HARDWARE ASSISTANCE FOR Z-BUFFER VISIBLE SURFACE ALGORITHMS

Kellogg S. Booth, David R. Forsey, and Alan W. Paeth

Computer Graphics Laboratory, Department of Computer Science
University of Waterloo, Waterloo, Ontario, Canada N2L 3G1
Tel: (519) 888-4534, E-Mail: KSBooth%watCGL@Waterloo.CSNet

ABSTRACT

The well-known "z-buffer" algorithm for solving the visible surface problem has a number of points in its favor, the main one being that it is amenable to very efficient hardware implementation at little additional cost in many existing frame buffer systems. The traditional software implementation of the algorithm assumes explicit initialization of both the image buffer and the z-buffer before each image is generated. This paper describes a simple technique for synchronizing initialization and image generation so the two can be performed in parallel, allowing complete overlap to be achieved and effectively eliminating the time needed for explicit initialization of the frame buffer. The technique assumes a modest investment in additional hardware within the frame buffer.

RÉSUMÉ

Parmi les algorithmes de surfaces cachés, l'algorithme du "z-buffer" a un certain nombre d'avantages à son actif. Le plus important de ceux-ci étant que cet algorithme est peu couteux à implémenter au niveau hardware dans les systèmes actuels de "frame-buffer" Les techniques traditionnelles d'implémentation de cet algorithme forcent le logiciel à initialiser explicitement le "z-buffer" et le buffer image avant le transfert de l'image. Cet article décrit une technique simple qui permet de synchroniser l'initialisation et la création de l'image afin qu'elles puissent être réalisées simultanément. Ceci permet d'éliminer le temps perdu lors de l'initialisation du "frame-buffer." Cette technique suppose un faible investissement en matériel additionnel à l'intérieur du "frame-buffer."

Keywords: *double-buffering, frame buffer, real-time, visible surfaces, z-buffer.*

INTRODUCTION

The problem of computing only the *visible surfaces* of a scene has by now been well-studied [12]. One approach that has gained popularity is the *z-buffer* algorithm first described by Catmull [4,5]. With a z-buffer the depth-sort required of a visible surface algorithm is accomplished by maintaining, for each pixel, a record of the *z-depth* of the object whose intensity (color) is stored at that pixel. As subsequent objects are scan converted into the frame buffer, their z-depths are compared and used to decide whether the new object is in front of or behind the object currently displayed at each pixel. In the former case both the intensity and z-depth are changed for the pixel, but in the latter case no action is taken. A complete explanation of the z-buffer algorithm appears in standard text books on computer graphics [7,10].

In the following sections we first define our model of a frame buffer and then look at different ways of adding additional hardware to the frame buffer to speed up the z-buffer algorithm. The first approach almost doubles the amount of memory in the frame buffer and is presented solely to motivate the other two. The second approach adds only a single bit to each pixel but requires a more complicated memory controller that could lead to significant timing problems in the video chain. The third approach adds two more bits to each pixel and hence admits an implementation that requires no additional function in the memory controller beyond what is available in current frame buffers, although a modest change is still required to hardware further down the video chain.

The basic z-buffer algorithm performs well in almost all respects except for considerations of *antialiasing*. This deficiency stems from the fact that the z-buffer maintains depth information on a pixel-by-pixel basis, and thus has no way to discriminate objects at subpixel resolution. This is unfortunate because aliasing artifacts introduced by the scan conversion process can be very objectionable in practice. Some researchers have suggested techniques to incorporate antialiasing strategies into a z-buffer algorithm, but those techniques either require auxiliary storage or

additional processing time beyond the basic scan conversion algorithm, or they fail to achieve the desired level of image quality [[3,6,8]]. The approaches described in this paper apply to z-buffer algorithms enhanced for antialiasing, although we do not address the issue explicitly. Instead, we adopt the attitude that an important application of the z-buffer technique is to high-performance raster systems for which real-time or quasi-real-time performance is desired and that proper antialiasing is a luxury as yet not consistent with that goal.

We distinguish between the *update rate*, the rate at which new images are computed by the algorithm, from the *refresh rate*, the rate at which the computed images are displayed on the monitor. The update rate is almost always established by limits in processing power and the desire to display complex images and thus tends to remain a bottleneck even as advances are made in hardware and software, whereas the refresh rate is set by the need to overcome flicker effects inherent in the human visual system and can be regarded as a relative constant.

Our definition of *real-time* will be that a complete image is generated within a single refresh cycle (usually 1/30 or 1/60 of a second). The notion of *quasi-real-time* will imply image generation that closely approximates that rate (at worst an update happens every 1/5 of a second). Ron Baecker has already championed the claim that such rates give an acceptable illusion of continuous simulation if the update rate and the refresh rate are suitably synchronized [[2]]. To be effective, a steady refresh of the current image must be maintained throughout the scan conversion process for the next image. Because of this, we will be interested only in the *double-buffered* version of the algorithm in which a *refresh processor* displays one image while an *update processor* is generating the next image.

With this set of ground rules, we are ready to discuss the performance of the z-buffer algorithm and to examine alternatives to the traditional implementation. The operations performed by the versions of the z-buffer algorithm that we will consider remain largely the same in each of the implementations. The differences lie in the way that the operations are partitioned between the two processors (the update processor and the refresh processor) and in the way that the two processors synchronize their operations.

THE FRAME BUFFER

Our model assumes that a frame buffer contains a large amount of *pixel memory* indexed by two-dimensional (x,y) addresses and that each pixel is divided into fields composed of a number of bits. For our purposes at least three fields are necessary in a pixel. These fields will be designated the I_0 and I_1 fields (two *intensity buffers*, one for the image currently being displayed by the refresh processor and the other for the image being computed by the update processor) and the Z field (a single *depth buffer*). Frame buffer memory is dual-ported to allow the update processor to read and write pixels (randomly) at the same time that the refresh processor reads pixels at video rates (in scan-line order). The refresh processor passes the pixels down the *video chain* where the color information stored in each pixel is converted to analog signals suitable for display on a monitor, possibly after interpretation by lookup tables or other devices whose function is not important to the present discussion.

Our model for a frame buffer is patterned after the Adage/Ikonas RDS 3000, the hardware on which we have implemented these algorithms. Most of the ideas presented here apply to other frame buffer architectures, although we do assume that the video chain is similar to the control, crossbar, and lookup table modules available in the Ikonas [[1,9]]. Not all of these features are required for most of the z-buffer techniques, although the final algorithm presented here actually assumes a slightly enhanced crossbar switch over what is supplied by Adage. Our goal is not to restrict attention to a particular frame buffer architecture, but to point toward general hardware features that will substantially enhance the performance of z-buffer algorithms at modest cost.

Double-buffering is accomplished in a frame buffer by the refresh processor reading from I_0 during odd update cycles and from I_1 during even update cycles, thus allowing the update processor to use I_1 and Z for scan conversion during odd update cycles and I_0 and Z during even update cycles. Only one z-depth field is necessary because once the image has been rendered, its z-depths are no longer needed.

The selection of specific fields for reads and writes by both the update processor and the refresh processor may be accomplished by masking and shifting (either in hardware or through a combination of hardware and software) or by using address offset registers when the various fields are stored in different areas of frame buffer memory. These details are not important for the discussion, so we will assume that the frame buffer maintains all of the fields associated with a pixel within a single "word" and that both processors are capable of selecting particular fields with no penalty in time. It is convenient to assume that this is accomplished by *mask registers* and *shift registers*, associated with each processor, that perform selective load and store operations to only those bits indicated by the mask, leaving the other bits of a pixel unchanged.

Any number of bits may be associated with the two intensity buffers. Common configurations use 8 bits (with color lookup tables to achieve a full color space) or 24 bits (8 bits each of red, green and blue). The z-depth must be able to discriminate objects within the scene, so between 8 and 32 bits are commonly assumed. Sutherland and Hodgman discuss a scaling strategy for making the best use of the precision

available [11]. This totals to from 24 to 80 bits per pixel, depending upon the amount of color and z-depth desired. The memory requirement can be reduced somewhat if only the intensity buffer currently being displayed is located within the frame buffer itself (the other fields being stored on the host) but our technique is designed for high-performance systems in which all of the memory resides within the frame buffer to achieve the necessary update and refresh rates. Only the two intensity buffers are accessed by the refresh processor in any of these schemes, so some savings in cost could be achieved by making the z-depth memory single-ported, but this might preclude using the memory for other purposes and is thus a less general architecture.

We will label each update cycle by an integer, but often will only use its even or odd parity (the low-order bit). Thus many places in our algorithms where k is manipulated as an integer modulo some base (usually two) the manipulation can be implemented with simple bit operations such as complementation, rather than with the more expensive increment and modulus operations.

The next three sections present increasingly sophisticated versions of the z-buffer algorithm. The first is the standard implementation, suitable for a very basic frame buffer. The subsequent versions achieve increased performance by partitioning the calculation differently among the update and refresh processor and by using modest hardware assistance to synchronize the calculations.

SOLUTION #1
TRADING OFF MEMORY FOR PROCESSING TIME

The basic appeal of the z-buffer algorithm is that it affords a complete solution to the visible surface algorithm for little more than the time required to perform simple scan conversion without the visible surface calculation. The algorithm is usually implemented entirely in the update processor, with the refresh processing serving only to display the resulting image on the monitor. The two procedures `Update#1` and `Refresh#1` shown in Figure 1 express the interlocked cycles in a standard implementation of the z-buffer algorithm. Normally the refresh cycle would be implemented entirely in hardware, but we describe it here as if it were implemented in software to provide a uniform presentation of the two processes. The processes are loosely synchronized in this basic version of the z-buffer algorithm. At the start of each update cycle the shared variable k is incremented and this causes the refresh processor to swap image buffers with the update processor. Actual implementations would usually include a provision to further synchronize the double-buffering so that buffers are swapped only at the end of a complete frame or field time.

This is the standard z-buffer algorithm presented in text books and is easily implemented on most frame buffers. The main procedure invokes (once) the setup

```
PROCEDURE Initialize#1;
  k := 0;
  FOR y := maxY DOWNTO 0 DO
    FOR x := 0 TO maxX DO
      I0[x,y] := background;
    OD;
END Initialize#1;

PROCEDURE Update#1;
  WHILE true DO
    k := k+1;
    FOR y := maxY DOWNTO 0 DO
      FOR x := 0 TO maxX DO
        Ik mod 2[x,y] := background;
        Z[x,y] := ∞;
      OD;
    FOR every object in the scene DO
      FOR every pixel (x,y) in the object DO
        IF (object[x,y].z < Z[x,y]) THEN
          Ik mod 2[x,y] := object[x,y].color;
          Z[x,y] := object[x,y].z;
        FI;
      OD;
    OD;
    { optional wait for next frame }
  OD;
END Update#1;

PROCEDURE Refresh#1;
  WHILE true DO
    FOR y := maxY DOWNTO 0 DO
      FOR x := 0 TO maxX DO
        display Ik-1 mod 2[x,y];
      OD;
    OD;
  OD;
END Refresh#1;
```

Figure 1. The Standard Z-Buffer Algorithm

in `Initialize#1` and then invokes (in parallel) the two procedures `Update#1` and `Refresh#1` which never terminate, but cycle continuously as the double-buffering scheme alternately updates the two image buffers. The refresh processor is merely cycling through memory performing the standard frame buffer readout to the video hardware. If the algorithm is being used to generate a single frame, there is really nothing to discourage its use. But if the algorithm is being used to generate a sequence of frames (as assumed here) the algorithm does not fully utilize the available hardware.

In this situation there is a bottleneck that may potentially degrade performance. The z-buffer algorithm requires that the intensity and z-depth fields be reset to their initial values (background color and

infinity) before each update cycle. The procedure Update#1 performs this initialization explicitly at the beginning of each update cycle. This can be time-consuming for two reasons. The first is that the update processor can only perform initialization when it is not performing scan conversion and thus the full bandwidth of the update processor is not available for scan conversion, an unfortunate consequence because real images frequently require substantially more update time than refresh time.

This is related to the second reason, which is that even update processors with special purpose hardware may not be able to write all of the pixels within the frame buffer in one refresh cycle. The refresh processor reads multiple pixels during a single memory cycle because it looks at pixels in scan-line order and thus can access multiple memory chips in parallel. This allows it to achieve a complete refresh within one frame time. The update processor typically does not do this because it is designed for random access to the frame buffer. The result is that one or more refresh cycles may be "wasted" between successive update cycles while the update processor is busy initializing instead of rendering. This degrades the real-time or quasi-real-time performance of the system by a significant percentage.

Because it accesses multiple pixels during a single memory cycle, the refresh processor is capable of performing the initialization in a single refresh cycle. Some frame buffers support this by allowing the refresh processor to change values in selected fields of pixel memory as each pixel is written back into memory after being read during the refresh cycle [[1]]. This allows the refresh processor to initialize the second image buffer and the z-depth buffer in a single refresh cycle.

Unfortunately, unless initialization is synchronized with image generation there is little advantage to this approach because the update processor must wait for at least one complete refresh cycle after update cycle k to insure that the refresh processor has completely reset both the I_{k+1} and Z fields before it can begin update cycle k+1. This means that the update processor will be idle a significant amount of the time (recall that the slowest update rate for quasi-real-time is 1/5 second so that even with a refresh cycle of 1/60 second the image processor would be idle more than 8% of the time — in the worst case the processor would waste 50% of its bandwidth while maintaining a 1/30 second refresh cycle and a 1/15 second update cycle). The percentage of idle time for the update processor is an important consideration because it determines an upper bound on the complexity of the image that can be rendered. This problem can be overcome by the addition of extra hardware, in this case a substantial increase in frame buffer memory.

SOLUTION #2
TRADING MORE MEMORY FOR PROCESSING TIME

```
PROCEDURE Initialize#2;
  k := 0;
  FOR y := maxY DOWNTO 0 DO
    FOR x := 0 TO maxX DO
      I0[x,y] := background;
      Z0[x,y] := ∞;
    OD;
END Initialize#2;

PROCEDURE Update#2;
  WHILE true DO
    k := k+1;
    FOR every object in the scene DO
      FOR every pixel (x,y) in the object DO
        IF (object[x,y].x < Zk mod 2[x,y]) THEN
          Ik mod 3[x,y] := object[x,y].color;
          Zk mod 2[x,y] := object[x,y].z;
        FI;
      OD;
    OD;
    { mandatory wait for next frame }
  OD;
END Update#2;

PROCEDURE Refresh#2;
  WHILE true DO
    FOR y := maxY DOWNTO 0 DO
      FOR x := 0 TO maxX DO
        display Ik-1 mod 3[x,y];
        Ik+1 mod 3[x,y] := background;
        Zk+1 mod 2[x,y] := infinity;
      OD;
    OD;
  OD;
END Refresh#2;
```

Figure 2. The Triple-Buffered Z-Buffer Algorithm

To achieve real-time or quasi-realtime performance a third image field I_2 can be added to the frame buffer along with a second z-depth field Z_1 (the original depth field becomes Z_0). In this case the update processor cycles between three image memories, with the refresh processor displaying from I_{k-1}, the update processor writing into I_k, and the refresh processor initializing I_{k+1} (all subscripts for I are now modulo 3 instead of modulo 2). Similarly, the update processor uses Z_k for its visible surface calculation while the refresh processor is initializing Z_{k+1} (these subscripts are modulo 2 since only two depth buffers are required).

As long as each update cycle requires at least one entire refresh cycle (a modest assumption since a faster update rate would imply that the image was being updated faster than it was being viewed on the monitor) the refresh processor will be able to initialize a new

image and depth buffer in time for each update cycle, thus freeing the update processor from any overhead for initialization. Procedures Update#2 and Refresh#2 to accomplish this are straightforward modifications to Update#1 and Refresh#1.

The clear drawback to this scheme is the massive increase in frame buffer memory. The requirement for image memory has increased by 50% (from two buffers to three) and the requirement for depth memory has increased by 100% (from one buffer to two). For the case of a full 24-bit image buffer and a 32-bit depth buffer, this is a total of 136 bits per pixel, an increase of 70%. While this might be appropriate for the increased performance, we are at best getting an improvement that is of the same order of magnitude as the increase in memory cost. We can do much better.

SOLUTION #3
TRADING OFF HARDWARE COMPLEXITY FOR LESS MEMORY

An alternative is to use one additional bit in the frame buffer as a cycle counter (a *dirty bit*) to achieve complete overlap of image generation and initialization while avoiding the necessity of adding an additional set of image and depth buffers. As for Solution #2, this will in fact achieve an update rate that is equal to the refresh rate for simple scenes (something not achievable with the standard software z-buffer algorithm) but at far less hardware cost. We assume that the frame buffer has been extended to include a one-bit field D containing the parity (low-order bit) of k, the update cycle counter.

The frame buffer is initialized once, before the actual z-buffer algorithm begins, so that I_0 is the "background" color and D is 0, the parity of the first image. The setting of I1 and Z are arbitrary. The update processor begins image generation cycle 1 with the refresh processor initializing both I_1 and Z. When the z-buffer algorithm has generated its first image, the refresh processor begins displaying from I_1 and initializing both I_0 and Z, but it performs the initialization selectively using the cycle number and information kept in the D field of each pixel. The system assumes a steady-state operation in which the two processors synchronize their activity after each update cycle through the shared cycle number and the D field.

This is accomplished in the following way. The update processor proceeds as it normally would, assuming that both I_k and Z have been initialized previously by the refresh processor, even though the refresh processor may not have visited some (or all) of the pixels. The key idea is that each time the update processor performs a depth comparison (testing the z-depth of an object against the value stored in the Z field of a particular pixel) it biases the comparison in favor of the new z-depth (that of the object) if the D field does not match the parity of the update cycle

```
PROCEDURE Initialize#3;
  k := 0;
  FOR y := maxY DOWNTO 0 DO
    FOR x := 0 TO maxX DO
      I0[x,y] := background;
      D[x,y] := 0;
    OD;
END Initialize#3;

PROCEDURE Update#3;
  WHILE true DO
    k := k+1;
    FOR every object in the scene DO
      FOR every pixel (x,y) in the object DO
        IF (D[x,y] = k-1 mod 2)
            OR (object[x,y].x < Z[x,y]) THEN
          Ik mod 2[x,y] := object[x,y].color;
          Z[x,y] := object[x,y].z;
          D[x,y] := k mod 2;
        FI;
      OD;
    OD;
    { mandatory wait for next frame }
  OD;
END Update#3;

PROCEDURE Refresh#3;
  WHILE true DO
    FOR y := maxY DOWNTO 0 DO
      FOR x := 0 TO maxX DO
        display Ik-1 mod 2[x,y];
        IF D[x,y] = k-1 mod 2 THEN
          Ik mod 2[x,y] := background;
          Z[x,y] := infinity;
          D[x,y] := k mod 2;
        FI;
      OD;
    OD;
  OD;
END Refresh#3;
```

Figure 3. The Z-Buffer Algorithm Using A Dirty Bit

number. This in effect allows the update processor, by checking the D field, to detect those pixels for which the refresh processor has yet to perform the appropriate initialization and to substitute the value infinity for whatever (incorrect) z-depth appears in the frame buffer. The update processor always re-writes the D field with the parity of the current update cycle number each time it stores into the frame buffer to avoid the problem of the refresh processor mistakenly initializing pixels that have already been used for the current update cycle.

The refresh processor must modify its operation so that it checks the D field before initializing a pixel. Were this not the case, it might overwrite a pixel that the update processor had already computed, since the initialization and update take place simultaneously. The refresh processor only performs an initialization operation if the D field is *not* the same parity as the current update cycle (put another way, it only initializes those pixels whose D fields equal k-1). Pixels being initialized have their D fields set to k (not for the benefit of the update processor, but so the refresh processor knows to re-initialize them during the update cycle k+1).

For this scheme to work two assumptions must hold. The first is that the refresh processor must be allowed to complete at least one complete cycle between update cycles. As we have already noted, this is a reasonable assumption and is easily guaranteed by a simple test performed at the start of each frame. The second assumption is that the refresh processor makes its access to memory in a single atomic operation ("read-modify-write"). If this were not the case, the refresh processor might overwrite a pixel whose D field changed between the time that it was read and the time that it was written back to memory. The implication of this assumption is that the refresh processor must have a reasonably sophisticated interface to frame buffer memory – it is fetching multiple pixels in parallel, all of which must have their D fields checked and their I and Z fields modified in a single memory cycle.

The procedures **Update#3** and **Refresh#3** indicate the z-buffer algorithm using the D field to overlap initialization by the refresh processor with image generation by the update processor. The algorithm as stated assumes that the update processor only begins a new cycle during the start of a new frame. This can be weakened significantly to the requirement that the update processor not begin a new cycle until the refresh processor has performed at least one complete refresh cycle since the last update cycle began (our standard assumption). It may also be desirable to insist that the refresh processor not change its value of k except at the beginning of a frame (or at least a field) because of disturbing video effects.

Before continuing, the reader may want to verify that the algorithm works as stated, with no race conditions existing that depend on the order in which objects are scan converted by the update processor or the order in which pixels are initialized by the refresh processor. In doing so, special note should be made of the assumption that the refresh processor's memory accesses are atomic.

The cost in additional memory for this scheme is minimal. Only one extra bit is needed at each pixel. The increased sophistication in the refresh processor, however, is more substantial and may push up the cost of the video hardware significantly. Instead of performing a simple read-modify-write cycle (as it would for Solution #2) in which the new values written back to memory are independent of those read from memory (at least for the fields that change) the refresh processor must now check the status of the D field (a single bit) to determine whether the original contents are to be left in the $I_{k \bmod 2}$ and Z fields or if they are to receive initialization values. All of this must be performed in parallel for anywhere from 16 to 64 pixels, depending upon the design of the frame buffer memory interface. We can avoid this necessity, while still retaining the performance, by adding a few more bits to each pixel.

```
PROCEDURE Initialize#4;
  k := 0;
  FOR y := maxY DOWNTO 0 DO
    FOR x := 0 TO maxX DO
      D0 := false;
      D1 := false;
    OD;
END Initialize#4;

PROCEDURE Update#4;
  WHILE true DO
    k := k+1;
    FOR every object in the scene DO
      FOR every pixel (x,y) in the object DO
        IF (NOT Dk mod 3[x,y])
            OR (object[x,y].x < Z[x,y]) THEN
          Ik mod 2[x,y] := object[x,y].color;
          Z[x,y] := object[x,y].z;
          Dk mod 3[x,y] := true;
        FI;
      OD;
    OD;
    { mandatory wait for next frame }
  OD;
END Update#4;

PROCEDURE Refresh#4;
  WHILE true DO
    FOR y := maxY DOWNTO 0 DO
      FOR x := 0 TO maxX DO
        IF Dk-1 mod 3[x,y] THEN
          display Ik-1 mod 2[x,y]
        ELSE
          display background;
          Dk+1 mod 3[x,y] := false;
        FI;
      OD;
    OD;
  OD;
END Refresh#4;
```

Figure 4. The Z-Buffer Algorithm Using 3 Dirty Bits

SOLUTION #4
TRADING BACK SOME OF THE MEMORY FOR HARDWARE SIMPLICITY

The reason the refresh processor's memory controller must be so complicated is that it must decide (very rapidly) which pixels must have their I and Z values modified. This dependence of certain bits within the pixel on other bits within the pixel may be difficult to determine, especially if the large pixel size requires some of the bits to reside on different boards. The solution proposed is to eliminate the necessity of checking the current pixel contents before deciding what to write back into memory during the refresh cycle. What we want is an *oblivious* memory controller, one which always writes the same pattern (or at least one which always changes the same bits to the same values) independent of the current pixel contents.

The trick is to use *three* dirty bits, one for each of the cycles k-1, k, and k+1. These can then be administered independently by the refresh controller. If we interpret D_k as a Boolean value (`true` or `false`) that tells whether the current pixel contents have been set during the corresponding update cycle, the job of the refresh processor becomes much simpler. During steady state, the refresh processor will be fetching pixels from I_{k-1} and initializing D_{k+1} while the update processor is modifying D_k and Z.

The only catch to this scheme is that pixels that are never rendered by the update processor (because they correspond to background) will remain uninitialized in their I and Z fields. This is not a problem if the refresh processor interprets the D field before passing pixel values on to the rest of the video chain. It must simply check D_{k-1} and if it is `false` (meaning that this pixel was never set during the previous update cycle) then it should pass on background color rather than what is stored in I_{k-1}. Hardware to perform this task is much simpler than the massive parallel checking required for Solution #2 because it can be performed on a pixel-by-pixel basis using techniques similar to lookup tables.

It is interesting to note that three dirty bits are necessary to implement this scheme. Two are not enough. D_{k-1} must remain untouched during update cycle k or else the refresh processor will become confused as to what is or is not background. D_k obviously must be initialized before update cycle k begins and cannot be changed except by the update processor. Neither field is free for initialization during update cycle k. Thus a third dirty bit D_{k+1} is required.

CURRENT IMPLEMENTATIONS

These algorithms are implemented on the Adage/Ikonas RDS 3000. Solution #1 is the standard z-buffer algorithm. The only change in the implementation from what has been presented is that the Z field is typically stored in off-screen pixel memory because the frame buffer has only 32 bits per pixel and 24 are used for intensity. On frame buffers with no off-screen memory the entire algorithm can be accommodated on-screen by decreasing the number of bits allocated to intensity and setting the lookup tables appropriately to ignore bits in the Z field as they are read out during display.

Solution #2 (the triple-buffered version) has also been implemented on the Ikonas, but with only limited intensity and z-depth information due to the requirement that all of the fields reside in on-screen memory. This stems from the fact that the *auto-clear* feature of the Ikonas (which writes zeros into memory during the refresh cycle, subject to a write mask that determines the bits in a pixel to be cleared) only processes visible pixels. Adopting conventions for intensity and z-depth that encode the background color and the maximum z-depth as zero allows existing hardware to handle the initialization in the refresh processor. A more natural encoding is possible if the auto-clear feature uses the shading registers (available on with the Ikonas GM memory boards) to set the values to be written into memory, rather than always writing zeros into the fields specified by the write mask registers. Buffer swapping is accomplished by manipulating the crossbar switch and the lookup tables.

Solution #3 (the dirty bit) is not directly implementable on the Ikonas because the auto-clear feature has no way of conditionally modifying bit fields. This is symptomatic of the objection raised earlier that this approach assumes more intelligence in the refresh processor's memory controller than is likely to exist in a frame buffer.

Solution #4 (three dirty bits) is easily implemented on the Ikonas using the convention that `true` is a 0 bit and `false` is a 1 bit. The auto-clear feature is used to initialize the dirty bits during refresh and a combination of the crossbar switch, the lookup tables, and the overlay option is used to modify the pixel readout to background color for pixels whose values have not been set by the update cycle. (The overlay option on the Ikonas allows certain bits – the appropriate dirty bit in our case – to select an alternate lookup table if the bits are non-zero. By setting all of the entries in the alternate lookup table to be the background color the correct modification is performed as pixels are read from memory during refresh.)

FURTHER CONSIDERATIONS

There is a question as to how dynamic the allocation of the various fields should be within a pixel. On the update processor all of the field selection can be fairly easily accomplished using masking and shifting. If the intensity and z-depth fields are multiples of eight bits, simple byte swapping logic can be used to present

an interface to the processor that is independent of the exact field being selected (i.e., the update processor will always present intensity of z-depth values as right-adjusted 32-bit values that will be automatically stored in the correct fields of a pixel). The dirty bits may have to be handled as special cases, although choosing appropriate conventions such as forcing all z-depths to be positive (and thus saving the sign bit for the dirty bit) could be employed. Generalized crossbar switches, similar to that on the Adage/Ikonas RDS 3000, used by both the update and refresh processors would clearly solve any efficiency issues related to problem.

In implementing such hardware, some *caveats* are in order. Any registers that allow values or fields to be selected should be both writable and readable. If necessary, there should be shadow registers so the values can be read back, although it is preferable that the registers themselves be readable directly.

It is worth noting that only the refresh circuitry needs a read-modify-write cycle that is atomic for the schemes to work. This is an important consideration, especially if the update processor is composed of distributed or pipelined tilers that separate their read accesses from their write accesses by multiple cycles. Such architectures are particularly useful for z-buffer algorithms because they increase parallelism and thus the update rate and yet require little or no synchronization among the multiple update processors because the z-buffer algorithm is itself inherently distributed. Our algorithms will not suffer in this case.

ACKNOWLEDGEMENTS

Schemes similar to those presented here have been implemented in custom hardware by Trillium Corporation in their flight simulator systems [13].

A lucid discussion of the issues involved in building the memory controller for the refresh processor is given in Whitton's excellent article on frame buffer design [14]. The crossbar switch for the Adage/Ikonas RDS 3000 was originally suggested by Henry Fuchs.

This article would not have been written except for the encouragement of Marceli Wein. Numerous discussions with Nick England and Mary Whitton have contributed to our appreciation of the Ikonas architecture and its versatility. Alain Brossard kindly provided the translation for the résumé. The research reported here was supported by the Natural Sciences and Engineering Research Council of Canada under a variety of grants.

REFERENCES

[1] Adage, "RDS 3000 Manual."

[2] R. M. Baecker, "Digital Video Display Systems and Dynamic Graphics," *Computer Graphics 17:3*, (August, 1979) pp. 48-56.

[3] L. Carpenter "The A-Buffer, An Antialiased Hidden Surface Method," *Computer Graphics 18:3*, (July, 1984) pp. 103-108.

[4] E. Catmull, *A Subdivision Algorithm for Computer Display of Curved Surfaces*, (doctoral dissertation) Technical Report UTEC-CSc-74-133, Department of Computer Science, University of Utah (December, 1974).

[5] E. Catmull, "Computer Display of Curved Surfaces," *Proceedings IEEE Conference on Computer Graphics, Pattern Recognition and Data Structures*, (May, 1975), reprinted in *Tutorial and Selected Readings in Interactive Computer Graphics*, H. Freeman (ed.), IEEE Computer Society (1980) pp. 309-315.

[6] K. D. Evans "An Approximate Method for Anti-Aliasing, Using a Random Access A-Buffer," *Proceedings Graphics Interface '84*, (May, 1984) p. 109.

[7] J. D. Foley and A. van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley (1982).

[8] E. Fiume, A. Fournier, and L. Rudolph, "A Parallel Scan Conversion Algorithm with Anti-Aliasing for a General-Purpose Ultracomputer," *Computer Graphics 17:3*, (July, 1983) pp. 141-150.

[9] S. A. Mackay and K. S. Booth, "Techniques for Frame Buffer Animation," *Proceedings Graphics Interface '82* (May, 1982) pp. 213-220.

[10] W. M. Newman and R. F. Sproull, *Principles of Interactive Computer Graphics*, second edition, McGraw-Hill (1979).

[11] I. E. Sutherland and G. W. Hodgman, "Reentrant Polygon Clipping," *Communications of the ACM, 17:1*, (January, 1974) pp. 32-42.

[12] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms," *Computer Surveys, 6:1* (March, 1974) pp.

[13] R. Swallow, personal communication.

[14] M. Whitton, "Memory Design for Raster Graphics Displays," *IEEE Computer Graphics and Applications, 4:3*, (March 1984) pp. 48-64.