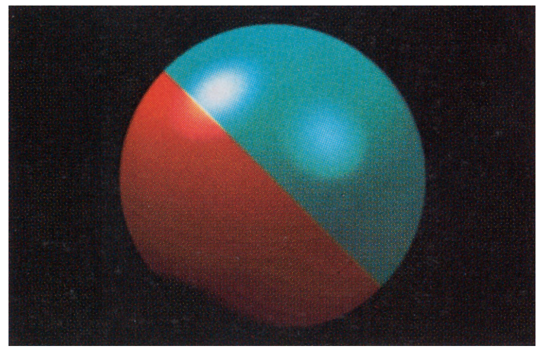

Figure 10. Two intersecting spheres.

Transparent surfaces

A Z-buffer algorithm can be modified to handle transparent surfaces by sorting any transparent surfaces back to front and painting them into the Z-buffer after all opaque surfaces have been processed. This trick cannot be used if sorting is impossible, such as when any two transparent surfaces intersect or any three are cyclically ordered. The ZZ-buffer, on the other hand, handles all these cases easily, since it does not require any sorting. The A-buffer also has problems with transparent surfaces, since the same kinds of situations that causes visibility problems (such as surfaces too close to each other) will also cause incorrect blending of the colors of transparent surfaces.

General primitives

Another advantage of the ZZ-buffer over the A-buffer is that the ZZ-buffer algorithm does not require all objects to be converted into polygons. The ZZ-buffer algorithm can directly render quadric surfaces, patches, surfaces defined by implicit equations [3], and many other types of geometric primitives.

Shadows

As we have seen, the ZZ-buffer can also be used to render shadows. The ZZ-buffer scheme has many advantages over shadowing methods based on ordinary Z-buffers, such as William's algorithm [31] or "percentage-closer filtering" [25]. For one thing, the ZZ-buffer allows us to correctly antialias sharp shadow boundaries and shadows cast by small objects and fine textures. In algorithms that use ordinary Z-buffers for the light's depth maps, any artifacts on the outlines of objects stored in the light's Z-buffer may become magnified many times when they are projected as shadow edges in the final image. Even if we use stochastic sampling to compute the light's Z-buffer, the resulting noise can still be magnified until it becomes objectionable. This problem does not arise with the ZZ-buffer since sampling is only done from the camera's viewpoint, which prevents the noise from being magnified, and since the illumination at each sample is computed exactly.

A consequence of these precise illumination tests is that in the ZZ-buffer scheme, perfect results are obtained regardless of the resolution of the light source's ZZ-buffer. The ZZ-buffer resolution, in fact, only affects the algorithm's speed, not its accuracy. This property is not true of other shadow algorithms based on ordinary Z-buffers. These algorithms require depth maps of resolution comparable to that of the final image—not necessarily to resolve any true high frequencies in the scene, but simply to prevent noise or aliasing artifacts at the edges of shadowing objects from becoming noticeable. For simple scenes with few overlapping surfaces, then, a very small ZZ-buffer is a big win, both in terms of memory and speed, over the large depth maps that would be required by a Z-buffer algorithm to produce a comparable result.

As we have already pointed out, the ZZ-buffer shadow algorithm is somewhat similar to the "light buffer" method of Haines and Greenberg [14]. However, it has the advantage of keeping both a *zmin* and a *zmax* for each tile, which allow it to prune the tile lists correctly.

Penumbrae

As we have shown, the ZZ-buffer algorithm can be used to optimize the computation of correct penumbrae by shooting off-axis rays toward the light sources. This capability is in fact another consequence of storing pointers to the original objects in the ZZ-buffer. Note that with percentage-closer filtering, the best that can be done is to simulate a penumbrae effect by a trick that does not take the actual geometry into consideration, producing penumbrae with incorrect sizes and positions.

Note also that the ZZ-buffer shadow algorithm correctly handles colored transparent surfaces that filter light, an effect that cannot be reproduced by percentage-closer filtering because its depth maps lose all information about the properties of the surfaces occluding the light.

Depth of field effects

The same mechanism that we use to produce penumbrae can be used to create images that simulate the effects of limited depth of field, as in figure 7.

4.2 Ease of programming

Considering the capabilities of the ZZ-buffer algorithm, it is surprisingly easy to program. Because the ZZ-buffer simply uses stochastic supersampling whenever the situation in a pixel is complex, it avoids many of the complications arising in the A-buffer algorithm. For example, the ZZ-buffer algorithm does not need to compute or store pixel masks, clip polygons to pixel boundaries, or estimate visibilities when two or more surfaces intersect within the same pixel.

As we noted before, the ZZ-buffer algorithm can directly render many kinds of primitives without converting them to an intermediate form. For certain kinds of geometric primitives, such as algebraically defined surfaces, this feature may simplify programming considerably, since the conversion of such primitives to polygons may require a very elaborate algorithm, whereas the computation of rough z ranges and the ray–object intersection is usually much simpler.

If we consider the ZZ-buffer as an acceleration mechanism for distributed ray tracing, then its programming complexity is no greater than that of other acceleration methods [1, 13, 17, 29].

4.3 Performance

In this section we give analytic estimates of the ZZ-buffer algorithm's running time and compare it with similar estimates for some competing algorithms.

It is very hard to compare the performance of algorithms when they are implemented by different programmers on different machines and applied to different scenes. In our analysis we will be forced to make several simplifying assumptions about the costs of individual operations and about the parameters of the scene being rendered. The results we obtain may be enlightening but not necessarily conclusive.

Cost of ZZ-buffer algorithm

We begin by considering the cost of the ZZ-buffer algorithm without shadows. This cost can be broken into three major terms: the preprocessing cost of building the ZZ-buffer P, the cost of determining visibility V, and the cost of shading S.

$$P = \langle \textit{num. objects} \rangle \times \langle \textit{avg. cells per object} \rangle \times \langle \textit{cost of estimating Z range in cell} \rangle$$

$$V = \langle \textit{num. complex pixels} \rangle \times \langle \textit{samples per pixel} \rangle \times \langle \textit{avg. HitTests per complex sample} \rangle \times \langle \textit{cost of one HitTest call} \rangle$$

$$S = (\langle \textit{num. complex pixels} \rangle \times \langle \textit{samples per pixel} \rangle + \langle \textit{num. simple pixels} \rangle) \times \langle \textit{avg. transparent layers} \rangle \times \langle \textit{cost of shading one sample} \rangle$$

The quantities $\langle$*num. simple pixels*$\rangle$ and $\langle$*num. complex pixels*$\rangle$ refer to the number of pixels that are rendered by the ZZ-buffer algorithm using a single sample or multiple samples, respectively.

The quantity $\langle$*avg. HitTests per complex sample*$\rangle$ is the number of tiles that must be considered when determining visibility for a given sample in a complex pixel, including any visible transparent tiles, as well as those tiles that are tested but are either missed by the ray or turn out to be hidden once their depths are determined exactly. For scenes consisting mostly of large, non-transparent objects, this quantity is typically 2 or less. For example, in the image shown in figure 11, its value turns out to be 1.1, since in most pixels, the tile list of the corresponding ZZ-buffer cell contains only a single opaque tile.

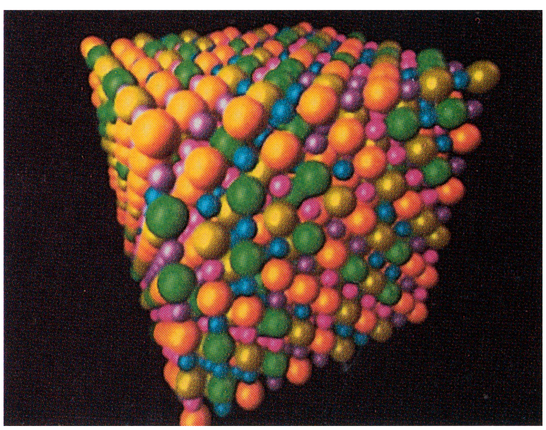

Figure 11. Many obscured objects.

The quantity $\langle$*avg. transparent layers*$\rangle$ is the number of tiles that are actually visible at an average sample point. This term is never greater than $\langle$*avg. HitTests per complex sample*$\rangle$ and is at most 1 if there are no transparent objects.

In our experience, the preprocessing cost P is almost always very small compared with the rendering costs V and S. Note that in determining visibility, if the first object in the tile list is opaque and covers the pixel entirely, then no visibility tests are performed, and the object's shade is sampled at just a single point. Note also that in any case the ZZ-buffer algorithm only shades objects that have been determined to be visible.

First-order distributed ray tracing

The ZZ-buffer algorithm we have described essentially implements the first and last steps of distributed ray tracing: it shoots rays from the eye to the first visible surface, and from the surface to each light source, but does not shoot any secondary rays to simulate inter-object reflections and refractions.

Indeed, the ZZ-buffer can be thought of as merely a simple indexing scheme for optimizing the primary ray–object intersection tests, and for deciding when we can get by with a single sampling ray per pixel. Therefore, we should compare the performance of the ZZ-buffer algorithm against that of other indexing schemes for ray tracing, such as the "item buffer" of Weghorst, Hooper, and Greenberg [29], the space subdivision method of Glassner [13], the hierarchical bounding volume scheme of Kay and Kajiya [17], and the ray classification scheme of Arvo and Kirk [1]. Due to space limitations, we will discuss here only the first of these schemes.

The "item buffer"

Our algorithm resembles most closely the "item buffer" of Weghorst, Hooper, and Greenberg. Although both the ZZ-buffer and the item buffer provide image-space indexes for optimizing the first ray-intersection test, the ZZ-buffer algorithm has the advantage of also using the index to recognize those places where a single sample per pixel will suffice for testing visibility and performing shading. Moreover, as we have already described, the ZZ-buffer algorithm incorporates stochastic sampling to avoid aliasing artifacts and can use its z intervals to prune the lists of potentially visible objects quickly. Finally, our mechanism can also be used for shadows and penumbrae, whereas the item buffer, because it does not store any z information, cannot.

The Z-buffer

If we want to compare our algorithm against the Z-buffer, we should compare it against a version of the algorithm that can compute images of similar quality. To make images of this quality, we need a very high-resolution Z-buffer, with one cell per sample, as opposed to one cell per pixel. Furthermore, to avoid aliasing, the z values and shades stored in each cell would have to be computed at jittered sample positions.

Rendering with such a high resolution Z-buffer amounts to testing every object against every sample ray within the object's projected bounding box. This algorithm is equivalent to a distributed ray tracing scheme in which the only optimization used is a naïve two-dimensional bounding box. Such an algorithm is obviously much more expensive than the ZZ-buffer algorithm.

The A-buffer

Compared to the A-buffer, The ZZ-buffer algorithm saves time by not clipping objects to pixel boundaries, by computing shades only for visible objects, by using only a single sample per pixel (as opposed to the A-buffer's implicit 32) whenever visibility within the pixel is simple, and by getting away with fewer samples (because the samples are jittered) when the visibility test is complex. The ZZ-buffer also saves by performing coarse visibility tests in the first phase that may allow it to avoid computing the precise z values that the A-buffer must compute everywhere. The ZZ-buffer also allows the possibility of computing texture colors with the same distributed rays that it uses to test visibility, whereas texturing for an A-buffer would have to involve a more complicated area-averaging technique [8, 32]. Finally, the ZZ-buffer can be built at low resolution for simple scenes, saving still more, whereas the resolution of the A-buffer is intrinsically tied to the resolution of the image being produced.

An additional cost of the A-buffer algorithm arises from its need to convert all geometric primitives into polygons. This conversion is expensive, not only because it needs to be performed for every primitive in the scene, whether visible or not, but also because every small polygon produced by the conversion contributes a fragment to every pixel that it touches. Thus, the total number of fragments inserted into the A-buffer is likely to be several times the number of tiles inserted into the ZZ-buffer for the same scene.

Note also that since the ZZ-buffer's z estimates do not need to be accurate, they can be encoded with less precision (say, 8 or 16 bits) than what is ordinarily required by the Z-buffer and A-buffer algorithms. Note that these low-resolution z values suffice even for scenes with deep perspective.

On the other hand, the main place where the A-buffer than the ZZ-buffer is that it can quickly compute a pixel mask for all 32 samples under a polygon, taking advantage of coherence, whereas the ZZ-buffer must often resort to tracing each jittered sampling ray separately. Even testing a mere 4 or 9 jittered samples separately may be more expensive than testing all 32 uniform samples at once.

A somewhat more careful analysis leads to the conclusion that the ZZ-buffer algorithm is likely to perform better than the A-buffer algorithm whenever the average number of A-buffer fragments that contribute to a pixel is greater than the number of samples per pixel used by the ZZ-buffer. Thus, the ZZ-buffer is likely to be a win for scenes that have many occluded objects or many subpixel-sized features (either as a part of the original model, or as a result of the A-buffer's conversion of primitives into small polygons).

5 Implementations

The ZZ-buffer rendering algorithm has been implemented in two forms: as a production system at Sogitec Audiovisuel in Paris, and as an experimental system at DEC Systems Research Center (SRC) in Palo Alto. The implementation at Sogitec has been used for commercial production since 1986 and has produced dozens of animated pieces. Since then it has been licensed to RTL Productions of Luxembourg, the Ecole Nationale Supérieure des Arts Décoratifs in Paris, and the Centre de Recherche de la Technologie de l'Audiovisuel et de la Communication in Rennes.

The original implementation at Sogitec was written in FORTRAN on a Perkin Elmer computer. That implementation has since been rewritten in C for a Convex C1-XP computer, and aspects of the code have been parallelized substantially. Figures 12 and 13 show two scenes from animations produced at Sogitec with the C implementation. Both images are 768 $\times$ 576 pixels in size, using 4 jittered samples in complex pixels. Figure 12 has about 3000 polygons and took about 15 minutes to produce; figure 13 has about 50,000 polygons and took about 25 minutes.

The implementation at DEC SRC was written in Modula2+ on the Firefly [27], a locally developed experimental workstation with four CVAX microprocessors. All the other figures in the paper were produced with this system. The small images of the comb and checkerboard were rendered at 200 $\times$ 150 pixels each, using 4 and 36 samples per pixel, while the larger images were rendered at 400 $\times$ 300 pixels at 16 samples per pixel.

6 Further Work

There are many aspects of the ZZ-buffer algorithm that suggest further research. Here are some examples:

Adaptive supersampling. We would like to incorporate an adaptive method for choosing the number of samples per pixel, based on estimates of the local variation in th eimage, such as the one proposed by Lee, Redner, and Uselton [18].

Exploiting Image Coherence. The idea is that the scan conversion routines can keep pointers from each object to the tiles derived from it, and keep re-using the tile records from the previous scanline whenever possible, updating their *zmin*, *zmax*, and *opaque* fields. This may substantially reduce the costs of record allocation and sorting.

Quad-tree ZZ-buffer. There may be some advantage in letting the size of the ZZ-buffer cells be determined adaptively, by starting with a single cell and then subdividing any cells that become too complex. Such an algorithm would resemble Warnock's [28] and would essentially amount to replacing the ZZ-buffer array by a quad-tree data structure.

Polygonal meshes. When scan converting a collection of polygons that come from a single continuous polygonal mesh, it would be highly desirable to eliminate any tiles that are completely obscured by the mesh as a whole. Right now the internal edges of the mesh create many tiles with *opaque* = `false`, and the program is not smart enough to realize that those tiles fit together into an opaque surface.

Recursively-defined Objects. It seems it would be easy to extend the ZZ-buffer algorithm to directly handle objects that are defined recursively, such as fractal terrains [11, 15] or Bézier patches [10].

Constructive solid geometry. Objects defined implicitly by formulas of constructive solid geometry also seem easy to incorporate into the ZZ-buffer algorithm. In the rendering phase, such objects would be handled in exactly the same way as for ordinary ray tracing [2, 26]. The novelty comes in scan conversion, where it is necessary to estimate the *opaque* flag of the union, intersection, and difference of primitives.

7 Conclusion

The ZZ-buffer algorithm produces images of exceptional quality. It can render simple scenes fast, and render complex scenes at a speed competitive with that of other algorithms. The ZZ-buffer is also quite simple—indeed, we have given most of its essential code in this paper. The versatility and robustness of the ZZ-buffer algorithm have been proven by its acceptance and everyday use in several commercial production environments.

Acknowledgements

We would like to thank Alain Behar, Olivier Multon, and Adrian Freed for illuminating discussions during the early development of this algorithm. We are grateful to Dominique Pochat for daring to be the first real user of this system. We would also like to thank Stephan Metz and Jean-Luc Ortega for kindly providing us examples of recent commercial work produced at Sogitec. Finally, we thank the DEC Systems Research Center for its support, as well as the people at Sogitec for their encouragement and *amitié*.

References

[1] Arvo, J. and Kirk, D., "Fast Ray Tracing by Ray Classification," Computer Graphics (SIGGRAPH '87) vol. 21 no. 4 (July 1987), 55–64.

[2] Atherton, P. R., "A Scanline Hidden Surface Removal Procedure for Constructive Solid Geometry," Computer Graphcis (SIGGRAPH '83) vol. 17 no. 3 (July 1983), 73–82.

[3] Blinn, J. F., "A Generalization of Algebraic Surface Drawing," ACM Transactions on Graphics, vol. 1 no. 3 (July 1982), 236–256.

[4] Carpenter, L., "The A-buffer, an Antialiased Hidden Surface Method," Computer Graphics (SIGGRAPH '84) vol. 18, 3 (July 1984), 103–108.

[5] Catmull, E., "A Subdivision Algorithm for Computer Display of Curved Surfaces," Ph.D. Thesis, University of Utah, Salt Lake City, December 1974.

[6] Cook, R. L., "Stochastic Sampling in Computer Graphics," ACM Transactions on Graphics 5, 1 (January 1986), 51–72.

[7] Cook, R. L., Porter, T., and Carpenter, L., "Distributed Ray Tracing," Computer Graphics (SIGGRAPH '84) vol. 18 no. 3 (July 1984), 137–145.

[8] Crow, F. C., "Summed-Area Tables for Texture Mapping," Computer Graphics (SIGGRAPH '84) vol. 18 no. 3 (July 1984), 207–212.

[9] Dippé, M. A. Z., and Wold, E. H., "Antialiasing through Stochastic Sampling," Computer Graphics (SIGGRAPH '85) vol. 19 no. 3 (July 1985), 69–78.

[10] Faux, I. D. and Pratt, M. J., Computational Geometry for Design and Manufacture, Ellis Horwood Ltd., Wiley and Sons (1979).

[11] Fournier, A., Fussell, D., and Carpenter, L., "Computer Rendering of Stochastic Models," Communications of the ACM vol. 25 no. 6 (June 1982).

[12] Fournier, A. and Fussell, D., "On the Power of the Frame Buffer," ACM Transactions on Graphics vol. 7 no. 2 (April 1988), 103–128.

[13] Glassner, A. S., "Space Subdivision for Fast Ray Tracing," IEEE Computer Graphics and Applications, vol. 4 no. 10 (October 1984), 15–22.

[14] Haines, E. A. and Greenberg, D. P., "The Light Buffer: A Ray Tracer Shadow Testing Accelerator," IEEE Computer Graphics and Applications, vol. 6 no. 9 (September 1986), 6–15.

[15] Kajiya, J. T., "New Techniques for Ray Tracing Procedurally-Definied Objects," ACM Transactions on Graphics, vol. 2 no. 3 (July 1983), 161–181.

[16] Kay, D. S., "A Transparency Refraction and Ray Tracing Model for Computer Synthesized Images," Master's thesis, Cornell University, Ithaca, New York, 1979.

[17] Kay, T. L. and Kajiya, J., "Ray Tracing Complex Scenes," Computer Graphics (SIGGRAPH '86) vol. 20 no. 4 (August 1986), 269–278.

[18] Lee, M. E., Redner, R. A., and Uselton, S. P., "Statistically Optimized Sampling for Distributed Ray Tracing," Computer Graphics (SIGGRAPH '85) vol. 19 no. 3 (July 1985), 61–67.

[19] Mitchell, D. P., "Generating Antialiased Images at Low Sampling Densities," Computer Graphics (SIGGRAPH '87) vol. 21 no. 4 (July 1987), 65–72.

[20] Mitchell, D. P., and Netravali, A. N., "Reconstruction Filters in Computer Graphics," Computer Graphics (SIGGRAPH '88) vol. 22 no. 4 (August 1988), 221–228.

[21] Moore, R. E., Interval Analysis, Prentice-Hall, Englewood Cliffs, N. J. (1966).

[22] Newman, W. M., and Sproull, R. F., "Principles of Interactive Computer Graphics," Second Edition, McGraw Hill (1979), pages 232–239.

[23] Porter, T. and Duff, T., "Compositing Digital Images," Computer Graphics (SIGGRAPH '84) vol. 18 no. 3 (July 1984), 253–259.

[24] Pratt, W. K., Digital Image Processing, Wiley and Sons, New York (1978).

[25] Reeves, W. T., Salesin, D. H., and Cook, R. L., "Rendering Antialiased Shadows with Depth Maps," Computer Graphics (SIGGRAPH '87) vol. 21 no. 4 (July 1987), 283–291.

[26] Roth, S. D., "Ray Casting for Modeling Solids," Computer Graphics and Image Processing vol. 18 (1982), 109–144.

[27] Thacker, C. P., Stewart, L. C., and Satterthwaite, E. H. Jr., "Firefly: A Multiprocessor Workstation," DEC Systems Research Center, Research Report 23 (December 1987).

[28] Warnock, J. E., "A Hidden-Surface Algorithm for Computer-Generated Halftone Pictures," Computer Science Department, University of Utah, TR 4-15 (June 1969).

[29] Weghorst, H., Hooper, G., and Greenberg, D. P., "Improved Computational Methods for Ray Tracing," ACM Transactions on Graphics, vol. 3, no. 1 (January 1984), 52–69.

[30] Whitted, T., "An Improved Illumination Model for Shaded Display," Communications of the ACM vol. 23 (1980), 343–349.

[31] Williams, L., "Casting Curved Shadows on Curved Surfaces," Computer Graphics (SIGGRAPH '78) vol. 12 no. 3 (August 1978), 270–274.

[32] Williams, L., "Pyramidal Parametrics," Computer Graphics (SIGGRAPH '83) vol. 17 no. 3 (July 1983), 1–11.

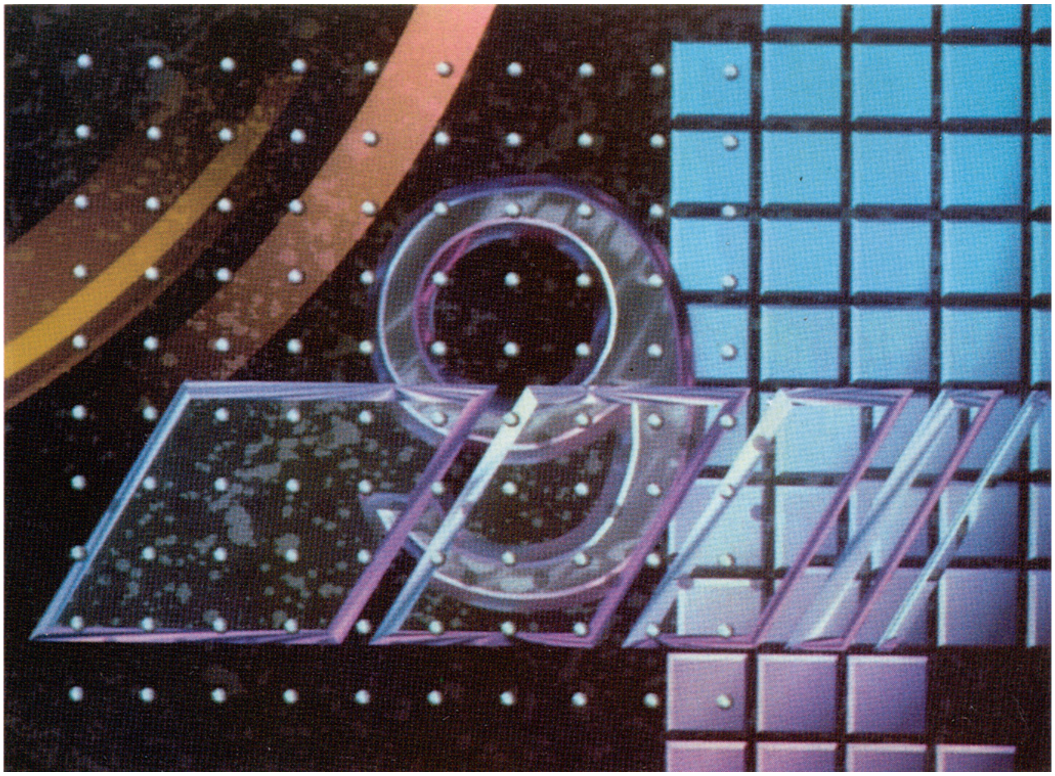

Figure 12. A scene from “Countdown.”

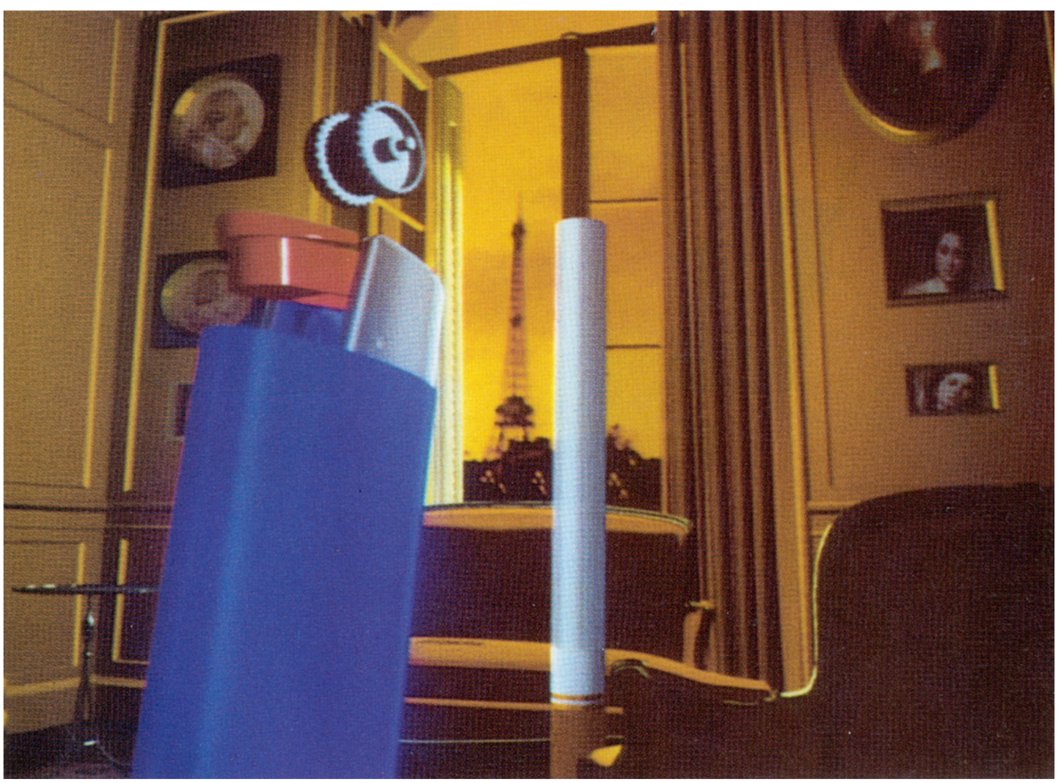

Figure 13. A scene from “Jumpin’ Jacques Splash.”