# A Parallel Scan Conversion Algorithm with Anti-Aliasing for a General-Purpose Ultracomputer

Eugene Fiume
Alain Fournier

Computer Systems Research Group
Department of Computer Science
University of Toronto
Toronto, Ontario, M5S 1A4

Larry Rudolph

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

*ABSTRACT*

Popular approaches to speeding up scan conversion often employ parallel processing. Recently, several special-purpose parallel architectures have been suggested. We propose an alternative to these systems: the general-purpose ultracomputer, a parallel processor with many autonomous processing elements and a shared memory. The "serial semantics/parallel execution" feature of this architecture is exploited in the formulation of a scan conversion algorithm. Hidden surfaces are removed using a single scanline, z-buffer algorithm. Since exact anti-aliasing is inherently slow, a novel parallel anti-aliasing algorithm is presented in which subpixel coverage by edges is approximated using a look-up table. The ultimate intensity of a pixel is the weighted sum of the intensity contribution of the closest edge, that of the "losing" edges, and that of the background. The algorithm is fast and accurate, it is attractive even in a serial environment, and it avoids several artifacts that commonly occur in animated sequences.

CR Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles - *Shared Memory*; D.3.3 [**Programming Languages**]: Language Constructs - *Concurrent programming structures*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems - *Geometrical problems and computations*; I.3.1 [**Computer Graphics**]: Hardware Architecture - *Raster display devices*; I.3.3 [**Computer Graphics**]: Picture/Image Generation - *Display algorithms*; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism - *Visible line/surface algorithm.*



## 1. Introduction

The performance of a raster graphics system is strongly influenced by the inefficiency of scan conversion. Several recent papers have proposed speeding up scan conversion by employing special-purpose hardware. These systems exploit parallel processing in various ways, some of which are:

(1) "Intelligent" VLSI-based memory. This includes systems such as PIXEL-PLANES, by Fuchs *et al.* [FuPo81, FPPB82], the smart memory architecture by Gupta *et al.* [GuSS81], and the Rectangular Area Filling Display System Architecture by Whelan [Whel82].

(2) Hardware enhancements or graphics engines. Clark's geometry engine, although not a scan conversion system, illustrates the latter [Clar82], and Whitted's enhanced frame buffer is an example of the former [Whit81]. The proposed systems of Fussell and Rathi [FuRa82], and Weinberg [Wein81], are graphics engines.

(3) Special-purpose, multiple-processor systems. These systems incorporate special-purpose hardware to broadcast image descriptions to the processors. Image memory is often partitioned to enhance parallelism. Examples include Fuch's central broadcast controller [Fuch77], Parke's splitter tree, and Parke's splitter tree/broadcast controller hybrid [Park80].

Obviously, any parallel-processing scheme should demonstrably hasten scan conversion. The above proposals are no exception. Several issues remain open, however. First, few proposals address the aliasing problem. Indeed, anti-aliasing is difficult to perform on the systems of Fuchs *et al.*, Fussell and Rathi, Whelan, Fuchs, and Parke. Second, display systems exploiting parallelism should always exhibit subserial behaviour. Third, it is not clear that a special-purpose system is the best approach if similar computational power is required for other tasks. It is likely that the feasibility of large-scale display processors with special-purpose hardware will coincide with that of general-purpose parallel processors. The ultracomputer, described below, is one such processor. We wish to demonstrate that the ultracomputer can be a very effective "graphics engine" in its own right. This is illustrated by presenting a parallel scan conversion

algorithm including anti-aliasing. The worst case behaviour of the algorithm is subserial.

Not all problems necessarily have faster parallel implementations. However, problems such as scan conversion, which naturally decompose into a large set of independent subproblems, are good candidates for parallel processing. The objective of a general-purpose parallel processor design is to maximise the degree of subproblem independence over a wide class of tasks. Otherwise, the major advantage of such a processor over special-purpose systems is lost. In our ultracomputer model, subproblem independence is facilitated by a small repertoire of powerful concurrent operations on shared memory. To each processing element (PE) of the ultracomputer, a concurrent operation appears to execute indivisibly. In fact, an intelligent, multistage network cleverly connects the PEs to shared memory, and combines all operations simultaneously directed at a variable into one operation. Programs thus appear to have a serial semantics, but parallel execution. Moreover, parallel programs are simply expressed, unlike the often more complicated techniques required to optimise computations on vector or pipeline processors. Because of this "serial semantics/parallel exection" property, the algorithms below can be implemented on any processor capable of simulating the concurrent operations, although the resulting programs may run more slowly.

Section 2 outlines the basic ultracomputer architecture. A scan conversion algorithm that utilises this parallel processing model is presented in Section 3. A novel anti-aliasing algorithm is given as an integral part of scan conversion. Lastly, the generality of the ultracomputer is illustrated by noting other problems to which it can be applied. This is discussed in Section 4, as are topics for future research.

## 2. Ultracomputer Architecture

An *ultracomputer* is a parallel processor composed of many processing elements (PEs), which have multiple-cycle access to shared memory. Ultracomputers are a good model of parallel computation. Schwartz has made an extensive survey of this field, summarising various upper and lower bounds for parallel sorting algorithms, set operations, matrix multiplication, etc. [Schw80]. Ultracomputers are more than just a theoretical model, however. Indeed, our ultracomputer model is based on the work done at New York University, at which a large-scale implementation is planned [GGKM81]. The model we propose is a very slight extension of the NYU model, incorporating additional concurrent instructions.

An NYU Ultracomputer is composed of $N = 2^D$ autonomous PEs and connected to N shared memory modules. Local memory for each PE is provided by means of a partitioned memory cache. PEs access shared memory via a $D = \log_2 N$-stage connection network composed of an N×D array of "intelligent" 2-input, 2-output switches[1]. Switch interconnection is based on Lawries's omega-network [Lawr75], illustrated in Figure 1.

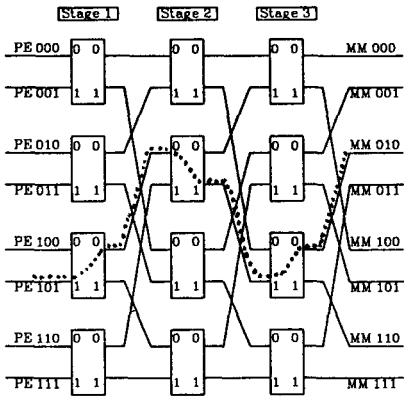



**Figure 1.** *Routing through an omega-network for 8 PEs.* Connections between PEs, switches, and MMs are by means of a *shuffle-exchange*: an object numbered $d_1 d_2 \cdots d_D$ in binary is connected to the object numbered $d_2 \cdots d_D d_1$ in the next stage of the network. A message transmitted from PE $p_D \cdots p_1$ to MM $m_D \cdots m_1$ uses output port $m_i$ when leaving the $i^{th}$ switch. Similarly for travelling from MM to PE. The route from PE 5 ($101_2$) to MM 2 ($010_2$) is indicated.

The novelty of the NYU design rests in the intelligent switches, which implement concurrent access to variables in shared memory. The network easily realises concurrent fetch or store operations. Other more powerful concurrent operations can be implemented. Presently, one such instruction is supported: the *replace-add*, which creates the illusion of indivisibly adding a value to a shared variable, and returning the sum to the requesting PE. Specifically, the format of the operation is **RepAdd**(V,e)[2], where V denotes a shared (integer) variable and e is an integer expression. Let V have value v. Suppose $PE_i$ issues the command $S_i$ =**RepAdd**(V,$e_i$), and $PE_j$ issues the command $S_j$ =**RepAdd**(V,$e_j$) simultaneously. Then, assuming V is not simultaneously updated by another PE, either

$$S_i = v + e_i$$
$$S_j = v + e_i + e_j,$$

or

$$S_i = v + e_j + e_i$$
$$S_j = v + e_j,$$

and in either case, the new value of V is $v+e_i+e_j$. Note that **RepAdd**(V,0) is a fetch instruction.

When operations on the same cell in shared memory meet at a switch, they are synthesised into a single instruction. This is sent to the next stage in the network in one cycle. Instruction combining can occur at any stage in the network. Hence of all the operations simultaneously directed at a single variable, V, only one cumulative operation actually "reaches" V. Thus memory traffic is reduced and network bandwidth is increased. Indeed, the processor has the following surprising property: it is

1. The entire architecture can be easily generalised to $N = k^D$ PEs and a $D = \log_k N$-stage network using k-input, k-output switches.

2. The semantics of this operation has recently been modified in the NYU design, and has since been renamed FetchAdd. Since RepAdd can be easily constructed from FetchAdd, we will continue to use RepAdd in our ultracomputer model.

particularly efficient when many operations are concurrently issued on a small set of variables. Simultaneous update of the same variable by all N PEs is resolved in O(log N) time, compared to O(N) time for typical parallel processors using semaphore-like mutual exclusion. This is a useful property which is often exploited. For example, RepAdd makes an effective synchronisation primitive [GoLR83]. Moreover, data structures allowing parallel access are conveniently implemented using RepAdd. A polygon display list can be nicely implemented as a parallel queue. Suppose the index NextPolygon is used as a subscript into a polygon list. Then every PE executing **RepAdd**(NextPolygon,1) is guaranteed to get a unique value for NextPolygon.

The standard NYU ultracomputer model supports the three concurrent instructions described above: fetch, store, and RepAdd. To realise these operations, a switch only needs a small amount of memory, and an adder. Implementation details, together with a network performance analysis, are found in [GGKM81]. Although these instructions have proved useful for constructing good parallel solutions to scientific and operating system problems, we believe a concurrent, flexible comparison instruction is needed. We propose a new concurrent instruction, *replace-minimum*, or RepMin, which is easily realised by adding a comparator to each switch. Its semantics is defined as follows. Let V denote a cell of shared memory having value v, and let e be an expression such that both v and e are pairs (intensity,depth) of values[3]. Then **RepMin**(V,e) causes all of V to be replaced by e iff e.depth < v.depth. The value returned by RepMin will be discussed presently. The utility of RepMin in scan conversion is obvious. Consider the following parallel version of the z-buffer algorithm found in [NeSp79]. Here, the entire z-buffer is assumed to be addressable as an n×m array of shared memory. Each PE executes the following.

```
while polygons remain do begin
   get P from polygon list (use RepAdd)
   ∀ pixels (x,y) ∈ P do begin
      i := PolygonIntensity(P,x,y)
      z := PolygonDepth(P,x,y)
      RepMin( (x,y), (i,z) )
   end
end
```

Let us now discuss the value returned by a RepMin operation. We only consider the case where n PEs ($0 \le n \le N$) simultaneously issue a RepMin for cell V. Informally, of all the RepMin's simultaneously directed at variable V, the value returned to a PE is one which has "lost" in at least one comparison. Moreover, any value sent by a particular PE is returned exactly once. Perhaps surprisingly, this is achievable in the switches, and can be shown by induction on n.

The NYU ultracomputer also presently lacks concurrent logical bit operations. The scan conversion algorithm below makes use of another concurrent instruction, the RepAnd. This operation has the same format as the RepAdd, but performs a logical *and* of the arguments instead of an addition. Note that in principle, only a few Nand gates in each switch would be required to realise all 16 boolean operations as concurrent instructions.

In general, an instruction supported by the connection network must be associative. Thus concurrent floating point operations *cannot* be properly realised[4]. There exist inherently non-associative operations, such as the group (or Fourier) commutator. Defined as $[a,b] = aba^{-1}b^{-1}$, this operation is not associative for non-commutative groups; thus a "RepCom" instruction for matrices under multiplication is inherently unrealisable.

The *serialisation principle* is a necessary property of the connection network: The network ensures that the *effect* of simultaneous operations by the PEs is equivalent to some serialisation of the operations.

## 3. A Fast Parallel Scan Conversion Algorithm

### 3.1. Preliminaries

Our definition of scan conversion is the traditional one (e.g. [NeSp79]). Given a scene represented by P simple polygons, determine the set of pixels and their intensities that best approximates the scene. The solution, based on the conventional single-scanline z-buffer algorithm, performs hidden-surface removal and anti-aliasing. Serial scanline algorithms typically require a YX-sort of polygon spans intersecting with a given scanline [SuSS74]. However, RepMin allows us to drop the X sort. The shared memory storing ultimate scanline intensities is assumed to be available to a video controller, by dual-ported memory, for instance.

### 3.2. The Algorithm

First we briefly outline the major steps performed by each PE. As in traditional scanline algorithms, a Y-scanline bucket is employed to determine polygon segments that enter the scene at scanline y.

(1) Remove backfacing polygons.

(2) Convert remaining polygons into sets of *span-areas*, i.e. trapezoidal or triangular regions. Insert each span-area into the Y-bucket corresponding to its largest y-value.

(3) Scan convert span-areas:
**for** y := ymin **to** ymax **do**

(a) The span-areas from bucket y are inserted into the *active span list* (ASL).

(b) Process active spans for scanline y. Each PE takes a span from the ASL. If the span is large, only a fraction of it is taken at a time, thus permitting parallel processing of the span. For each pixel in its portion of a span, the PE computes intensity and depth values, and performs a table look-up to approximate the portion of the pixel covered by the span. The left and right endpoints of the span are then updated. If the span-area is exhausted, it is removed from the ASL.

(c) Anti-aliasing. For each non-empty pixel, an approximate anti-aliasing procedure is performed by

3. To make the replace-minimum instruction quite general, the extent of the intensity and depth subwords could be controlled by a modifiable bit-mask stored in each switch. Clearly, the names of the subwords, "intensity" and "depth", are illustrative. In practice, the subwords could be known by arbitrary names.

4. In most computers, $((10^{\alpha}-10^{\alpha})+1) \neq (10^{\alpha}+(-10^{\alpha}+1))$, for large $\alpha$.

determining the intensity contribution of the closest span, and adding in the average contribution of the "losers". The coverage information computed in step (b) is used in these calculations.

### 3.2.1. Data Structures

For clarity, we only use static storage in shared memory. Assume there are P input polygons found in the array InputList. In what follows, let $V_i$ be the number of vertices in input polygon $P_i$, and let $V$ be the largest such $V_i$. Assume the PEs are programmed in a high-level language such as Pascal or Euclid which allows programmer-defined data types. Note that arrays in shared memory are possible, since their starting addresses can be stored in the local memory for each PE. The names assigned to variables in shared memory begin with an upper case letter. Local variables begin with a lower case letter.

```
{ Polygon display list }
 InputList: array 1..P of Polygon
 Ngon: array 1..P of Polygon
  - each polygon Pi contains an array 1..Vi of (x,y,z).

{ Y bucket. Yp gives next available position for scanline y }
 Y: matrix ymin..ymax 1..PV of SpanArea
 Yp: array ymin..ymax of 0..P

{ Active Span List. S reflects the number of spans. }
 ASL: array 1..PV of SpanArea
 S: 1..PV := 0

{ Some indices }
 PolyIn, PolyOut, CurrentSpan: Integer

{ Locks for synchronisation. Assume they are initialised to 0 }
 Lock1, Lock2: 0..P := 0

 SpanArea: type
  record of
     yt    { top y }
     dy    { height of span-area }
     xl    { current LHS }
     xr    { current RHS }
     xm    { multiplicity-see below; initially xm=xl-M}
     dxl   { Δx/Δy of LHS }
     dxr   { Δx/Δy of RHS }
     dyl   { Δy/Δx of LHS }
     dyr   { Δy/Δx of RHS }
     DepthInfo
     IntensityInfo
  end
```

### 3.2.2. Synchronisation, Initialisation, and Backfacing Polygon Removal

Since the code in this section is familiar, it is a good place to illustrate some principles of synchronisation and initialisation. Assume each PE has access to a unique identifier in the manifest constant PEid, which takes on a value between 1 and N. The following code initialises PolyIn and PolyOut, performs synchronisation, and removes backfacing polygons as in [NeSp79, Appendix III]. We assume the polygons in the input list have undergone perspective transformation. The reader may wish to verify that two locks are necessary to have fully reusable locks for synchronisation.

```
 i, j, p: integer
 InputList, Ngon, Lock1, Lock2, PolyIn, PolyOut: shared

{ The first PE in initialises PolyIn, PolyOut }
 if RepAdd(Lock1,1) = 1 then PolyIn := PolyOut := Lock2 := 0
 while Lock1 < N do {nothing}

{ The last PE out resets Lock1 for future use }
 if Lock2 = N-1 then Lock1 := 0
 RepAdd(Lock2,1)
 while Lock2 < N do {nothing}

 p := RepAdd(PolyOut,1)
 while p ≤ P do begin
    for polygon InputList[p], calculate c
    c := Σ_{i=1}^{V_p} (V[i].x-V[j].x)(V[i].y+V[j].y)
         where j=i+1 if i<V_p ; otherwise j=1
    if c ≤ 0 then {the polygon faces us, add it to Ngon}
       Ngon[RepAdd(PolyIn,1)] := InputList[p]
    p := RepAdd(PolyOut,1)
 end
```

In the average case, each PE processes about P/N polygons. This algorithm assumes that N<P, since otherwise those PEs with PEid>P do no work. The amount of memory traffic this algorithm would cause is suboptimal, since polygon definitions are moved around, rather than their pointers.

### 3.2.3. Decomposition of Polygons into Span-areas

As presented in this paper, the scan conversion algorithm presumes the input polygon list has been decomposed into span-areas: trapezoidal or triangular regions. This idea is not new (see [Lee81, Wein81, WhWe81]). Unlike polygons, span-areas have a bounded, concise specification in terms of left and right edges. Thus span-areas are useful in scanline-oriented algorithms. However, desirable properties of trapezoids such as planarity are not necessarily preserved after geometric transformations. Consequently, the input polygon list is preprocessed for each frame. This additional computation can be circumvented if polygons are triangulated once and for all, since triangles remain (trivially) planar after geometric transformations (see [FuRa82, Whit81]). The scan conversion algorithm easily adapts to triangles, but since span-areas are so simple to work with, the algorithm is presented using span-areas. Both triangles and span-areas can lead to fragmentation of very small (pixel-sized) polygons, making anti-aliasing critical.

A maximum of V-1 span-areas are generated for a polygon of V vertices. An O(V logV) serial algorithm to decompose a simple polygon into span-areas was recently published [Lee81]. A straightforward, polygon-per-PE parallelisation of this algorithm would yield an $O(\frac{PV}{N} \log V)$ average-case running time. As each span-area is generated, it is inserted into the Y-bucket corresponding to the largest y value of the span-area. This can be determined on-the-fly with no change in the order statistic.

### 3.2.4. Scan Conversion

Each PE performs the following scan conversion loop.

```
for y:=ymin to ymax do begin
    UpdateASL(y)
    InitialiseScanLine
    ScanConvert(y)
    <synchronise>
end for
```

UpdateASL places the contents of bucket Y[y] into the active span list. All PEs synchronise at the completion of scan conversion for each scanline. This is not necessary. If sufficient memory is available, the algorithm easily generalises to k-scanlines, k ≥ 1. We now consider the scan conversion process in more detail.

```
procedure InitialiseScanLine
  InitialiseXBucket
  CurrentSpan := 1
  <synchronise>
end InitialiseScanLine

procedure ScanConvert(y: ymin..ymax)
  span: SpanArea
  spansLeft: Boolean
  X: shared

  GetSpan(span,spansLeft)
  while spansLeft do begin
    ∀x ∈ span calculate pixelInfo:
       intensity, depth, and coverage mask
       UpdatePixel(x, pixelInfo)
    GetSpan(span,spansLeft)
  end while
  AntiAliasScanline(y)
end ScanConvert
```

The X bucket contains all required scanline information. It will be discussed shortly, as will the routines UpdatePixel and AntiAliasScanline.

GetSpan does the obvious: it returns an unprocessed span to the scan converter. However, the routine is complicated by the fact that we wish to get a subserial worst case behaviour. In particular, large spans should receive parallel treatment, for otherwise all PEs could wait for one PE to complete a long span. One approach is for PEs to recursively subdivide large spans so that each PE processes a smaller portion of the span. However elegant this solution appears, this approach is likely to increase memory traffic substantially. The approach we have taken avoids this problem. Assume there is a constant M which denotes the maximum number of pixels in a span that a PE is allowed to process at a time. This value may be empirically or theoretically determined, and represents a good balance between the overhead in GetSpan and the increased efficiency in parallel processing of large spans. Multiple copies of a span may be returned; the index xm is used to indicate the leftmost point of the unprocessed portion of the span[5]. The following is one possible implementation of GetSpan. It is somewhat tricky since it must cope with the unlikely event that two PEs simultaneously try to get an exhausted span.

```
procedure GetSpan(var span: SpanArea;
                  var spansLeft: Boolean)
  gotSpan: Boolean
  S, ASL, CurrentSpan: shared
  M: Constant
  newLHS: Integer

  spansLeft := CurrentSpan ≤ S
  gotSpan := false
  while ~gotSpan and spansLeft do begin
    span := ASL[CurrentSpan]
    with ASL[CurrentSpan] do begin
      { calculate new LHS of span, and see if LHS>RHS }
      newLHS := RepAdd(xm,M)
      gotSpan := newLHS ≤ xr
      if ~gotSpan then
        { if span is exhausted, the first PE advances CurrentSpan
            and processes the remaining span segment }
        if newLHS-xr ≤ M then
           RepAdd(CurrentSpan,1)
      spansLeft := CurrentSpan ≤ S
    end with/while
  if gotSpan then span.xm := newLHS
  spansLeft := gotSpan
end GetSpan
```

### 3.2.5. Anti-aliasing

The aliasing problem is immediately apparent to anyone who has seen synthesised raster images. Various aliasing artifacts are possible in both still and moving images. An abundant literature describes the problem and some of its solutions. See [Crow77, Crow81] for a start. It is thus of prime importance to examine whether anti-aliasing can be incorporated into our algorithm. Since we currently compute the picture scanline by scanline without backtracking over scanlines, we cannot use any scheme where the value at one pixel depends on the value of some of its neighbours, unless we arbitrarily privilege the $x$ direction[6].

The best solution under the circumstances is what we can call the Exact Area Sampling solution, where the intensity for the pixel is $I = \frac{1}{A}\sum_i I_i A_i$. $A_i$ and $I_i$ are the areas and intensities of the visible surfaces within the pixel, and $A$ is the total area of the pixel. If colour is used, this formula is used for the three primaries. As pointed out in [Catm78], and implemented there and in [FuBar79], this requires a hidden surface algorithm at the pixel level.

We can establish a more formal lower bound, by showing that any algorithm that computes the EAS can be used to determine the order of a list of $n$ non-negative integers. The reduction is as follows. Given a list $N_1, N_2, \cdots, N_n$ of numbers, construct a scene with $n$ rectangles of depth $N_i$, with the left, top and bottom edges coincident with the pixel left, top and bottom, and the right edge of rectangle $i$ at $N_i$. Without loss of generality, assume that the pixel right edge is at $\max(N_i)$. Let the intensity $I_i$ of each rectangle be $D^{i-1}$ where $D$ is a constant greater than $\max(N_i) - \min(N_i)$.

The answer to the EAS problem is then:

5. See the definition of the SpanArea data type above.

6. The idea is not totally without merit, since as seen on broadcast television it produces decent images. Note, moreover, that a k-scanline version (k > 1) of the algorithm would permit a multiple-pixel anti-aliasing scheme.

$H \times \sum_i D^{i-1}(N_i - N_p)$ where $H$ is the height of the pixel, and $N_p$ is the predecessor of $N_i$ in the sorted order. The predecessor of $\min(N_i)$ is 0. This transformation can be done in O(n) time. It is clear that the answer, when expressed as a base $D$ number, contains $N_i - N_p$ in the $i$ th digit (from the least significant), and that therefore in O(n) time one can find, for every number, its predecessor in the sorted order. Computing the answer to the EAS problem allows sorting with a O(n) time transformation, and therefore takes at least O(nlogn) time. While this does not prove that it is necessary to solve the hidden surface problem to solve the EAS problem, this shows that nothing easier than sorting will do it. For other results about the EAS, see [FoFu83].

In view of this result, we will aim for an approximate solution. Our approach will be to limit the amount of computation and to utilise parallelism as much as possible.

We subdivide the pixel into $n \times n$ subpixels. It is convenient to have $n$ a power of 2, for example $n = 2^3 = 8$. For each line which intersects a pixel, the two intersection points along the boundaries of the pixel are used as an index into a lookup table, whose entries give the subpixels covered by the halfplane defined by this line. We will call this entry the *mask* for this halfplane. In our example, the mask would be a 64 bit number. Each intersection with the boundaries of the pixel is computed with $k$ bits of fraction (where there are $2^k$ intervals, since the fraction n/n=1 in the current pixel is 0 on the next pixel). In our example, then, k=3. Thus each intersection can be fully described as a $k+2$ bit number, 2 bits to identify the boundary crossed, and $k$ bits to give the crossing position along the boundary (see Figure 2). The total entry for a line is then a $2(k+2)$ bit number, which in our example is a 10 bit number. This gives a $1K \times 64$ bit table, which is small enough to allow a copy for each PE. Alternatively, several PEs could directly share such a read-only table.

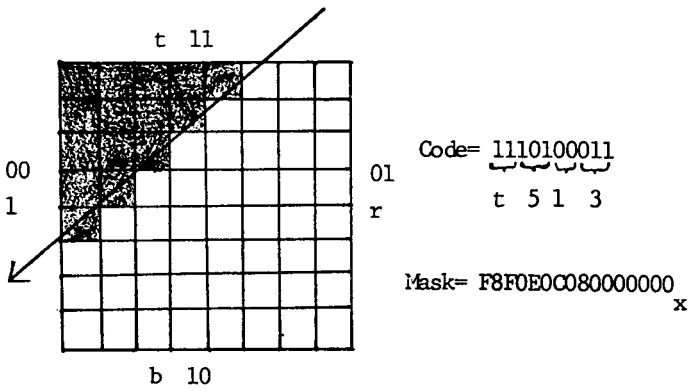



**Figure 2.** *Pixel-line intersection encoding.*

The order of the intersections is relevant, since the line should be oriented. We adopt a convention that the inside is to the right when going from the first intersection to the second. The size of the table can be reduced by making it into a triangular array, and using an extra bit to indicate the direction, which will tell whether to complement the mask or not. The table is of course precomputed, and each bit is on if the subpixel corresponding to it is more than half-covered by the halfplane described by the index.

Of the four edges of a normal span-area, two are horizontal, and are relevant only at the start and at the end of its scanning. For these, a small special lookup table can be used, with the $y$ fraction used as the index. For the other two edges, updating the intersection information from pixel to pixel is fairly simple, and requires only additions and subtractions.

The coverage mask has interesting boolean properties. Indeed, the mask for the intersection of a span-area with a pixel is the *and* of the masks of the span-area's edges which cross the pixel. Thus we get an accurate representation of the subpixels covered by a given span-area. It is also easily seen that the mask for the background (indicating the subpixels where the background is seen) is the complement of the *or* of all the span-area masks for this pixel. It is unfortunately impossible to go much farther without making some approximations. The problem is that we do not want to compute the Z values at the subpixel resolution, since it would be tantamount to going to a higher resolution. Each span-area at a given pixel is then associated with only one Z value, namely its Z at the centre of the pixel. Given that, we cannot guarantee that the depth comparison allows the visible areas to be exactly determined, unless the planes of support of the span-areas do not intersect within the pixel (see Figure 3). We will give two approximation algorithms, and discuss where they succeed, and where they fail. Let *weight* (*mask*) be the fraction of the pixel covered by a mask (this can be easily computed by counting the number of one bits in the mask). The span-area with the smallest Z value is called the *winner;* the others are called *losers*.

There are two ways to compute the final pixel intensity. One way necessitates the use of an X-bucket to hold pixel information for each span-area intersecting with the current scanline; a pass over the content of this bucket would be performed at the end of the scanline, since the final intensity cannot be computed until the winner is known. The other approximation can be computed on-the-fly, and is almost as accurate as the first. The two methods calculate intensities $I_1$ and $I_2$, respectively, as follows.

$$I_1 = WinnerComp + LoserComp_1 + BackgroundComp$$
$$I_2 = WinnerComp + LoserComp_2 + BackgroundComp$$

$$WinnerComp = I_w \times weight(mask_w)$$
$$BackgroundComp = I_b \times weight(\wedge AllMasks)$$
$$LoserComp_1 = Cor_1 \times \sum_{all\ i} I_i \times weight(mask_i \wedge mask_w)$$
$$LoserComp_2 = Cor_2 \times weight(\overline{mask_w}) \times \sum_{all\ i} I_i \times weight(mask_i)$$
$$Cor_1 = \frac{weight(\overline{mask_b} \wedge \overline{mask_w})}{\sum_{all\ i} weight(mask_i \wedge \overline{mask_w})}$$
$$Cor_2 = \frac{weight(\overline{mask_b} \wedge \overline{mask_w})}{\sum_{all\ i} weight(mask_i)}$$

The subscripts $w$, $l$, and $b$, stand for "winner", "loser", and "background", respectively. The correction factors are ratios of the actual coverage by the losers compared to the sum of their individual coverage as computed by each algorithm. Therefore the correction factors give a measure of the amount of overlap of the losers, hence of the possible error.

### 3.2.5.1. First approximate anti-aliasing algorithm

This solution requires an X bucket. For each pixel, several additional pieces of information are kept: the current winner, background data, the losers' intensity, and their sum of coverage-mask weights. The following data structures are used.

```
{ X bucket. Xp contains list of number of span-areas per pixel }
 X: matrix xmin..xmax 1..PV of PixelInfo
 Xp: array xmin..xmax of 0..PV := 0

{ Additional pixel information }
 Pixels: array xmin..xmax of
           Winner, Back: PixelInfo
           LoserInt, SumOfWeights: Integer

 PixelInfo: type record of
      Depth
      Int          { intensity }
      Mask         { coverage mask }
 end PixelInfo
```

The ScanConvert routine above executes the following version of UpdatePixel and AntiAliasScanline. Recall that each PE executes ScanConvert.

```
procedure UpdatePixel(x: xmin..xmax, pix: PixelInfo)
   { Add pixel from this span into bucket }
   X[x,RepAdd(Xp[x],1)] := pix
   { pix may be a "winner" }
   RepMin(Pixels[x].Winner,pix)
   { Determine how much background is covered by pix }
   RepAnd(Pixels[x].Back.Mask, pix.Mask)
end UpdatePixel
```

```
procedure AntiAliasScanline(y: ymin..ymax)
   x: Integer
   winner,pix: PixelInfo

 Initialise Cx to xmin
 while Cx ≤ xmax do begin
    { Many PEs work on each pixel (i.e. X bucket) }
    x := RepAdd(Xp[Cx],-1) + 1        { get pixel info for span }
    winner := Pixels[Cx].Winner
    while x > 0 do begin
      pix := X[Cx,x]
      if pix ≠ winner then begin
        { pix is a loser, calculate its contribution }
        newMask := pix.Mask ∧ winner.Mask
        newInt := Weight(newMask) × pix.Int
        RepAdd(Pixels[Cx].LoserInt, newInt)
        RepAdd(Pixels[Cx].SumOfWeights, Weight(newMask))
      end if
      x := RepAdd(Xp[Cx], -1) + 1
    end while
    if x = 0 then begin
      { PE with x=0 adds background and losers' contrib }
      for Pixels[Cx], compute:
      c := Weight(Back.Mask∧Winner.Mask)/SumOfWeights
      RepAdd(Winner.Int, c×LoserInt + Back.Int)
      RepAdd(Cx,1)
     end then
     else <synchronise> {all other PEs wait}
  end while
end AntiAliasScanline
```

### 3.2.5.2. Second approximate anti-aliasing algorithm

No X bucket is required in this solution. We only keep four pieces of information for each pixel, Winner, Back, SumOfWeights, and Losers. Winner, Back, and SumOfWeights are as in the first solution; Losers is used to keep track of the losers' coverage and intensity contributions on-the-fly.

```
procedure UpdatePixel(x: xmin..xmax, pix: PixelInfo)
   loser: PixelInfo

   loser := RepMin(Pixels[x].Winner, pix)
   intContrib := loser.Int × Weight(loser.Mask)
   RepAdd(Pixels[x].Losers.Int, intContrib)
   RepAnd(Pixels[x].Back.Mask, loser.Mask)
   RepAdd(Pixels[x].SumOfWeights, Weight(pix.Mask))
end UpdatePixel
```

```
procedure AntiAliasScanline(y: ymin..ymax)
   { each PE handles a pixel, so if N > X, some PEs are idle }
   x := PEid + xmin - 1
   while x < xmax do begin
     pix := Pixels[x]
     { compute background and losers' intensity contrib }
     backInt := pix.Back.Int × Weight(pix.Back.Mask)
     c := Weight(pix.Back.Mask∧pix.Winner.Mask)/pix.SumOfWeights
     loserInt := pix.Losers.Int × Weight(pix.Winner.mask) × c
     RepAdd(Pixels[x].Winner.Int, backInt+loserInt)
     x := x + N
   end while
end AntiAliasScanline
```

### 3.2.5.3. Analysis of the approximations

These approximations, and indeed all approximations of this kind, should be characterised in three ways: when they are right (here right is to be understood exact within the subpixel resolution), when they are wrong and how wrong they can be, and when they are *consistently* wrong. The last is important, since aliasing is particularly noticeable

in motion, by *crawling*, *scintillation* and other annoying artifacts. If an algorithm computes a wrong shade, but is consistent as the polygons move, then these artifacts will be avoided.

Both solutions will be right when there is only one span-area within the pixel, whether it covers the whole pixel or not. As long as a span-area covers at least one subpixel (1/64 of a pixel in our example), it will contribute to the total intensity of the pixel. Both solutions are also right when none of the span-areas overlap. This is especially important, since we might have cut a polygon into numerous small span-areas. Fortunately we will not have to pay a heavy price in aliasing problems. In fact, the problems, if any, will be at the silhouette edges of the objects, and not against the background, but against each other. The first solution has the additional advantage of being right when the winner overlaps the losers, but the losers do not overlap each other. The second algorithm will be right in case of overlap by the winner if the loser coverage ratio is sensibly the same under the winner as in the rest of the pixel.

Figures 3 and 4 give examples of wrong cases, and the errors made by each algorithm. Figure 3 shows the worst case for both algorithms, where the amount of overlap of the losers and the area they cover is maximal. Figure 4 shows a case where the first algorithm is right and the second is wrong.

**Figure 3.** *The worst case for both algorithms.* Span 1, the winner, covers a sliver of the pixel. Losing span 2 obscures another loser, 3.

$Z_1 < Z_2 < Z_3$

Correct Answer $\simeq I_2$

Computed Answer = $\frac{I_2 + I_3}{2}$

Error = $\frac{I_2 - I_3}{2}$

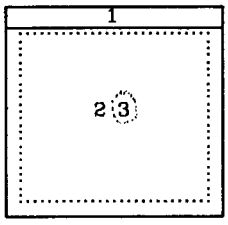



**Figure 4.** *A bad case for algorithm 2 only.* Winning span 1, covering half the pixel, obscures losing span 3. Losing span 2 covers half the pixel as well.

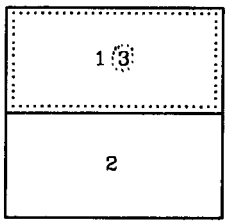
$Z_1 < Z_2 < Z_3$

Correct Answer = $\frac{I_1 + I_2}{2}$

Algorithm 1 = $\frac{I_1 + I_2}{2}$

Algorithm 2 = $\frac{I_1 + \frac{I_2 + I_3}{2}}{2}$

Error 2 = $\frac{I_2 - I_3}{4}$




A gross estimate of the extent of the errors for $10^3$ polygons, covering an average of $10^3$ pixels each, and with $10^2$ boundary pixels each, on a screen with $10^6$ pixels, shows that less than 5% of the pixels would have an error, and that for these the average error would be less than 10% of the shade of the pixel.

As the polygons move with respect to each other, we avoid the numerous problems of point sampling. Since the wrong cases are computed from averages, the errors made will not exhibit large discontinuities, but will be consistent from frame to frame. In the example of Figure 3, as polygon 3 moves out of the pixel, its contribution to the pixel intensity will go smoothly from $\frac{I_3}{4}$ (which is wrong), to 0 (which is right).

### 3.2.6. Discussion

An implementation of this algorithm has been made, demonstrating that the approach works, and illustrating a realisation in a pseudo-concurrent language. The implementation is written in Concurrent Euclid, a language developed at the University of Toronto which supports processes and monitors. Concurrent operations such as RepAdd are simulated using monitors. The only relevant aspect in which our implementation differs from one on an ultracomputer is speed. The lack of "true" concurrency, and the O(N) performance of concurrent operations (compared to O(logN) on an ultracomputer), make our implementation somewhat slower than would be expected on an ultracomputer.

The algorithm above has several appealing properties. It is independent of N, the number of PEs in the ultracomputer. Indeed, the speed of the algorithm is inversely proportional to N, up to a lower bound constant when $N \geq \frac{XY}{M}$. A good serial algorithm is obtained when N=1. We emphasise the fact that the anti-aliasing techniques presented here easily transfer to serial environments, as illustrated here. Another property of the algorithm is that although it scan converts polygons, the general approach adapts to other scene representations (e.g. scanline methods for parametric surfaces as in [BCLW80]).

Several improvements could be made to the algorithm. An issue deserving of attention is space complexity and memory traffic. By using dynamically allocated shared memory and pointers, the amount of storage required would be drastically reduced; memory traffic would decrease, since pointers would travel through shared memory. However, indirect shared memory references require two passes through the connection network. A solution is to make greater use of the cache memory local to each PE. A copy of the static pointers may be placed in the local memory for each PE, thus saving the O(log N) connection network cycle time.

## 4. Other Ultracomputer Applications and Future Research

As the plethora of published parallel algorithms shows [Schw80], the ultracomputer is truly a powerful, general-purpose tool. Fast parallel algorithms exist for matrix multiplication, sorting, linear programming, fluid dynamics, etc. We hope to have demonstrated that the ultracomputer has great

potential in the computer graphics field. Other applications would also significantly benefit from ultracomputer implementation. For instance, a parallel queue could be exploited to parallelise ray-tracing algorithms [Whit80]. Since the processing of one ray (or alternatively, one pixel) is an independent task, we believe significant speed-up in ray-tracing can be achieved on an ultracomputer. Similarly, we believe many problems in image processing, signal processing, and artificial intelligence are likely to benefit.

## 5. Acknowledgements

We wish to thank John Amanatides and Peter Schoeler for their suggestions, which have improved the clarity of this paper. The first two authors gratefully acknowledge the financial support of the Natural Sciences and Engineering Research Council of Canada.

## References

**BCLW80** Blinn, J.F., L.C. Carpenter, J.M. Lane, and T. Whitted, "Scan line methods for displaying parametrically defined surfaces", *Comm. ACM 23*, 1 (Jan. 1980), 23-34.

**Catm78** Catmull, E., "A Hidden-Surface Algorithm with Anti-Aliasing", *Computer Graphics (ACM), 12*, 3, (Aug. 78), 6-11.

**Clar82** Clark, J.H., "The geometry engine: a VLSI geometry system for graphics", *Computer Graphics (ACM) 16*, 3 (July 1982), 127-134.

**Crow77** Crow, F.C., "The Aliasing Problem in Computer-Generated Shaded Images", *Comm. ACM 20*, 11 (Nov. 1977), 799-805.

**Crow81** Crow, F.C., "A Comparison of Antialiasing Techniques", *IEEE Computer Graphics and Applications, 1*, 1 (Jan. 81), 40-49.

**FPPB82** Fuchs, H., J. Poulton, A. Paeth, and A. Bell, "Developing PIXEL-PLANES, a smart memory-based raster graphics system", *1982 Conference on Advanced Research in VLSI, MIT*, January 1982, 137-146.

**FoFu83** Fournier, A. and D. Fussell, "On the Power of the Frame Buffer", unpublished manuscript, 1983.

**FuBar79** Fuchs, H. and J. Barros, "Efficient Generation of Smooth Line Drawings on Video Displays", *Computer Graphics, 13*, 2, (Aug. 79), 260-269.

**FuPo81** Fuchs, H., and J. Poulton, "PIXEL-PLANES: a VLSI-oriented design for 3-D raster graphics", *CMCCS Conference Proceedings*, (June 1981), 343-348.

**FuRa82** Fussell, D., and B.D. Rathi, "A VLSI-oriented architecture for real-time raster display of shaded polygons", *Graphics Interface '82*, May 1982, 373-380.

**Fuch77** Fuchs, H., "Distributing a visible surface algorithm over multiple processors", *Proceedings of ACM 1977*, Seattle (Oct. 1977), 449-451.

**GGKM83** Gottlieb, A., R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer--designing an MIMD shared memory parallel computer", *IEEE Transactions on Computers, C-32*, 2 (Feb. 1983), 175-189.

**GoLR83** Gottlieb, A., B.D. Lubachevsky, and L. Rudolph, "Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors", *Transactions on Programming Languages and Systems (ACM) 5*, 2 (Apr. 1983), 164-189.

**GuSS81** Gupta, S., R.F. Sproull, and I.E. Sutherland, "A VSLI architecture for updating raster scan displays", *Computer Graphics (ACM) 15*, 3 (Aug. 1981), 71-78.

**Lawr75** Lawrie, D.H., "Access and alignment of data in an array processor", *IEEE Transactions on Computers, C-24*, 12 (Dec. 1975), 1145-1155.

**Lee81** Lee, D.T., "Shading of regions on vector display devices", *Computer Graphics (ACM) 15*, 3 (Aug. 1981), 37-44.

**NeSp79** Newman, W.M., and R.F. Sproull, *Principles of Interactive Computer Graphics*, Second Edition, McGraw-Hill, New York, 1979.

**Park80** Parke, F.I., "Simulation and expected performance of multiple processor z-buffer systems", *Computer Graphics (ACM) 14*, 3 (July 1980), 48-56.

**Schw80** Schwartz, J.T., "Ultracomputers", *Transactions on Programming Languages and Systems (ACM) 2*, 4 (Oct. 1980), 484-522.

**SuSS74** Sutherland, I.E., R.F. Sproull, and R.A. Schumacker, "A characterization of ten hidden-surface algorithms", *Computing Surveys (ACM) 6*, 1 (March 1974), 1-55.

**Wein81** Weinberg, R., "Parallel processing image synthesis and anti-aliasing", *Computer Graphics (ACM) 15*, 3 (Aug. 1981), 53-62.

**WhWe81** Whitted, T., and D.M. Weimer, "A software test-bed for the development of 3-D raster graphics systems", *Computer Graphics (ACM) 15*, 3 (Aug. 1981), 271-277.

**Whel82** Whelan, D.S., "A rectangular area filling display system architecture", *Computer Graphics (ACM) 16*, 3 (July 1982), 147-153.

**Whit80** Whitted, T., "An improved illumination model for shaded display", *Comm. ACM 23*, 6 (June 1980), 343-349.

**Whit81** Whitted, T., "Hardware enhanced 3-D raster display systems", *CMCCS Conference Proceedings*, (June 1981), 349-356.

**Figure 5.** *Aliased cube. Enlarged view at 256×256 resolution.*

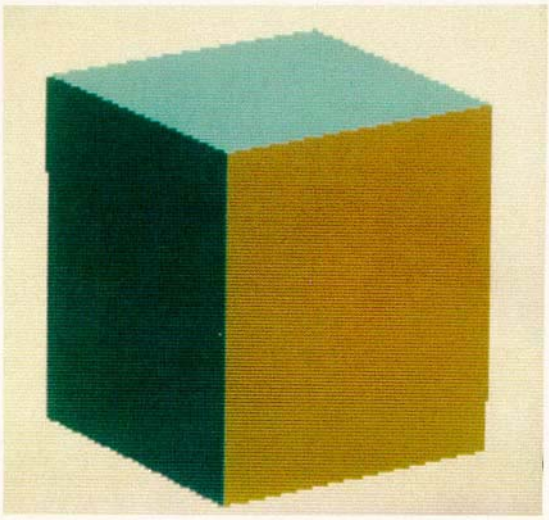


**Figure 6.** *Anti-aliased cube.*

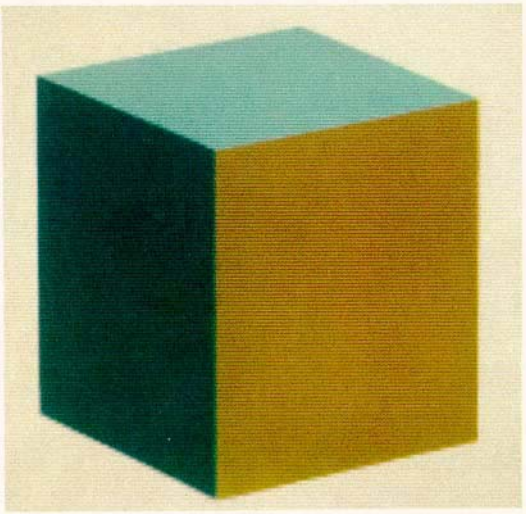


**Figure 7.** *Aliased Italian tablecloth.*

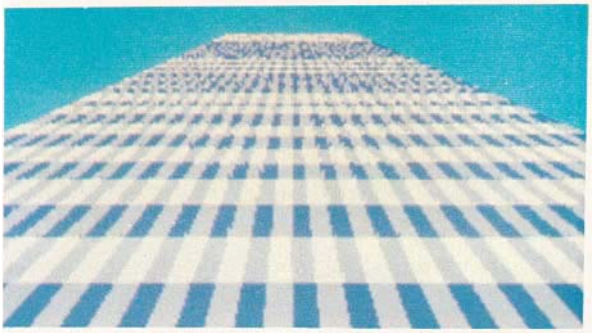


**Figure 8.** *Anti-aliased Italian tablecloth.*

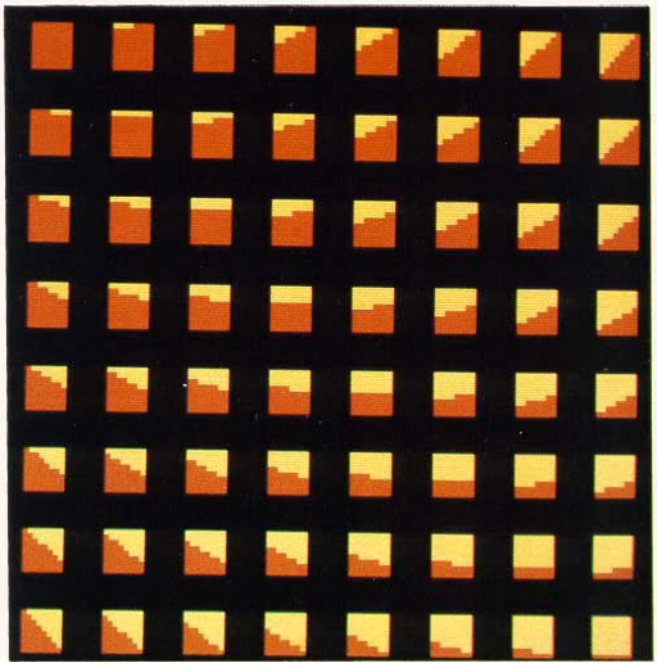


**Figure 9.** *Close-up of some coverage masks.* Each yellow area indicates the subpixel coverage of an oriented line.