Tiled Polygon Traversal Using Half-Plane Edge Functions

Joel McCormack*, Robert McNamara†

Compaq Computer Corporation

Abstract

Existing techniques for traversing a polygon generate fragments one (or more) rows or columns at a time. (A fragment is all the information needed to paint one pixel of the polygon.) This order is non-optimal for many operations. For example, most frame buffers are tiled into rectangular pages, and there is a cost associated with accessing a different page. Pixel processing is more efficient if all fragments of a polygon on one page are generated before any fragments on a different page. Similarly, texture caches have reduced miss rates if fragments are generated in tiles (and even tiles of tiles) whose size depends upon the cache organization.

We describe a polygon traversal algorithm that generates fragments in a tiled fashion. That is, it generates all fragments of a polygon within a rectangle (tile) before generating any fragments in another rectangle. For a single level of tiling, our algorithm requires one additional saved context (the values of all interpolator accumulators, such as *Z* depth, *Red*, *Green*, *Blue*, etc.) over a traditional traversal algorithm based upon half-plane edge functions. An additional level of tiling requires another saved context for the special case of rectangle copies, or three more for the general case. We describe how to use this algorithm to generate fragments in an optimal order for several common scenarios.

CR Categories and Subject Descriptors: I.3.1 [Computer Graphics]: Hardware Architecture – Graphics processors; I.3.3 [Computer Graphics]: Picture/Image Generation – Line and curve generation

Additional Keywords: rasterization, tiling, graphics accelerators

1. INTRODUCTION

In the rasterization stage of a graphics pipeline, a fragment must be generated for each pixel position within a polygonal object. A fragment contains all the information required to paint the surface at the pixel position, such as color, Z depth, texture coordinates, etc. Existing object traversal algorithms fall mainly into two categories: scanline based, and half-plane function based.

Scanline traversal is intuitively obvious. In its simplest form: (1) start at the top scanline contained within the polygon, (2) visit each pixel position on the scanline within the polygon from left to right, (3) repeat from top to bottom for each scanline in the polygon. But scanline traversal has several problems. First, the inverse slope of each polygon edge must be computed, requiring a divide operation per edge. Second, it is hard to generate several fragments in parallel with any degree of efficiency. Finally, it is hard to generate fragments that are sampled at several points for supersampled antialiased rendering.

Consequently, many graphics accelerators [1][7][8][10] rasterize polygons using half-plane edge functions [3][7][11]. Each (directed) edge of the polygon is described by a function which separates the (x, y) screen space plane into points to the left, on, or to the right of the edge. A point is within the polygon if it is on the same side of each directed edge. Starting near one vertex of the polygon, edge information from several points near the current position are combined to determine the next position to visit.

Such rasterizers address many of the problems of scanline generators. Setup of the edge functions requires no divides. Fragment generation is easily parallelized by using a $2^w \times 2^h$ pixel fragment "stamp," and simultaneously evaluating the edge functions and the color, *Z* depth, etc. interpolators at each pixel position within the stamp. Supersample antialiasing is accommodated by evaluating the edge functions at each of the many sample points belonging to each pixel position's filter.

However, published algorithms traverse objects in an order similar to scanline algorithms. This is non-optimal for fragment processing operations further down the pipeline. Several studies of texture caches [4][5][6] show the benefits of using a tiled rasterization order, in which the screen is tiled into rectangles that are related to the size of the texture cache. All fragments within one tile are generated before any fragments within another tile. Similarly, most frame buffers physically tile the screen into rectangular pages, and tiled rasterization that respects these page boundaries allows for fewer page crossings that are more efficiently prefetched [8].

In this paper, we first review half-plane edge functions and existing traversal algorithms based upon them. We then show how a different algorithm appears to have similar characteristics, but which enables tiled rasterization at little extra cost. Such tiling requires one new saved context and minimal changes to the logic that decides where to move next. We show how this tiling algorithm can be extended for other scenarios, such as copying data, where both the source and destination have tiling boundaries that might be respected for optimum performance.

2. HALF-PLANE EDGE FUNCTIONS

The three directed edges of a triangle, or the four edges of a line, can be represented by planar (affine) functions. Each edge function is negative for points to the left of the edge, positive for points to the right, and zero for points on the edge. A fragment is inside an object if all edges in a clockwise series are non-negative, or if all edges in a counterclockwise series are negative. (We

* Compaq Computer Corporation Western Research Laboratory, 250 University Avenue, Palo Alto, CA 94301. Joel.McCormack@Compaq.com.

† Compaq Computer Corporation Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301. Bob.McNamara@Compaq.com.

HWWS 2000, Interlaken, Switzerland

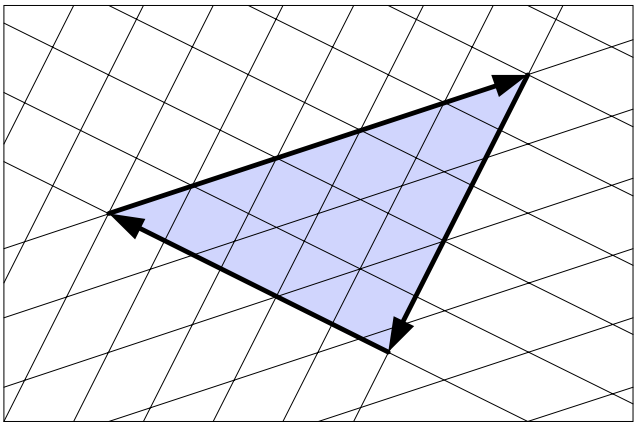

Figure 1: A triangle described by three edge functions.

slightly adjust an edge function's initial value in order to assign points along a shared edge or vertex to exactly one object.)

Figure 1 shows a triangle described by three clockwise edges, which are shown with bold arrows. The half-plane where each edge function is positive is shown by several thin lines parallel to the edge. The shaded portion shows the area where all edge functions are non-negative.

An edge function is simple to set up. The edge from (x_0, y_0) to (x_1, y_1) is described by the edge function E:

$$\Delta x = (x_1 - x_0)$$
$$\Delta y = (y_1 - y_0)$$
$$E(x, y) = (x - x_0)\,\Delta y - (y - y_0)\,\Delta x$$

The Pixel-Planes system [3] computes this function in parallel for all pixels in the frame buffer. Pineda [11] observed that an edge function is easy to incrementally update. For example, here are the four Manhattan neighbors:

$$E(x+1, y) = E(x, y) + \Delta y$$
$$E(x-1, y) = E(x, y) - \Delta y$$
$$E(x, y+1) = E(x, y) - \Delta x$$
$$E(x, y-1) = E(x, y) + \Delta x$$

This property is well suited for a more conventional rasterizer, which steps through a polygon, generating one or a group of fragments at each step.

3. WHAT POSITIONS INTERSECT THE OBJECT?

Before constructing algorithms that traverse a convex polygonal object, we must determine what moves from a given position are possible. That is, given a fragment stamp that is 2^w pixels wide, and 2^h pixels high, at a position that intersects an object, what nearby positions also intersect the object?

We limit ourselves to Manhattan moves—positions 2^h pixels up or down, or 2^w pixels left or right, from the current position. This limitation means that we may move to non-productive positions that are known to generate no fragments, especially when traversing thin diagonal objects. We could reduce the frequency of non-productive moves by allowing moves to diagonally adjacent positions, or even to non-adjacent positions. However, such algorithms involve substantially more complex decision-making logic, speculative computations, and multiplexing. Our experience with the Neon graphics accelerator [8], which uses a single sample point per pixel, convinced us that the increased cycle time outweighed any advantages. The possible advantages are even smaller if supersample antialiasing is used, as such non-productive moves become extremely rare. Almost any stamp position that intersects the object also has at least one supersample point inside the object.

Figure 2 shows a fragment stamp that is 4 pixels wide by 2 pixels high. The thin lines are a grid of pixels. The stamp boundaries are shown with thick solid lines. The object's edge functions are evaluated at eleven points. The circled stamp *origin* is both a *sample point* to determine if the upper left pixel is in the object, and a *probe point* to assist stamp movement. The other seven *sample points*, shown with an ×, determine if the associated pixels are in the object.

Three additional *probe points*, enclosed in diamonds, assist stamp movement. These probes are labeled *RT* (right top), *RB* (right bottom), and *LB* (left bottom). The stamp edge segments are defined as (*ORIGIN*, *RT*), (*RT*, *RB*), (*RB*, *LB*), and (*LB*, *ORIGIN*).

The sample and probe points are located on the corner of the pixels to simplify their computation: OpenGL semantics are accommodated by using a constant half-pixel x and y offset on all object vertex coordinates, which effectively puts the sample points at pixel centers.

The movement algorithm tests each edge segment of the stamp to see if it *intersects* the object, that is, if any point along the stamp edge is inside the object.

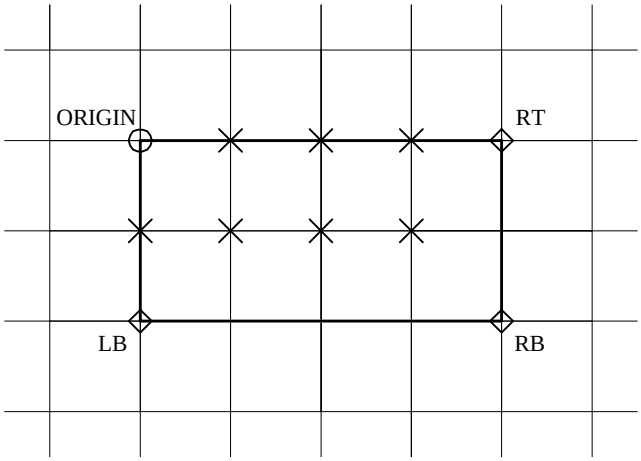

Figure 2: A 4 x 2 fragment stamp with three probe points (diamonds), seven sample points (×'s), and an origin (circle).

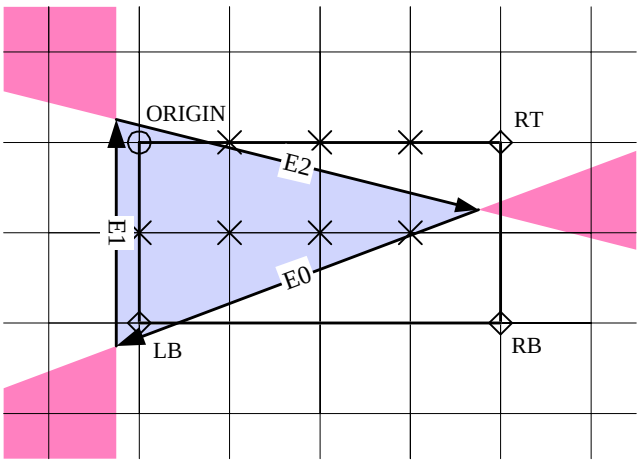

Figure 3: A triangle intersects the stamp's top, left, and bottom edge segments, while a shadow intersects the right edge segment.

Computing the intersection of a stamp edge segment with the object requires two tests. The first test computes if, for each of the half-plane functions surrounding the object, one or both of the probes at the ends of the stamp edge segment are on the inside of the half-plane equation. Note that this does not require that the *same* probe be inside each half-plane equation. Figure 3 shows the 4 x 2 stamp with a lightly shaded lavender triangle that intersects all but the right stamp edge segment. For example, the triangle intersects the top stamp edge (*ORIGIN*, *RT*) because *ORIGIN* and *RT* are inside *E0* and *E1*, and *ORIGIN* is inside *E2*. It is easy to see that this test will be true if the stamp edge intersects the object.

However, this doesn't quite implement the desired intersection semantics, as this test is also true if a stamp edge segment has both probes outside one of the object's *shadows*, but the segment spans the shadow. The shadows are areas that are outside two edges, but inside the remaining edge(s). The darker rose portions of Figure 3 show the three shadows of the triangle. Note how the right edge segment (*RT*, *RB*) satisfies the first intersection test: *RT* is inside *E0*, both *RT* and *RB* are inside *E1*, and *RB* is inside *E2*. However, the right edge does not truly intersect the triangle.

Thus, we add a second test that ensures that the stamp edge segment is inside the minimal rectangular bounding box of the object, where the bounding box's edges are horizontal and vertical. If both these tests are true, then the stamp edge segment *probably* intersects the object. Vertices that do not lie on the bounding box can still cast a deceitful shadow. This does not cause any correctness problems, but only efficiency problems. And these efficiency problems are small: such a vertex must be at an obtuse angle, and so the shadow grows quickly beyond the vertex. Within one or two stamp positions outside the object, the shadow is so large that one or both ends of the stamp edge segment fall into the shadow, and the intersection test is no longer fooled.

A stamp position that meets both intersection tests is called a *valid* position. The position *left* is valid if the left edge segment (*LB*, *ORIGIN*) intersects the object, etc.

This simple scheme for determining valid positions includes many positions that might be rejected easily, as these positions obviously contain no sample points within the object. We have versions of all the traversal algorithms described below that avoid moving to such "sliver" positions. However, we do not describe these algorithms due to space constraints, as avoiding slivers involves some subtle special cases. And this complexity is only worth implementing when there is a single sample point per pixel; sliver avoidance confers an almost negligible performance advantage to supersampled antialiasing. For more details, please see [9].

4. THE CENTERLINE TRAVERSAL ALGORITHM

We now describe a variant of the PixelVision [7] "centerline" traversal algorithm. This algorithm always starts with the top-most vertex, sweeping out an entire horizontal "stampline" before moving down to the next stampline. (A stampline here refers to a row of pixels the height of the stamp; in general, it may also refer to a column of pixels the width of the stamp.) It sweeps out each stampline by first noting if the *right* position is valid. If so, it saves the values of all of the edge functions and all other interpolated values, evaluated at point *RT*, into the context *rightSave*. It then traverses left across the stampline, until the *left* position is no longer valid. To visit the remainder of the stampline (if any), it examines the *rightSave* context to see if it is valid. If so, it restores the context *rightSave* (restoring a context also invalidates it), and traverses right across the stampline until the *right* position is no longer valid.

As it sweeps left, then right, across the stampline, the algorithm also looks for valid *down* positions. The first such position found on the stampline is saved into the *downSave* context. When the current stampline is complete, the algorithm moves down to the next stampline by restoring the *downSave* context. If *downSave* is invalid, rasterization of the object is complete. For most triangles, the algorithm steps down the first few stamplines in a vertical line, hence the name "centerline." When it hits the bottom edge of the triangle, it veers off this centerline in order to step down to the left-most or right-most position in each stampline.

Figure 4 shows this algorithm traversing a triangle. The octagon positions are stored in *downSave*; the circle positions are stored in *rightSave*. In this and all algorithms described below, saved contexts are bypassed appropriately. If *left* is not valid from the first position on a stampline, for example, the stamp immediately moves *right*, rather than taking one cycle to store the *right* position into *rightSave*, and another cycle to restore *rightSave*. And if both *left* and *right* are not valid, the stamp immediately moves *down*. When a position bypasses the corresponding saved context, the context indicator (octagon or circle here) is shown using dashed lines.

This algorithm requires two saved contexts, in addition to the current context. A context includes not only the values of the edge functions, but also all of the current color, *Z*, and texture coefficient values being interpolated from values provided at the vertices. (A context saves only the accumulator values for each type of data being interpolated, not the corresponding values used to increment or decrement the accumulators, as these are constant across the object.) A context involves a large amount of data, so we wish to minimize the number of saved contexts.

To allow the stamp to immediately move to the next position without stalling, the edge functions must be evaluated each cycle at the origin of the three possible next adjacent positions—*left*, *right*, and *down*. They are already evaluated at the origin of the *right* position (via probe *RT*) and the *down* position (via probe *LB*), and so must also be evaluated for the *left* position. This speculative evaluation applies only to the edge functions, as the other interpolated values like color, *Z*, and texture coordinates can lag the stamp by one cycle. This allows us to compute exactly one set of new interpolated values—the ones we actually end up using—each cycle. This is a bit tricky when saved contexts are involved—how can we evaluate a single set of interpolated values at the first position on a stampline, in which both *right* and *down* are valid and must be saved? We address this problem by storing

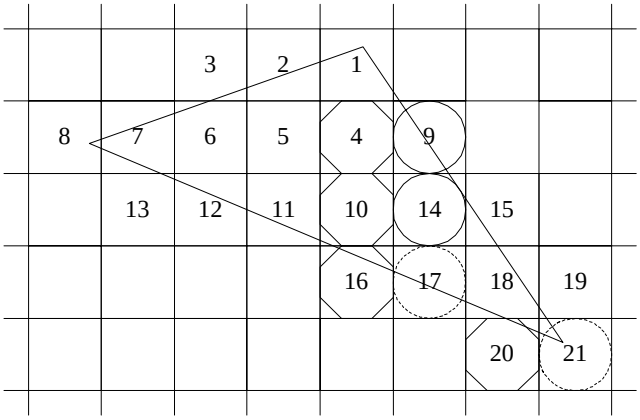

Figure 4: PixelVision's "centerline" algorithm. Octagons are *downSave* positions, circles are *rightSave* positions and dashed circles are bypassed *right* positions.

the *current* interpolator values, and defer computing the values we really want until the saved values are restored. For example, when the *right* position is stored into *rightSave*, the *current* value of the colors, etc., is stored into *rightSave*. When *rightSave* is restored, the saved values are added to the *x* increments that move the interpolated values one stamp position to the right.

The original PixelVision algorithm actually starts at the first vertex provided, at the cost of three saved contexts. Since the bounding box must be computed anyway, it is a small matter to select the top-most vertex and start there, eliminating the need for one of PixelVision's saved contexts. We observe in passing that the *rightSave* state can also be eliminated by starting at a vertex that is on a corner of the bounding box, and allowing the algorithm to traverse the triangle either top to bottom or bottom to top. However, this optimization does not work for antialiased lines, in which none of the four vertices is at a corner of the bounding box.

The choice of starting at the top-most vertex is arbitrary. In fact, the Neon graphics accelerator [8] starts at any vertex on the bounding box, and can traverse an object in stamplines that are either rows or columns. This allows it to paint OpenGL wide dashed lines, which require column stamplines for *x*-major lines, row stamplines for *y*-major lines, and traversal from the first vertex toward the second vertex. We have implemented all traversal algorithms with this generality, but we describe them in terms of a particular starting vertex for simplicity and clarity. For the same reason, we have also omitted descriptions of serpentine versions, which sweep back and forth across stamplines (or tilelines, in the tiled versions).

Neon extended the centerline algorithm for tiled traversal. Neon's implementation uses three additional contexts, at approximately 600 bits per context, and rather complex next-move decision making logic. We later discovered that two of the contexts were mutually exclusive, and so tiling could have been implemented with two additional physical contexts. But we omit a description of these algorithms in favor of an even better one.

5. AN ALTERNATE TRAVERSAL ALGORITHM

We now present an alternate way to traverse the triangle. Expressed as a non-tiling algorithm, this alternative appears to have no advantage over the centerline algorithm. However, the extension to tiling can be made with a single additional saved context, as described below in Section 6.

This algorithm starts at the left-most vertex. It sweeps the first stampline from left to right. It records the first valid *up* position in *upSave*, and the first valid *down* position in *downSave*. When it finishes the first stampline, it moves to the position in *upSave*, and sweeps right across that stampline, again recording the first valid *up* position in *upSave*. When it finishes the top stampline, it finds that it cannot load a valid context from *upSave*. It instead loads the position in *downSave*, and proceeds to sweep all the stamplines from there down, saving the first valid *down* position it finds in each stampline. When it finishes the bottom stampline, it can't load a valid position from *downSave*, and so is finished with the object.

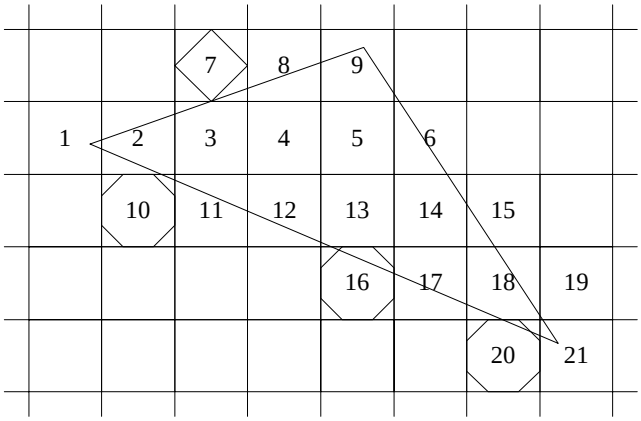

Figure 5: An alternate traversal algorithm. Octagons are *downSave* positions, diamonds are *upSave* positions.

Figure 5 illustrates this algorithm in action. Here, the octagons indicate positions stored in *downSave*, and diamonds indicate positions stored in *upSave*.

6. A TILING TRAVERSAL ALGORITHM

We can now describe an efficient tiling traversal algorithm. We add one more context, *rightTileSave*, to the previous algorithm. We also need to know for the *up*, *down*, and *right* positions if they are in the current tile or a different tile. If the tile and stamp height and width are restricted to powers of two, this is accomplished by logically ANDing or ORing the current stamp *x* and *y* coordinates with a precomputed mask or its complement, then testing the results for all 1's or all 0's.

While traversal of stamplines is horizontal (rows of stamps), traversal of tilelines is vertical (columns of tiles). In each tileline, the algorithm operates in three phases. In phase 0, the stamp visits all stamplines in the object including and below the starting position that are also inside the first tile of the tileline. In phase 1, it visits all stamplines in the object above the starting position that are in the tileline. In phase 2, it returns and finishes visiting all positions in the object in the tiles below the starting tile. The algorithm then repeats this process for the next vertical tileline to the right.

In phase 0, the algorithm proceeds along a row stampline from left to right, stopping as soon as it reaches the rightmost position in the tile or the object. It also saves the first valid *up* and *down* positions in *upSave* and *downSave*, regardless of tile boundaries. When it reaches the tile's right edge, if *right* is valid, it stores the *right* position (which is in the next tile) in *rightTileSave*. It then loads the *downSave* context, and visits from left to right all positions on that stampline that are within both the tile and object. Again, it saves the first valid *down* position it sees into *downSave*. It continues visiting new stamplines by restoring *downSave* until *downSave* is in a new tile or no longer valid. It then restores the *upSave* position, and enters phase 1.

In phase 1, it sweeps stamplines from left to right, stopping at the object's or tile's right edge, and saving the first valid *up* position in each stampline. It moves from stampline to stampline by restoring the *upSave* context at the end of each stampline. It traces out the rest of the object above the starting position and within the tileline, without regard to tile boundaries in the *up* direction. (It still respects the tile boundary in the *right* direction.) When it reaches the top of the object, and cannot restore from *upSave*, it instead restores *downSave* and enters phase 2.

In phase 2, it traces out the rest of the tileline, that is, the portion of the object below the starting tile, by moving from stampline to stampline in the *down* direction. Like phase 1, it only respects the tile boundary in the *right* direction. When it reaches the bottom of the object, and cannot restore from the *downSave* context, it restores from *rightTileSave* and enters phase 0 again. If *rightTileSave* is empty, the object has been traversed.

Figure 6 shows this algorithm traversing a larger triangle. The thick grid lines represent tile boundaries, which here are 4 pixels wide by 4 pixels high. Octagons are *downSave* positions, diamonds are *upSave* positions, and triangles are *rightTileSave*

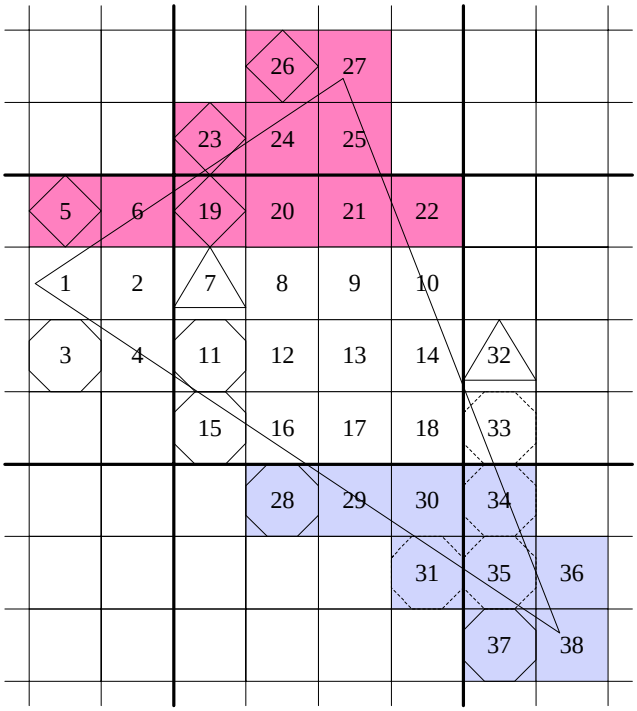

Figure 6: Tiling traversal algorithm. Octagons are *downSave* positions, diamonds *upSave*, and triangles *rightTileSave*. Phase 0 is white, phase 1 is dark rose above, and phase 2 is light lavender below.

positions. White positions represent phase 0, dark rose phase 1, and light lavender phase 2. In this example, no positions are visited in phase 2 in the left-most tileline, and no positions are visited in phase 1 in the right-most tileline. Just as bypassing saved contexts avoids wasting cycles, phases in which there is no work should also be bypassed.

7. USES AND VARIANTS OF TILING

Two uses of tiling have been mentioned previously: tiling with respect to DRAM page boundaries, and tiling with respect to a texture cache size.

7.1. Tiling to Frame Buffer Pages

Respecting DRAM page boundaries affords two benefits [8]. First, tiling reduces the number of same-bank page transitions, for which the second page cannot be prefetched. When using any type of DRAM for a frame buffer, such transitions require all accesses to the first page to complete, and then an expensive precharge/row activate command sequence to load the second page into the bank. This takes several cycles longer than accessing data on an already open page. Since tiling produces fewer transitions between pages, it also produces fewer transitions from one page to another in the same bank.

Second, tiling increases the effectiveness of prefetching pages. Multibank DRAM (e.g. synchronous DRAM or RAMBUS Direct RAM) allows accessing a page in one bank while prefetching (issuing a precharge/row activate sequence for) a page into another bank. By grouping all accesses for an object on a page, tiling spaces transitions from one page to another further apart in time. This allows more time to prefetch the new page, so

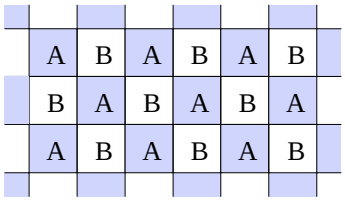

Figure 7: Checkerboarded banking.

that the precharge and row activate cycles can be hidden by data accesses to different banks.

When multibank DRAM is used, the tiling traversal algorithm can be modified to further decrease, on average, the number of nonprefetchable transitions from a page in one bank to another page in the same bank. For example, a graphics accelerator that uses 2-bank DRAM might checkerboard the two banks across the screen, as shown in Figure 7. Moving to a horizontally or vertically adjacent tile moves to a page in a different bank, while moving to a diagonally adjacent tile moves to a page in the same bank. In the basic tiling algorithm illustrated in Figure 6, the jump from position 31 to position 32 moves from one page in one bank to another page in the same bank.

When painting objects that span many tiles, the basic tiling algorithm may jump a large distance when it restores *rightTileSave* to move from one tileline to another. The new tile has good chance of being in the same bank. We can modify the algorithm to instead move to a horizontally adjacent tile, which is in a different bank, whenever possible. This modified algorithm visits tiles in a serpentine fashion. It tries to visit tiles in one tileline from top to bottom, then from bottom to top in the next tileline, and again from top to bottom in the next tileline, etc. This tile traversal order increases the frequency of transitions from one bank to another when moving from one tileline to another.

Serpentine tile traversal is accomplished by saving the last (rather than first) valid *right* position in the next tileline into *rightTileSave*, and swapping the roles of *up* and *down* in alternate tilelines. The serpentine algorithm moves from position 31 to position 35 in the tile immediately to the right, and thus to a page in a different bank.

Slightly better performance can be obtained by using another saved context, and replacing *rightTileSave* with *rightTileSaveEven* and *rightTileSaveOdd*. A valid *right* position in an even bank page in the next tileline (bank A in the 2-bank example above) is stored into *rightTileSaveEven*, while a valid *right* positions in an odd bank page (bank B) is stored into *rightTileSaveOdd*. If both of these contexts contain a valid position, then the transition from one tileline to another restores whichever is in a different bank from the current position. In Figure 6, for example, position 32 might be saved in *rightTileSaveEven*, and position 34 in *rightTileSaveOdd*. At position 31, when the current tileline has been completely visited, the algorithm restores position 34 from *rightTileSaveOdd*, and then invalidates both the *rightTileSaveEven* and *rightTileSaveOdd* contexts. Note, however, that as DRAM pages grow larger, and more banks are provided, this optimization becomes less significant.

7.2. Tiling for Texture Cache Performance

Rather than matching the tile dimensions to the frame buffer page dimensions, we can instead make the tile size a function of the size and organization of the first level of the texture cache. This improves the texture cache hit rate [4][5][8].

Many of the four or eight texels required to bilinearly or trilinearly texture a fragment will also be used to texture nearby

fragments in the object. If a long stampline causes texels fetched for the left-most fragments on the stampline to be ejected from the texture cache, the shared texels must be refetched from memory in order to texture the left-most fragments on the next stampline.

Tiling forces the fragment generator to move to the next stampline before these texels are ejected from the cache. Hakura & Gupta [4] found that for scenes with large triangles, tiling reduces the miss rate by a factor of three for fully associative texture caches between 2 kbytes and 8 kbytes in size. Tiling also reduces the conflict misses in less associative caches: a 2-way set associative cache with tiling and the proper texture memory organization performs nearly as well as their fully associative cache.

Tiling means that the texels used to texture fragments at the right-hand side of the tile will probably be ejected from the cache before visiting the left-hand side of the tile to the right, which is in a different tileline. These texels must be refetched when the next tileline is visited. If the texture cache is small, special care should be taken to minimize this effect. For example, the Neon graphics accelerator [8] has 8 tiny fully associative 32-byte texture caches, each with a line size of one 4-byte texel, for a total of 256 bytes of texture cache. Neon uses a tile that is 16 pixels wide by 1 pixel high when texture mapping is enabled. These dimensions maximize the shared tile perimeter between vertically adjacent tiles in a tileline, which are generated closely in time, while minimizing the shared perimeter between horizontally adjacent tiles in different tilelines, which are generated much farther apart in time.

Serpentine tiling further increases sharing of cache data between tiles when moving from one tileline to another. Many positions within a tile near the top or bottom of a tileline will not be within the object, and so will not require texel data. But the tile size is usually chosen for the worst-case situation, when all fragments within the tile need texture data. Thus, the cache may contain texels used by several tiles at the end of a tileline. By making the transition from one tileline to another with as small a jump as possible, serpentine tiling increases the odds that this texel data will still be there when nearby tiles in the next tileline are visited.

Finally, note that tiling can be made object-relative, rather than screen-relative, by using the starting position of the stamp as an offset to the stamp's (x, y) position before testing for a tile boundary. This has the effect of aligning tiles to the starting vertex of the object, which may slightly increase cache performance for large triangles.

8. METATILING

Multiple levels of tiling (metatiling) may be desirable in several circumstances described in the following subsections. Metatiling may be *inclusive*, where each tile belongs to exactly one metatile, or *noninclusive*, where a tile may belong to two or more metatiles. Fortunately, the same stamp movement rules can cover both cases. Traversal with metatiling generates all fragments within a metatile before moving to another metatile. The method further generates all fragments on the portion of a tile that is within the current metatile before moving to a position in a different tile. If metatiling is inclusive, this means that *all* fragments in a tile will be generated before any fragments in a different tile.

Metatiling is a straightforward (though nontrivial) extension to tiling, in which the relationship between a tile and a metatile is almost like the relationship between a stamp position and a tile. Each of the metaphases 0, 1, and 2 contain a version of the basic tiling algorithm's phases 0, 1, and 2 that consider a metatile boundary the end of the world. For example, the embedded tile phase 1 does not visit all positions in the object and the current tileline that are below the starting position, but rather stops at a metatile boundary. Unfortunately, given a starting vertex on an edge of the bounding box, we have been unable to construct a metatiling algorithm with fewer than three additional saved contexts: *upMetatileSave*, *downMetatileSave*, and *rightMetatileSave*. This results in seven contexts.

Alternatively, metatiling can be implemented with five contexts by always starting at a vertex on the corner of the bounding box. As previously noted, antialiased lines do not have such a vertex. But they can be rendered as a quadrilateral and a triangle, which share a horizontal or vertical edge and have the same vertex at the corner of the two subbounding boxes. In fact, the Neon accelerator uses a similar technique for antialiased and X11 wide lines, in order to avoid computing a starting position at a vertex. Instead, it starts at one of the provided endpoints that are interior to the wider desired line, which it then effectively renders as a pentagon and a triangle. Such a scheme can be further optimized to avoid visiting many of the same stamp locations that are bisected by the two subpolygons. Note, however, that this five-context algorithm cannot be extended to support serpentine tiling.

8.1. Inclusive Metatiling

With a single level of tiling, it is impossible to simultaneously tile to frame buffer page boundaries, and to optimize the hit rate of the texture cache. A tile size chosen for minimizing page crossings may not be optimal for reducing cache misses, and vice-versa. If this problem is severe, metatiling may be warranted.

Alternatively, metatiling might be used to further improve only texture cache performance or only frame buffer access efficiency. For example, if the first level texture cache is very small, another level of tiling can improve second level texture cache efficiency. Similarly, if DRAM with multiple cache levels is used, like FBRAM [2], another level of tiling can improve efficiency of frame buffer memory accesses.

8.2. Noninclusive Metatiling

Noninclusive metatiling can efficiently group memory accesses when a source rectangle is copied to a destination rectangle. Without metatiling, if the destination position (x_d, y_d) determines tile boundaries we indiscriminately cross source page boundaries; if the source position (x_s, y_s) determines tile boundaries we indiscriminately cross destination page boundaries. And if the copy involves off-screen data, the source and destination pages may not even be the same height and width, or contain the same number of pixels.

With noninclusive metatiling, the destination page dimensions determine the metatile dimensions, while the source page dimensions determine the tile dimensions (or vice-versa). The destination position (x_d, y_d) is tested against metatile boundaries, while the source position (x_s, y_s) is tested against tile boundaries. In general, source tile boundaries are not aligned with the destination metatile boundaries. Tiles are no longer subsets of metatiles, but the traversal order rules remain the same. The copy rectangle is visited in an order that traces out all positions within a destination metatile before moving to another destination metatile. And all positions within both the current source tile and the current metatile are visited before moving to another source tile.

Since copies involve rectangles rather than arbitrary polygons, we always start at a vertex on the edge of the bounding box, and thus can use the 5-context metatiling algorithm. Assume that we wish to copy starting at the top left corner, from left to right, then top to bottom. (The other three vertices are similar.) For this case we label the five physical contexts *current*, *downSave*, *downMetatileSave*, *rightTileSave*, and *rightMetatileSave*. The *downSave* context records the first valid *down* position in the

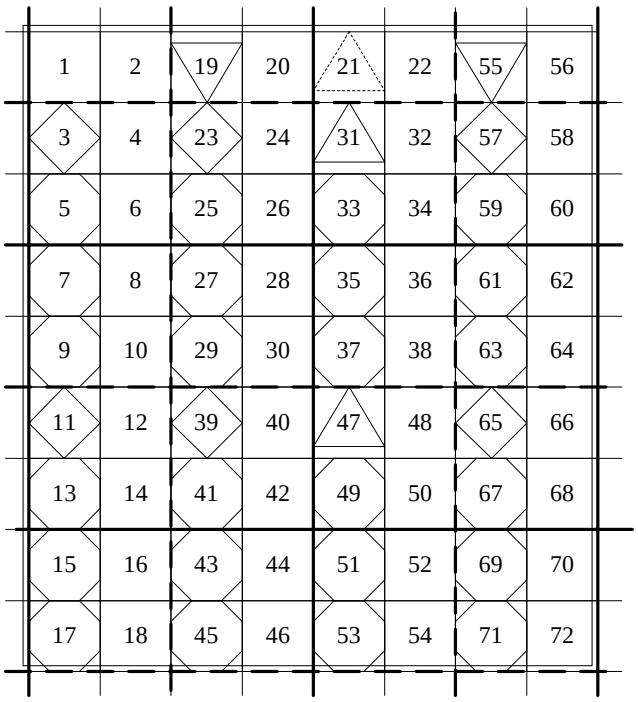

Figure 8: Copy operation with metatiling. Octagons are *downSave* positions, diamonds are *downMetatileSave*, upright triangles are *rightTileSave*, and upside-down triangles are *rightMetatileSave*.

same metatile. The *downMetatileSave* context records the first valid *down* position in the next metatile. The *rightTileSave* records the first valid *right* position in the next tile and the same metatile, while *rightMetatileSave* records the first valid *right* position in the next metatile.

Figure 8 shows an example of this algorithm in operation. The thick solid grid lines are again tile boundaries, while the thick dashed grid lines are metatile boundaries. (In this example, the four saved contexts mentioned are never simultaneously needed, but would be if there were more tiles within the metatiles.)

9. CONCLUSIONS

The efficiency of both frame buffer and texture memory accesses can benefit from tiled rasterization of polygonal objects, yet we know of no published hardware algorithms for tiling. We have presented a simple algorithm that tiles the traversal of polygons. The algorithm is simpler than our previous, more expensive algorithm, which was nonetheless worth the implementation cost. The algorithm is easily extensible to serpentine traversal, which further improves frame buffer and texture cache efficiency. The algorithm is also easily extensible to metatiles, though at a cost that probably warrants its use only when both tiles and metatiles are fairly small, and thus the benefits are large. Further details, as well as a set of optimizations that avoid moves to unproductive stamp positions, are available in [9].

10. ACKNOWLEDGEMENTS

Laura Mendyke and Todd Dutton were largely responsible for implementing six-context tiling ("chunking") in the Neon graphics accelerator. We apologize for coming up with a better algorithm after they were done.

References

[1] Kurt Akeley. RealityEngine Graphics. *SIGGRAPH 93 Conference Proceedings*, ACM Press, New York, August 1993, pp. 109-116.

[2] Michael F. Deering, Stephen A. Schlapp, Michael G. Lavelle. FBRAM: A New Form of Memory Optimized for 3D Graphics. *SIGGRAPH 94 Conference Proceedings*, ACM Press, New York, July 1994, pp. 167-174.

[3] Henry Fuchs, et. al. Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes. *SIGGRAPH 85 Conference Proceedings*, ACM Press, New York, July 1985, pp. 111-120.

[4] Siyad S. Hakura & Anoop Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, ACM Press, New York, June 1997, pp. 108-120.

[5] Homan Igehy, Matthew Eldridge & Kekoa Proudfoot. Prefetching in a Texture Cache Architecture. *Proceedings of the 1998 EUROGRAPHICS/SIGGRAPH Workshop on Graphics Hardware*, ACM Press, NY, August 1998, pp. 133-142.

[6] Homan Igehy, Matthew Eldridge & Pat Hanrahan. Parallel Texture Caching. *Proceedings of the 1999 EUROGRAPHICS/SIGGRAPH Workshop on Graphics Hardware*, ACM Press, NY, August 1999, pp. 95-106.

[7] Brian Kelleher. *PixelVision Architecture*, Technical Note 1998-013, System Research Center, Compaq Computer Corporation, October 1998, available at http://www.research.digital.com/SRC/publications/src-tn.html.

[8] Joel McCormack, Robert McNamara, Chris Gianos, Larry Seiler, Norman Jouppi, Ken Correll, Todd Dutton & John Zurawski. *Neon: A (Big) (Fast) Single-Chip 3D Workstation Graphics Accelerator*. Research Report 98/1, Western Research Laboratory, Compaq Computer Corporation, Revised July 1999, available at http://www.research.compaq.com/wrl/techreports/pubslist.html.

[9] Joel McCormack & Robert McNamara. *Efficient and Tiled Polygon Traversal Using Half-Plane Edge Functions.* Research Report 2000/4, Western Research Laboratory, Compaq Computer Corporation, August 2000, available at http://www.research.compaq.com/wrl/techreports/pubslist.html.

[10] John S. Montrym, Daniel R. Baum, David L. Dignam & Christopher J. Migdal. InfiniteReality: A Real-Time Graphics System. *SIGGRAPH 97 Conference Proceedings*, ACM Press, New York, August 1997, pp. 293-302.

[11] Juan Pineda. A Parallel Algorithm for Polygon Rasterization. *SIGGRAPH 88 Conference Proceedings*, ACM Press, New York, August 1988, pp. 17-20.