Depth-Cull Optimization of 2D Scenes for 3D Graphics Hardware

Andrew Church
http://achurch.org/

Abstract

Many recent consumer computing devices include graphics hardware which emphasizes strong performance in three-dimensional (3D) graphics computations at the expense of pixel operations, a tradeoff which negatively impacts applications using 2D graphics. We describe a rendering method by which a scene consisting solely of alpha-blended, partially opaque 2D images can be rendered more quickly than the simple overpainting method without requiring time-consuming manual effort to optimize each image.

1. Introduction

In recent years, advances in computing technology have enabled the use of high-performance 3D graphics hardware in portable, low-power consumer devices such as smartphones and tablets. This has enabled the development of games displaying 3D environments, such as the award-winning *Infinity Blade*.

However, these advances have come at the cost of relatively reduced performance for individual pixel operations such as color blending. Table 1 shows relative performance of three consumer devices: Sony's first-generation PlayStation Portable game system, which uses custom hardware with limited 3D graphics support, and Apple's first- and third-generation iPad tablet computers, which use commodity 3D graphics hardware.

Device	Year produced	Screen size (pixels)	Overdraws /second	Mpixels /second
PSP	2005	480×272	1,550	202
iPad 1	2010	1024×768	248	195
iPad 3	2012	2048×1536	609	1,920

Table 1. Relative pixel-operation performance of three consumer computing devices. "Overdraws/second" indicates the number of times the entire screen can be painted in one second; "Mpixels/second" gives the same rate in millions of pixels per second. Performance was measured by repeatedly rendering a 10% opaque, flat-shaded, untextured rectangle over the entire screen using a 32-bit render buffer, doubling the number of render operations until the total time taken fell in the range [0.25,0.5) seconds, then dividing the number of render operations by the time taken. Renderer synchronization was performed both before and after each test, using relevant system calls on the PSP and OpenGL's `glFinish` command on the iPad.

As can be seen from the table, the PSP outperforms the much newer iPad 1 in both relative and absolute measures; even the iPad 3, which has a greater absolute rate of operations, suffers from a larger screen size and cannot match the PSP's overdraw rate.

These per-pixel operations form the backbone of games based on 2D art, and a decrease in overdraw rate leads directly to a decrease in graphics fluidity. To maintain a redraw rate of 60 frames per second, the native display rate of most modern consumer devices,[1] a game can render over the screen up to 25 times on the PSP but only 4 times on the iPad 1 – and even less when system overhead is taken into consideration.[2]

The need for multiple overdraws in 2D games stems from the fact that 2D art generally cannot be mapped to geometric shapes, and therefore cannot take advantage of the geometry-based 3D hardware; each image must be rendered on top of the current screen state, a method known as "overpainting". As the number of these 2D objects in a scene increases, so does the number of screen overdraws required to display the complete scene. To make things worse, 2D objects are typically stored in rectangular image buffers; if the object is not rectangular, the hardware may waste significant time "drawing" the transparent pixels outside the border of the object but within the object's image buffer.

The presence of 3D hardware suggests the optimization of treating each 2D object as a polygon parallel to the screen in 3D space; by rendering the opaque portions of each object from nearest to farthest, the hardware will be able to discard pixels which will not be seen by the user and thus reduce the total number of pixel operations required to render the scene. However, in order to take advantage of this optimization, each image must be divided into opaque and non-opaque parts, and a bounding polygon must be constructed around each image to exclude transparent pixels. In complex games with thousands of separate images, this can be a highly time-consuming manual task – one which must be repeated each time an image is changed. The polygons must also be rendered in the proper order for the hardware to properly discard overpainted pixels.

2. Method

We describe a method for automating the generation of polygon meshes[3] for scenes consisting solely of 2D images. The method consists of three stages: boundary polygon generation, opaque polygon generation, and scene reconstruction. See Appendix A for a complete pseudocode listing, of which the `GenerateSceneMeshes` routine is the entry point; each component of the method will be discussed in detail below.

Of the three stages referenced above, boundary and opaque polygon generation rely only on the image data. If the generated polygon data from these stages is cached externally, it can be reused for other scenes as long as the image itself does not change.

In this description:

- *scene* refers to the graphic output presented to the user for a single display frame, consisting of zero or more elements;
- *element* refers to a single graphical component of a scene, minimally consisting of an image and positioning information (such as size, rotation, and scale); and
- *image* refers to a single graphic asset, which may be used in multiple elements.

For the purposes of this description, we assume that every element has an associated image. In practice, some elements may be solid or shaded rectangles with no associated image, but these are

1 Actually $60 \div 1.001 \approx 59.94$ frames per second, to match NTSC display devices such as televisions.

2 Indeed, the author found a practical limit of approximately 2.5 overdraws per frame on the iPad 1 using iOS version 3.2 when following Apple's published best practices for OpenGL software [Apple], which at the time recommended disabling the system's automatic rotation features and instead manually rotating OpenGL transformation matrices to improve throughput.

3 The term "mesh" is admittedly something of a misnomer, since the contained polygons are generally not connected and are grouped together only to reduce the total number of render operations.

in essence images whose pixels are all opaque, and they can be handled by trivial modifications to the algorithm. Similarly, elements may have various visual effects applied to them, but by using a separate mesh for each effect, the effect can then be applied to the mesh as a whole.

2.1. Boundary polygon generation

For each distinct image used in the scene, we generate a bounding polygon which encloses all non-transparent pixels in the image and as few transparent pixels as possible. These polygons will be used in place of the image rectangles when reconstructing the scene so the hardware does not waste time rendering transparent pixels. See the `GenerateBoundaryPolygon` routine for a pseudocode description of the algorithm. Figures 1a-1d show an example of boundary polygon generation.

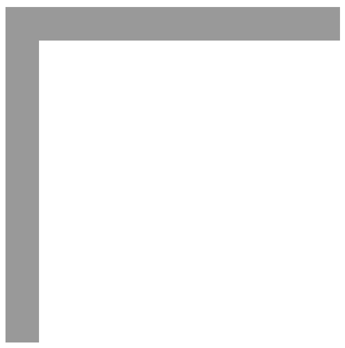
Figure 1a. A simple image around which a boundary polygon will be fitted.

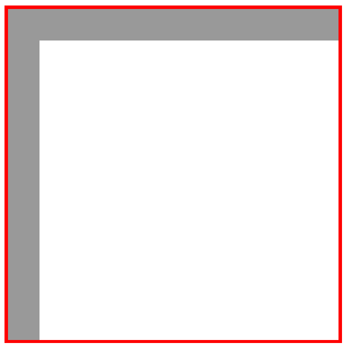
Figure 1b. The boundary polygon is initialized to the image's axis-aligned bounding box.

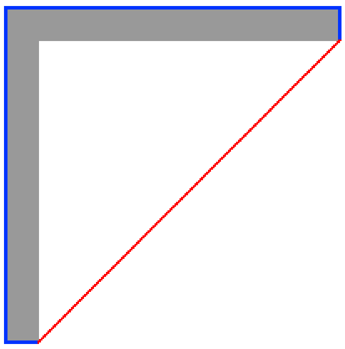
Figure 1c. The lower-right corner is cut, removing the lower-right vertex and adding a new vertex on each of the adjacent edges.

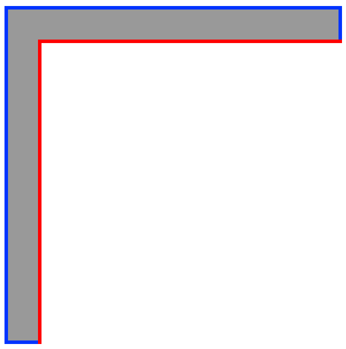
Figure 1d. A vertex is inserted along the new diagonal edge, pulling it up against the inner corner of the image. In this case, the polygon is now a perfect fit for the image, and processing terminates.

The basic algorithm is to initialize the boundary polygon to the axis-aligned bounding box of the image, then add one vertex at a time until the polygon either fits the image perfectly or has enough vertices to satisfy the desired balance of accuracy and performance. In the pseudocode, "enough" is defined by the constant `MAX_BOUNDARY_VERTICES`, but it could also be set for each image based on the image's frequency of use or other factors.

The algorithm uses two methods to add a new vertex to the polygon. One is to cut a transparent corner from the polygon, deleting the vertex at that corner and inserting a new vertex on each adjacent edge (as in Figure 1c); the other is to insert a vertex along an existing edge, using it to "pull" the edge closer to the boundary of the image (as in Figure 1d).

Insertion of a new vertex may create a transparent corner in the boundary polygon, typically as the result of shifting an adjacent vertex outward. This can occur in complex polygons when several vertices are close together, but more often it is the result of inserting the new vertex close to one of the existing vertices, creating a thin “spike” at that vertex (see Figure 2b for an example of this case). Removing this corner will bring us back to the original number of vertices with a smaller area than the original polygon, so if we find such an corner, we remove it and return immediately, letting the caller repeat the iteration with the new polygon. An example of transparent corner removal is shown in Figures 2a through 2c, below.

2.1.1. Corner cutting

Of the two vertex addition methods, the corner-cutting method is the simpler one. The corner-cutting algorithm searches for potential cut lines by keeping two cut points that run along the polygon in the same direction: one (call it A) starting from a given vertex V, and the other (call it B) starting from the previous vertex. A is advanced one pixel at a time until it reaches the next vertex; at each step, B is advanced toward V until the line segment AB covers only transparent pixels, then the polygon created by removing triangle AVB from the original polygon is added to the set of candidate results. If B ever reaches V, processing is terminated since no further cuts can be made. Once all candidate results have been enumerated, the function returns the candidate with the smallest area to the caller, as long as that candidate is smaller than the original polygon.

See `CutBoundaryCorner` for a pseudocode description of the corner-cutting algorithm.

2.1.2. Vertex insertion

By contrast, the vertex insertion method is somewhat more complex due to the larger search space and the effects on adjacent vertices. The vertex insertion algorithm chooses the location for the new vertex by finding the point on the edge with the longest perpendicular distance to the image (*i.e.,* to a non-transparent pixel), then shifting that vertex inward one pixel at a time until it reaches the image. At each position, the adjacent vertices are shifted along their respective edges as needed to prevent any edge from crossing the image; if this causes a vertex to move outside the bounds of the image’s pixel buffer or to reach the end of its edge, the search is aborted at that point. The resulting polygon with the smallest area is then returned, as long as it is smaller than the original polygon.

Since pulling the edge inward may require shifting adjacent vertices outward, the point on the edge with the greatest perpendicular distance to the image may not necessarily be the best insertion point for the vertex, and the perpendicular to the edge may not be the best direction in which to shift the inserted vertex. However, searching the entire area between the edge and the image would result in fourth-power complexity per vertex for the search, so the greatest-distance heuristic was chosen to avoid this increase in complexity.

See `InsertBoundaryVertex` for a pseudocode description of the vertex insertion algorithm.

2.1.3. Transparent corner removal

As mentioned above, the insertion of a new vertex may result in a completely transparent corner which can be cut to reduce both the number of vertices and the polygon’s area. For five sequential vertices P, A, B, C, and Q in a polygon (where P and Q may be the same vertex), there are up to three triangles at vertex B whose removal would reduce the polygon’s vertex count:

- the triangle formed by vertices A, B, and C;
- the triangle formed by vertices A and B and the intersection of ray PA with edge BC; and
- the triangle formed by vertices B and C and the intersection of ray QC with edge AB.

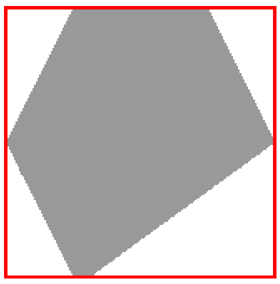

Figure 2a. An image with its initial boundary polygon.

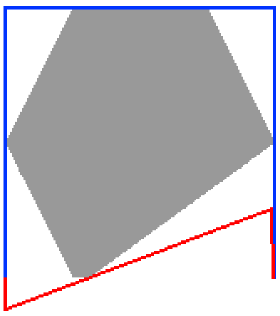

Figure 2b. A vertex is inserted on the lower edge immediately adjacent to the existing lower-right vertex. This leaves a narrow transparent "spike" jetting out from the lower-right corner of the polygon.

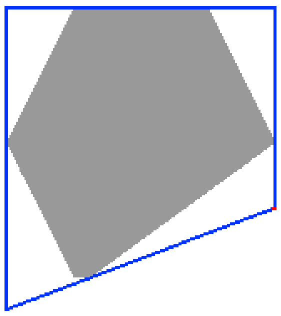

Figure 2c. The transparent "spike" is cut by extending the new lower edge of the polygon to intersect with the right edge. Combined with the previous step, this has the effect of shifting both of the lower vertices at once, with the result that the polygon's total area is now smaller and some of the transparent area has shifted from the lower-right corner to the lower-left corner.

For each of the three cases listed above, the algorithm traces from the common vertex (B) to each pixel location on the opposite edge, checking whether the triangle contains any non-transparent pixels. This is somewhat wasteful in that some pixels will be checked multiple times, but it is simpler than trying to ensure that each pixel is checked only once without missing any, and the runtime of this routine is typically insignificant compared to the overall process.

See `RemoveTransparentCorner` for a pseudocode description of the transparent corner removal algorithm.

2.2. Opaque polygon generation

Along with the boundary polygon, we also generate for each image a set of polygons which cover as many opaque pixels in the image as possible but no non-opaque pixels. These polygons will be rendered in advance of the full-image boundary polygon, allowing the hardware to discard the pixels from the full-image polygon (and any polygons beneath it) when rendering. See `GenerateOpaquePolygons` for a pseudocode description of the overall algorithm.

The algorithm first creates a directed graph of all opaque horizontal segments in the image – that is, all sets of adjacent opaque pixels with the same Y coordinate such that there is not an opaque pixel adjacent to the outermost pixels in the set. The algorithm then repeatedly looks for an opaque quadrilateral which covers as many not-yet-covered pixels in the image as possible, continuing until either:

- the requested maximum number of opaque polygons has been generated,
- the requested minimum opaque coverage fraction has been reached, or
- there are no remaining opaque quadrilaterals covering enough uncovered opaque pixels that the time saved not rendering the obscured pixels is expected to exceed the time spent rendering the additional polygon (this threshold is defined in the pseudocode as the constant `MIN_OPAQUE_AREA`).

Since the intent of generating opaque polygons is to cover as much of the original image as possible, not just to find any large polygons, the algorithm keeps track of which pixels have been covered and uses this data to find the "best" polygon in each iteration. In the pseudocode, this data is stored in the variable `covered_map`; this variable has one element for each pixel in the image, with value 0 if the pixel is not opaque, +1 if the pixel is opaque but not yet covered, and −1 if the pixel is opaque and has already been covered.

2.2.1. Opaque segment generation

The algorithm first iterates down the image one line at a time, scanning across each line for sequences of opaque pixels. When such a sequence is found, the algorithm adds a new segment record for the sequence, then finds all segments on the previous line whose horizontal positions overlap with the current segment and creates graph edges between them.

See `FindOpaqueSegments` for a pseudocode description of the opaque segment generation algorithm.

2.2.2. Opaque polygon detection

Once the opaque segment graph has been created, the algorithm proceeds to recursively find all quadrilaterals that can be formed from chains of vertically-adjacent opaque segments, returning the one which covers the largest number of opaque pixels that have not already been covered. As the algorithm recurses down a branch of the opaque segment graph, it tracks the segments which limit the angles of the side edges – in other words, the segments on each side of the quad at whose Y coordinate the edge would first encounter a non-opaque pixel if it was rotated outward from the top. These segments are used to compute the slopes of the two side edges, and thus the width of the bottom edge (which always lies within the bottom segment); if the limiting segments constrain the bottom edge to a width of zero, then no further quads exist on that branch, and the algorithm moves on to the next branch. Figures 3a through 3i illustrate how the algorithm processes an image with a moderately complex shape.

See `FindOpaqueQuad` for a pseudocode description of the opaque polygon generation algorithm.

The astute reader will notice that a straightforward implementation of `FindOpaqueQuad` has a search complexity of $O(N \cdot h^b)$, where N is the total number of opaque horizontal segments in the image, h is the image height, and b is the average branching factor of the opaque segment graph. For an image whose opaque region is a simple convex polygon, $N = h$ and $b = 1$, so this reduces to $O(h^2)$; for more complex images, such as an image with many small holes, performance can drop off significantly. On top of this, counting the number of covered pixels is $O(w \cdot h)$, where w is the width of the image; this operation is required for every opaque polygon found, so the overall runtime complexity of the algorithm balloons to $O(w \cdot h^3)$ even in the case of a simple opaque region (thus $O(h^4)$ for a square image).[4]

2.3. Scene reconstruction

Once we have generated the boundary and opaque polygons for each image, we reconstruct the scene by replacing the image rectangle for each element with the image's polygon set and assigning appropriate Z-values (effectively depth values) so that the graphics hardware can detect and discard pixels which will not be seen by the user. Scene reconstruction includes three distinct stages, as shown in `ReconstructScene`:

4 This routine in fact consumed the bulk of the time required to process most scenes in *Aquaria,* despite optimizations to avoid repeated covered-pixel counts for simple shapes.

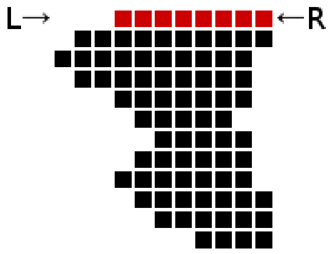

Figure 3a. Searching starts with the topmost opaque segment. (In this image, each line has only one opaque segment, so the segment graph has only a single, linear branch.)

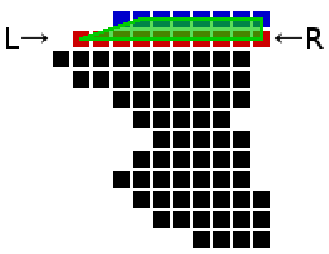

Figure 3b. In the first iteration, a simple quadrilateral is formed from the inital segment and the segment immediately below it. Note how the upper-left corner is shifted inward by 1 pixel so the quad does not cover any non-opaque pixels.

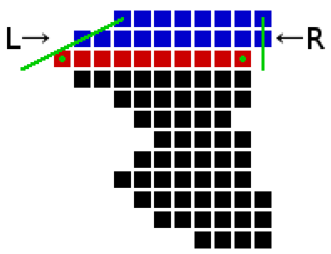

Figure 3c. In subsequent iterations, the edges of the next segment are compared against the extensions of the previous quad's sides. (For this comparison, obtuse-angle corners are *not* shifted inward.)

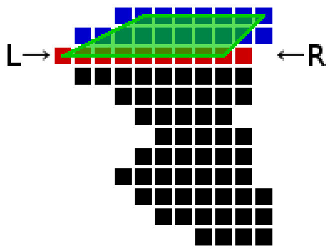

Figure 3d. In this case, the quad's edges are outside the next segment on both sides, so both the left and right limiting segments are set to the new segment.

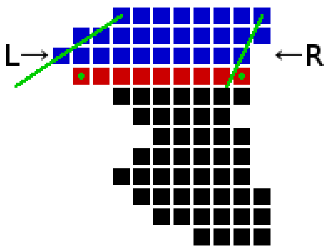

Figure 3e. For this iteration, the left edge of the quad is outside the next segment, but the right edge is is inside the segment . . .

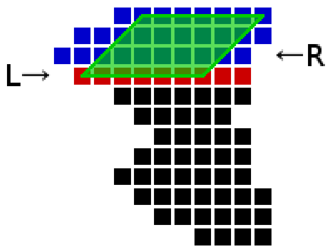

Figure 3f. . . . so the right limiting segment is left alone. However, the right edge is still extended down to the next segment to give a horizontal bottom edge.

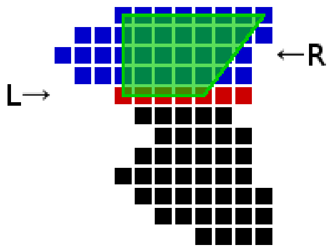

Figure 3g. The process is repeated for each subsequent segment on the branch . . .

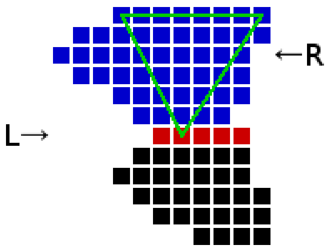

Figure 3h. . . . until the left and right edge constraints are such that the two edges intersect at or above the segment. (Here, a triangle could be generated, but this is not included in the set of candidate polygons.)

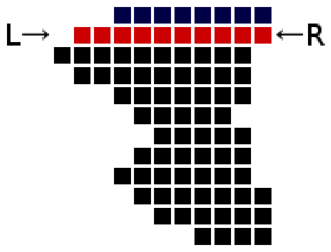

Figure 3i. The process is repeated, using the next segment as the top edge, and so on until all possible quadrilaterals have been generated.

- *element sorting,* to move elements which can share a mesh next to each other in the rendering order when possible;
- *depth value assignment,* to choose Z (depth) coordinates for each element so that opaque polygons properly occlude those underneath them; and
- *mesh generation,* to create the actual polygon meshes to be used for rendering the scene.

2.3.1. Element sorting

One of the keys to reducing the workload on the renderer is to reduce the number of distinct meshes rendered. In the unoptimized overpainting method, each element is effectively a separate mesh, even though a given image may be reused many times in a scene. The element sorting step searches for elements using the same image[5] and reorders the element list such that, when possible, such elements are adjacent in the list; this will allow the elements to be rendered as a single mesh.

Of course, it is imperative that any optimizations performed on the scene do not change how the scene actually looks. Here, the sort operation must ensure that any changes in the order of elements do not affect their front/back ordering when rendered. To this end, the algorithm first generates a directed graph of all elements in the scene, with an edge from an element A to another element B (indicating that A is in front of B) if A, as rendered, would overlap any part of B. (In the pseudocode, the graph is generated in `ReconstructScene` and passed in, since the graph is also used in the next step of assigning depth values.)

With this graph in hand, the algorithm proceeds to scan over each element in the scene's element list. For each element E_i (except the last) in the list, the routine scans forward for the next element E_j ($j > i$) which uses the same image. If there is *no* intermediate element E_k ($i < k < j$) such that the element graph contains an edge from E_j to E_k, then E_j can be moved to the position immediately after E_i without changing the appearance of the scene. Otherwise, the algorithm continues scanning forward for another E_j until it finds an element that can be moved or reaches the end of the list.

See `SortElements` for a pseudocode description of the element sorting algorithm.

2.3.2. Depth value assignment

The algorithm for depth value assignment is trivial; essentially, it calculates the distance on the element graph (as used in element sorting, described above) from each element to the far plane of the scene and converts this to a Z coordinate value suitable for passing to the renderer as a vertex position. (Since 2D scenes are normally rendered using an orthogonal camera transformation, Z values are purely depth values and do not affect how the scene appears.)

As the algorithm iterates over elements, it assigns depth values such that there is at least one intermediate value between the values assigned to any pair of elements which the renderer will treat as distinct from both elements' depth values. If an element E is assigned depth value k, then E's boundary polygon is rendered using Z coordinate k, but E's opaque polygons are rendered using Z coordinate $k+\varepsilon$ (in other words, the intermediate value mentioned above); this ensures[6] that the renderer does not render opaque pixels of a given polygon twice.

See `AssignDepthValues` for a pseudocode description of the element sorting algorithm.

2.3.3. Mesh generation

The final step in scene reconstruction is to generate the new meshes to be used to render the scene. The algorithm consists of iterating over each element in the scene, adding its boundary polygon to

5 More precisely, it checks for elements which can be rendered as part of the same mesh, which depends on the set of element parameters which must be constant during a single render operation. Additional effects such as blending may prevent two elements using the same image from being merged into a single mesh; conversely, the use of texture atlases may allow elements using different images to be rendered in the same mesh. For this description, we assume that the image is the sole determinant of whether elements can share meshes and that elements with different images can never share the same mesh. In the pseudocode, this decision is encapsulated in the `CanShareMesh` function.

6 Since we never assign the same depth value to overlapping elements, this "intermediate depth value" may in fact be unnecessary if the renderer's depth test is configured such that pixels with the same depth value as an already-rendered polygon fail the test (`glDepthFunc(GL_LESS)` in OpenGL). The intermediate value was used for clarity, and to avoid depending on specific renderer behavior more than necessary.

one mesh and its opaque polygons (if any) to a second mesh. When the algorithm encounters an element which cannot share meshes with the previous element in the list, it terminates the previous meshes, pushes them onto non-opaque and opaque mesh lists respectively, and creates new meshes into which the new element's polygons will be added. See `GenerateMeshes` for a pseudocode description of the algorithm.

The use of two separate meshes is to allow all opaque polygons to be rendered before all non-opaque polygons. This separation improves performance on some renderers by allowing pixel blending to remain disabled for the entire sequence of render calls; for example, the documentation for the PowerVR SGX graphics hardware explicitly recommends drawing all opaque polygons before any polygons which have blending enabled, to take advantage of the hardware's hidden surface removal capabilities [ImgTec p11].

Since some renderers cannot render, or are inefficient at rendering, polygons of arbitrary vertex count, the polygons are triangulated before being added to their respective meshes. For the opaque polygons generated as described in 2.2 above, every polygon is known to be a convex quadrilateral, so this is a simple matter of dividing the quadrilateral down one of its diagonals and and adding the two triangles so created to the mesh. Boundary polygons, however, typically have a larger number of vertices, so they must be analyzed and appropriately triangulated. There are a number of methods for performing this triangulation, but we use the ear-clipping method, as described in (among others) "FIST: Fast Industrial-Strength Triangulation of Polygons" [Held]. See `AddPolygon` for a pseudocode description of the triangulation algorithm.

3. Results

The effectiveness of this optimization method was verified through application in the iPad port of the PC game *Aquaria,* a 2D-only game which as originally designed rendered its scenes using simple overpainting. As described in the introduction, the iPad's limited pixel operation rate is a significant bottleneck for the overpainting method; indeed, initial trials showed a frame rate[7] on the order of 10 frames per second, not only much slower than the original PC version (which had no difficulty reaching 60 frames per second on even modest hardware) but also slower than the rate of 15 frames per second below which the human eye has difficulty perceiving motion [Reed p24]. The scenes were optimized as described above, and the frame rate of the game with both optimized and unoptimized scenes was measured across a representative set of scenes, as shown in Table 2.

In addition to measuring the frame rate of the game, the total number of graphic objects (*i.e.,* render calls) and triangles in each scene was measured through static analysis of the scene data, and the number of pixels rendered (shown in the table as the equivalent number of full screen overdraws) was measured by drawing the scene using untextured polygons and an additive blend operation, such that each time a pixel was rendered, its color component values were increased by 1.

As can be seen from the table, optimization significantly reduced both the number of graphic objects and the number of pixel operations required to draw each scene, and the frame rate of all scenes improved considerably, in some cases nearly doubling. The final product was locked to 30 frames per second to avoid visual distraction caused by fluctuations in the frame rate, but this frame rate locking would not have been possible without optimizing the scenes.

7 The frame rates referred to in this section are the inverse of the average time required to display one game frame. This includes time spent by the CPU for game logic and other processing unrelated to rendering, but most of this processing runs in parallel with the rendering operations performed by the graphics hardware, and in any case consistently took less than 0.01 seconds per frame; it is thus considered insignificant in this analysis. Indeed, to the extent it affects the results, it reduces the apparent effectiveness of the optimization, and an analysis that looked purely at rendering time would find a greater optimization effect.

Scene	Unoptimized				Meshes Only				Fully Optimized			
	Obj	**Tri**	**OD**	**FPS**	**Obj**	**Tri**	**OD**	**FPS**	**Obj**	**Tri**	**OD**	**FPS**
A1@3050,15500	5801	11782	5.0	22	30	11782	5.0	27	47	25404	3.9	31
C2@5250,14050	2660	7024	6.3	25	140	7024	6.3	28	170	14765	4.3	34
C3@5200,19600	5062	12560	7.2	22	62	12560	7.2	26	81	24772	5.2	31
ET1@2600,7000	1477	3026	5.7	27	30	3026	5.7	31	43	7939	3.9	40
ET4@16750,4000	2185	4430	5.6	25	43	4430	5.6	27	67	10474	4.1	38
F1@9100,20450	4370	9028	4.9	26	26	9028	4.9	35	40	20541	3.1	42
F5@14600,18750	3695	7582	6.3	25	243	7582	6.3	28	262	17066	4.8	37
FV@3350,9800	1895	4102	4.1	34	16	4102	4.1	38	27	9590	2.9	46
IC@15300,13130	2166	4404	8.9	22	10	4404	8.9	25	18	10788	5.6	38
MA@13450,16350	5530	12812	4.9	23	56	12812	4.9	29	82	30178	3.8	32
M1@6350,13700	3975	9546	4.9	30	118	9546	4.9	33	149	22205	2.8	39
NC@5700,5600	888	2508	8.9	21	34	2508	8.9	23	49	7807	5.7	32
OW4@13000,6200	4597	9890	6.8	23	59	9890	6.8	25	75	22081	5.1	30
SC@12370,13880	3333	7146	5.0	26	48	7146	5.0	32	64	16970	3.0	39
ST@14940,17460	804	1656	6.1	27	68	1656	6.1	27	102	4985	3.9	37
VC@5820,3320	2036	4300	5.8	26	65	4300	5.8	31	99	11271	4.4	38
V1@5920,9750	4243	8894	4.1	32	63	8894	4.1	36	82	22770	3.3	38
V3@4900,14050	3727	7610	5.0	27	82	7610	5.0	34	101	18524	3.5	38
W@6230,10230	337	686	8.4	22	51	686	8.4	22	66	2636	4.7	30

Table 2. Rendering characteristics and performance of selected scenes in *Aquaria* on the iPad 1.
The "Meshes Only" column shows data for the scenes with meshes generated as in section 2.3
but without the boundary and opaque polygon optimizations described in sections 2.1 and 2.2.
Abbreviations: "Obj" = graphic objects, "Tri" = triangles, "OD" = overdraws, "FPS" = frames per second

It is worth noting that the iPad's system software (iOS) has also changed since the initial porting work for *Aquaria* was performed. The data in Table 2 was taken under iOS version 5.0.1, but a comparison of the unoptimized scene performance against early development notes suggests an improvement factor of 2-3 compared to the initial tests, which were made under iOS version 3.2. Optimized scene performance does not show a significant change, suggesting that iOS now handles large numbers of graphic objects more efficiently. Accurate data is unavailable because Apple does not allow the downgrading of iPad devices to earlier versions of iOS.

4. Caveats

One limitation of the method as described is that it does not handle images with disjoint parts, such as a single image containing two disconnected circles. *Aquaria* had only one such image, and it was manually split into two separate images to avoid this limitation. A more general solution might be to break each image into one or more subimages and process each subimage separately; in addition to the disjoint-parts case, this solution could also simplify processing for fully connected but complex images, by splitting the image into simpler adjoining shapes. (Care would of course be needed to avoid artifacts at subimage boundaries.)

Another limitation of this method is that it requires non-trivial preprocessing of scene elements which are to be optimized, which hinders optimization of dynamic elements such as game characters. While the images used for such characters could be analyzed and polygon data stored at build time, depth values would have to be assigned at render time based on the position of each dynamic element. One solution might be to reserve a sufficiently large range of depth values for dynamic elements, then trivially assign values from that range to each element based on the elements' rendering order.

The opaque polygon generation algorithm described above works acceptably well for simple images, such as those approximating filled convex polygons; for more complex images, the execution time and the optimality of the generated polygons can deteriorate rapidly. As described earlier, the algorithm's running time is $O(w \cdot h^3)$ in the image size even for simple opaque areas, and this grows exponentially worse if there is any significant branching between opaque horizontal segments. The algorithm's results may also be suboptimal for images with multiple internal holes or non-horizontal edges. One alternative approach might be to use a variation of the boundary polygon generation algorithm, starting with a small, trivial opaque polygon and pushing the vertices outward (instead of pulling them inward) until the polygon's edges reach the boundary between opaque and non-opaque pixels.

5. Conclusion

The method described in this paper was successfully used in the iPad port of the PC game *Aquaria* to considerably improve performance. In initial tests on the iPad 1, many scenes could not even reach 10 frames per second with the traditional overpainting method. The optimized scenes, however, play at a consistent 30 frames per second, providing players with a smoothly animated gameplay experience.

References

[Apple] Apple Inc. "OpenGL ES Programming Guide for iOS" (2010).

[Held] Held, Martin. "FIST: Fast Industrial-Strength Triangulation of Polygons". *Algorithmica* 30(4): 563-596 (2001).

[ImgTec] Imagination Technologies Ltd. "POWERVR SGX OpenGL ES 2.0 Application Development Recommendations". Version 1.8f (2009).

[Reed] Read, Paul; Meyer, Mark-Paul; Gamma Group. *Restoration of motion picture film.* ISBN 0-7506-2793-X (2000).

Appendix A. Pseudocode listing

Readers should note that this pseudocode has been written so as to illustrate the algorithm as concisely as possible; to this end, many optimization opportunities have been deliberately skipped.

```
////////////////////////////////// Section 2 //////////////////////////////////

// Arbitrary thresholds for terminating boundary and opaque polygon
// generation, to strike a balance between quality and rendering cost.
constant MAX_BOUNDARY_VERTICES = 10;
constant MAX_OPAQUE_POLYGONS = 4;
constant MIN_OPAQUE_COVERAGE = 0.75;  // Out of 1.0.

// Analyzes the elements used in a scene, and returns a list of polygon meshes
// which can be used to render the scene more efficiently than simply drawing
// each image in sequence.
GenerateSceneMeshes(scene) {
    images_used ← set();  // Initialize to an empty set.
    for element in scene.elements {  // List of elements in rendering order.
        images_used.add(element.image);
    }
    for image in images_used {
        image.boundary_polygon ← GenerateBoundaryPolygon(
            image, MAX_BOUNDARY_VERTICES);
        image.opaque_polygons ← GenerateOpaquePolygons(
            image, MAX_OPAQUE_POLYGONS, MIN_OPAQUE_COVERAGE);
    }
    return ReconstructScene(scene.elements);
}

///////////////////////////////// Section 2.1 /////////////////////////////////

// Returns a list of vertices representing a boundary polygon for the image.
GenerateBoundaryPolygon(image, max_boundary_vertices) {
    x0, x1, y0, y1 ← FindBoundingRectangle(image);
    vertices ← list((x0, y0), (x1, y0), (x1, y1), (x1, y0));
    while vertices.length < max_boundary_vertices {
        if not AddBoundaryVertex(image, vertices) {  // Modifies vertices.
            break;
        }
    }
    return vertices;
}

// Finds and returns the axis-aligned bounding box of the image.
FindBoundingRectangle(image) {
    x0, x1 ← image.width - 1, 0;
    y0, y1 ← image.height - 1, 0;
    for x, y in image {  // Short for "iterate over each pixel coordinate".
        if image.pixel(x, y).alpha > 0 {
            x0, x1 ← min(x0,x), max(x1,x);
            y0, y1 ← min(y0,y), max(y1,y);
        }
    }
    return x0, x1+1, y0, y1+1;
}

// Attempts to add a vertex to the boundary polygon; returns false if
// no vertex can be added in a way that reduces the polygon's area.
// On success, the vertex list |vertices| is modified.  Note that the
// number of vertices will not necessarily increase on success; the
// function may find a cut which does not require adding a new vertex,
// or even a cut which deletes a vertex.
AddBoundaryVertex(image, vertices) {
    old_vertices ← vertices;
    old_area ← CalcPolygonArea(vertices);
    best_area ← old_area;
    succeeded ← false;

    for i in [0 ... vertices.length - 1] {
        // Try to cut a corner off of the polygon.
        new_vertices ← old_vertices;
        if CutBoundaryCorner(image, new_vertices, i) {
            new_area ← CalcPolygonArea(new_vertices);
            if new_area < best_area {
                vertices ← new_vertices;
                best_area ← new_area;
                succeeded ← true;
            }
        }

        // Try to insert a vertex along edge (i,i+1).  This will
        // typically help when the image has a significant concave
        // portion, such as an L-shaped image.
        new_vertices ← old_vertices;
        if InsertBoundaryVertex(image, new_vertices, i) {
            // Remove any resulting transparent corner.
            if (RemoveTransparentCorner(image, new_vertices, i)
                or RemoveTransparentCorner(image, new_vertices,
                                           (i+2) % vertices.length)) {
                // Ensure no straight angles are left in the polygon.
                do {
                    deleted_one ← false;
                    for j in [0 ... new_vertices.length - 1] {
                        // We assume wraparound indexing for arrays.
                        v1 ← new_vertices[i-1];
                        v2 ← new_vertices[i];
                        v3 ← new_vertices[i+1];
                        if (((v3.x - v2.x) * (v1.y - v2.y))
                            - ((v1.x - v2.x) * (v3.y - v2.y)) == 0) {
                            new_vertices.delete(i);
                            deleted_one ← true;
                        }
                    }
                } while deleted_one;
                // We now have a smaller polygon with no more vertices
                // than the original, which is a guaranteed improvement,
                // so start again with the new polygon.
                vertices ← new_vertices;
                return true;
            }
            // We did not delete a vertex, so compare areas as usual.
            new_area ← CalcPolygonArea(new_vertices);
            if new_area < best_area {
                vertices ← new_vertices;
                best_area ← new_area;
                succeeded ← true;
            }
        }
    }

    return succeeded;
}

// Returns the area of (number of pixels covered by) the polygon described
// by the given vertices.  The returned value is a mathematical estimate
// and may not exactly match the result of rasterization.
CalcPolygonArea(vertices) {
    area ← 0;
    for i in [0 ... vertices.length - 1] {
        j ← (i+1) % vertices.length;
        area ← area + ((vertices[i].x * vertices[j].y)
                       - (vertices[i].y * vertices[j].x));
    }
    return area;
```

```
}

//////////////////////////////// Section 2.1.1 ////////////////////////////////

// Attempts to remove a transparent corner (specifically, a triangle
// containing vertex i and part or all of the adjacent edges) from the
// boundary polygon described by |vertices|.  Modifies |vertices| in place
// and returns true on success; returns false if no cut can be made.
CutBoundaryCorner(image, vertices, i) {
    candidates ← set();

    A ← vertices[i];
    B ← vertices[i-1];
    // For brevity, we only show pseudocode for the case where the edge
    // from A to vertices[i+1] is closer to horizontal than vertical
    // (dx > dy).  For the opposite case (dy > dx), we would step on Y
    // coordinates instead of X coordinates.
    delta_A ← (vertices[i+1] - A) / abs(vertices[i+1].x - A.x);
    // Likewise, here we show the case where edge BA is closer to
    // vertical than horizontal (dy > dx).
    delta_B ← (A - B) / abs(A.y - B.y);

    while A != vertices[i+1] {
        A ← A + delta_A;
        while B != vertices[i] {
            if IsLineEmpty(image, A, B) {break;}
            B ← B + delta_B;
        }
        if B == vertices[i] {break;}
        new_vertices ← vertices;
        new_vertices.insert(B, i);  // Insert B before V...
        new_vertices[i+1] ← A;      // ... and replace V with A.
        candidates.add(new_vertices);
    }

    best_candidate, best_area ← min(
        map(i → (i, CalcPolygonSize(i)), candidates))
    if best_area < CalcPolygonSize(vertices) {
        vertices ← best_candidate;
        return true;
    } else {
        return false;
    }
}

// Returns whether the line segment AB contains only transparent pixels.
// |A| and |B| are 2D vectors and may have non-integral coordinates.
IsLineEmpty(image, A, B) {
    delta ← (B - A) / max(abs(B.x - A.x, B.y - A.y));
    P ← A - delta;
    while P != B {
        P ← P + delta;
        // P will likely contain non-integral coordinates here.  A proper
        // test would ensure that all pixels around P are transparent, to
        // avoid depending on the behavior of a particular renderer.
        if image.pixel(P).alpha > 0 {return false;}
    }
    return true;
}

//////////////////////////////// Section 2.1.2 ////////////////////////////////

// Attempts to insert a vertex on edge (i,i+1) of the boundary polygon
// described by |vertices|.  Modifies |vertices| in place and returns
// true on success; returns false if no vertex can be added.
InsertBoundaryVertex(image, vertices, i) {
    // Pull out vertices for convenience.  The edge being processed is AB;
    // the adjacent edges are CA and BD.
    C ← vertices[i-1];
    A ← vertices[i];
    B ← vertices[i+1];
```

```
D ← vertices[i+2];

// Determine the coordinate delta for the perpendicular search.  We
// assume the vertices are in clockwise order (with respect to the
// pixel grid, with the Y axis vertically inverted from the standard
// Cartesian grid), so this is always 90 degrees clockwise from AB.
perp_vec ← unit(-(B.y - A.y), B.x - A.x);  // Force to a unit vector.

// Look for the point on the edge with the greatest perpendicular
// distance to a non-transparent pixel.  If at any point we hit the
// edge of the texture coordinate range, we give up because we can't
// handle textures with separate parts cleanly.
best_P, best_distance ← (0,0), 0;
// Shorthand for "P iterates over each rasterized point in line segment AB
// except for A and B themselves" (exclusive range).
for P in (A ... B) {
    distance ← 0;
    do {
        distance ← distance + 1;
        Q ← P + distance * perp_vec;
        if not IsPointInsideImage(Q, image) {return false;}
    } while image.pixel(Q).alpha == 0;
    if distance > best_distance {
        best_P, best_distance ← P, distance;
    }
}
if (best_distance == 0) {return false;}

// Use parametric arguments to track the shifted positions of vertices A
// and B, so we can ensure the lengths of edges CA and BD remain positive.
dA ← unit(A - C);
dB ← unit(B - D);
tA ← (A - C) / dA;
tB ← (A - C) / dB;

// Save the direction for shifting vertices A and B; if the vertex is
// a reflex angle, we shift it toward the adjacent vertex, otherwise we
// shift it away.  To determine whether a vertex is reflex, we make use
// of the fact that the cross product of two rays PQ and PR has the
// same sign as the sine of the angle QPR; thus if the cross product is
// negative, the angle is reflex.  For our coordinate system and vertex
// order, we take sequential vertices CAB and measure from AB to AC for
// the angle of vertex A, and similarly for vertex B.
if CrossProduct(B-A, C-A) < 0 {  // Is A reflex?
    d_tA ← -1;
} else {
    d_tA ← +1;
}
if CrossProduct(D-B, A-B) < 0 {  // Is B reflex?
    d_tB ← -1;
} else {
    d_tB ← +1;
}

candidates ← set();
for tP in [1 ... best_distance - 1] {
    P ← best_P + tP * perp_vec;
    while not IsLineEmpty(image, A, P) {
        tA ← tA + d_tA;
        A ← C + tA * dA;
        // If the vertex goes outside the image or the edge becomes
        // zero length, stop looking for candidate polygons.
        if tA <= 0 or not IsPointInsideImage(A, image) {break;}
    }
    if tA <= 0 or not IsPointInsideImage(A, image) {break;}
    while not IsLineEmpty(image, B, P) {
        tB ← tB + d_tB;
        B ← D + tB * dB;
        if tB <= 0 or not IsPointInsideImage(B, image) {break;}
    }
    if tB <= 0 or not IsPointInsideImage(B, image) {break;}
```

```
        new_vertices ← vertices;
        new_vertices[i] ← A;
        new_vertices[i+1] ← B;
        new_vertices.insert(P, i+1);

        // Check that the candidate polygon isn't self-intersecting, and
        // stop processing if it is.  (We assume the original polygon is
        // not self-intersecting, so we only need to check the four new or
        // changed edges against the rest of the polygon.)
        self_intersecting ← false;
        for j in [i + 3 ... i + new_vertices.length - 2] {
            M ← new_vertices[j];
            N ← new_vertices[j + 1];
            if ((N != C and DoEdgesIntersect(M, N, C, A))
                or DoEdgesIntersect(M, N, A, P)
                or DoEdgesIntersect(M, N, P, B)
                or (M != D and DoEdgesIntersect(M, N, B, D)))
            {
                self_intersecting ← true;
                break;
            }
        }
        if self_intersecting {break;}

        candidates.add(new_vertices);
    }

    best_area, best_candidate ← min(
        map(i → (CalcPolygonSize(i), i), candidates))
    if best_area < CalcPolygonSize(vertices) {
        vertices ← best_candidate;
        return true;
    } else {
        return false;
    }
}

// Returns whether the given point is located within the given image's bounds.
IsPointInsideImage(P, image) {
    return P.x >= 0 and P.x < image.width and P.y >= 0 and P.y < image.height;
}

// Returns whether edges (line segments) AB and CD intersect.
DoEdgesIntersect(A, B, C, D) {
    AB, CD ← B - A, D - C;
    determinant ← CrossProduct(AB, CD);
    if determinant == 0 {
        // Parallel lines; check if the lines are coincident, and if so,
        // whether the segments overlap.
        AC ← C - A;
        if AB.x*AC.y - AC.x*AB.y != 0) {return;}  // Not coincident.
        AB_xmin, AB_xmax ← min(A.x,B.x), max(A.x,B.x);
        AB_ymin, AB_ymax ← min(A.y,B.y), max(A.y,B.y);
        return ((C.x in [A.x ... B.x] and C.y in [A.y ... B.y])
                or (D.x in [A.x ... B.x] and D.y in [A.y ... B.y]));
    } else {
        // Intersecting lines; find the parametric point of intersection
        // on each line and see whether it falls within the respective
        // line segment's range (t=[0,1]).  To avoid loss of precision,
        // we skip dividing by the determinant.
        tAB ← CD.y*(C.x-A.x) - CD.x*(C.y-A.y);
        tCD ← AB.y*(C.x-A.x) - AB.x*(C.y-A.y);
        return (tAB >= 0 and tAB <= determinant
                and tCD >= 0 and tCD <= determinant);
    }
}

// Returns the cross product of vectors AB and CD.
CrossProduct(AB, CD) {
    return AB.x*CD.y - AB.y*CD.x;
}
```

```
/////////////////////////////// Section 2.1.3 ////////////////////////////////

// Checks whether the polygon described by |vertices| has a transparent
// triangle formed by any of the following:
//    - vertices i-1, i, and i+1
//    - vertices i-1 and i and the intersection of the line containing
//      edge (i-2,i-1) with edge (i,i+1)
//    - vertices i and i+1 and the intersection of the line containing
//      edge (i+1,i+2) with edge (i-1,i)
// If so, deletes vertex i (modifying |vertices| in place) and returns
// true; otherwise, returns false.
RemoveTransparentCorner(image, vertices, i) {
    M ← vertices[i-2];  // For brevity.
    A ← vertices[i-1];
    B ← vertices[i];
    C ← vertices[i+1];
    N ← vertices[i+2];

    // The vertex can't be part of a transparent corner if it's convex.
    if CrossProduct(C-B, A-B) <= 0 {return false;}

    if IsTransparentTriangle(image, A, B, C) {
        vertices.delete(i);
        return true;
    }

    P = IntersectLineAndEdge(M, A, B, C);
    if P != null and IsTransparentTriangle(image, A, B, P) {
        vertices[i-1] = P;
        vertices.delete(i);
        return true;
    }

    P = IntersectLineAndEdge(A, B, C, N);
    if P != null and IsTransparentTriangle(image, P, B, C) {
        vertices[i+1] = P;
        vertices.delete(i);
        return true;
    }

    return false;
}

// Returns whether the portion of |image| contained in triangle ABC is
// fully transparent.
IsTransparentTriangle(image, A, B, C) {
    for P in [A ... C] {  // Inclusive range.
        if not IsLineEmpty(image, B, P) {return false;}
    }
    return true;
}

// Returns the point of intersection of line AB and segment CD, or null
// if they are parallel or the intersection of the lines is not within
// segment CD.
IntersectLineAndEdge(A, B, C, D) {
    AB, CD ← B - A, D - C;
    determinant ← CrossProduct(AB, CD);
    if determinant == 0 {return null;}
    tCD ← AB.y*(C.x-A.x) - AB.x*(C.y-A.y);
    if tCD < 0 or tCD > determinant {return null;}
    return C + tCD * CD;
}

//////////////////////////////// Section 2.2 /////////////////////////////////

// Ignore opaque polygons smaller than this number of pixels, on the
// assumption that it would take more time to render the additional
// polygons than would be saved by not drawing the covered pixels.
// This value typically depends only on the characteristics of the
// renderer and not on the specific image being processed, so it is a
// constant rather than a parameter to GenerateOpaquePolygons().
constant MIN_OPAQUE_AREA = 64;

// Returns a set of opaque polygons which cover as much of the image as
// possible.  Generation will terminate after either |max_opaque_polygons|
// polygons have been generated, the fraction |min_opaque_coverage| of the
// image has been covered by opaque polygons, or no opaque polygon can be
// constructed that covers an additional |MIN_OPAQUE_AREA| opaque pixels.
GenerateOpaquePolygons(image, max_opaque_polygons, min_opaque_coverage) {
    opaque_segments ← FindOpaqueSegments(image);
    // Map opaque pixels to a value of 1, all other pixels to 0.
    // (This assumes real-valued color components.)
    coverage_map ← map(pixel → floor(pixel.alpha), image.pixels);
    opaque_polygons ← set();
    while (length(opaque_polygons) < max_opaque_polygons
           and OpaqueCoverage(coverage_map) < min_opaque_coverage)
    {
        vertices, covered_area ←
            FindOpaqueQuad(coverage_map, opaque_segments);
        if covered_area < MIN_OPAQUE_AREA {break;}
        FillCoveredPixels(vertices, coverage_map);
        // Push the vertices in by 0.5 pixels on each axis so the
        // edges don't touch any non-opaque pixels when rasterized.
        vertices[0] ← vertices[0] + (+0.5,+0.5);
        vertices[1] ← vertices[1] + (-0.5,+0.5);
        vertices[2] ← vertices[2] + (-0.5,-0.5);
        vertices[3] ← vertices[3] + (+0.5,-0.5);
        opaque_polygons.add(vertices);
    }
    return opaque_polygons;
}

// Returns the fraction of the number of opaque pixels in the image
// which have already been covered by an opaque polygon.
OpaqueCoverage(coverage_map)
{
    num_opaque_pixels ← count(map(value → value != 0, coverage_map));
    num_covered_pixels ← count(map(value → value < 0, coverage_map));
    return num_covered_pixels / num_opaque_pixels;  // Real division.
}

// Marks opaque pixels covered by the given polygon as covered.
// This assumes that |vertices| was returned from FindOpaqueQuad().
// |coverage_map| is modified in place.
FillCoveredPixels(vertices, coverage_map) {
    dx_left ← vertices[3].x - vertices[0].x;
    dx_right ← vertices[2].x - vertices[1].x;
    dy ← vertices[3].y - vertices[0].y;
    num_pixels ← 0;
    for y in [vertices[0].y ... vertices[3].y-1] {
        left ← vertices[0].x + round(dx_left/dy * y);
        right ← vertices[1].x + round(dx_right/dy * y);
        for x in [left ... right-1] {
            coverage_map.value(x, y) ← -1;  // Mark it as covered.
        }
    }
}

//////////////////////////////// Section 2.2.1 ////////////////////////////////

// Returns a set containing all opaque horizontal segments in the image, with
// links between vertically adjacent segments to form a graph from the top to
// the bottom of the image.
FindOpaqueSegments(image) {
    segments ← set();
    for y in [0 ... image.height-1] {
        x ← 0;
        while x < image.width {
            while x < image.width and image.pixel(x, y).alpha < 1.0 {
                x ← x + 1;  // Not an opaque pixel.
            }
            if x >= image.width {break;}  // Reached the end of the line.
            segment ← {x0: x, y: y};
            while x < image.width and image.pixel(x, y).alpha == 1.0 {
                x ← x + 1;
            }
            // Ignore all segments shorter than 3 pixels to avoid having to
            // deal with zero-length segments later.
            if x - segment.x0 < 3 {continue;}
            segment.x1 ← x - 1;  // Inclusive endpoint.
            segment.adjacent_up ← set();
            segment.adjacent_down ← set();
            for other in segments {
                if (other.y == segment.y - 1
                    and other.x1 >= segment.x0
                    and other.x0 <= segment.x1)
                {
                    segment.adjacent_up.add(other);
                    other.adjacent_down.add(segment);
                }
            }
        }
    }
}

//////////////////////////////// Section 2.2.2 ////////////////////////////////

// Returns the quadrilateral covering as many uncovered opaque pixels as
// possible.  The return value is a pair consisting of the quad's
// vertices and the number of uncovered pixels covered by the quad.
// For a return value (|vertices|, |area|), if |area| is not zero, the
// polygon described by |vertices| satisfies the following conditions:
//    - the polygon is a quadrilateral;
//    - the quadrilateral has horizontal top and bottom edges;
//    - vertex 0 has the smallest X and Y coordinates; and
//    - vertices 0 and 1 form a horizontal edge.
FindOpaqueQuad(coverage_map, opaque_segments) {
    candidates ← union(map(
        segment → FindOpaqueQuadRecursive(
            coverage_map, opaque_segments,
            segment, segment, segment, segment),
        opaque_segments));
    if candidates {
        return max(candidates);
    } else {
        return {}, 0;  // Nothing left.
    }
}

// Recursively searches for opaque quads and returns a set of
// (vertices, covered_pixels) pairs for all quads found.
// Takes four segment parameters:
//    - |top| is the segment which will form the top of the quad.
//         (Here, we use "up" or "top" to refer to smaller Y values, and
//         "down" or "bottom" to refer to larger Y values.)
//    - |left| and |right| are the segments which currently limit
//         the left and right edges of the quad; the quad will be
//         internally tangent to the edge of the opaque area at
//         the edges of these segments.
//    - |current| is the segment currently being analyzed.
FindOpaqueQuadRecursive(coverage_map, opaque_segments,
                        top, left, right, current) {
    candidates ← set();
    for next in current.adjacent_down {
        dy_next ← next.y - top.y;

        // Check whether the segment limits the sides of the quad bounded
        // by |top| and |next|.
        new_left ← left;
        new_right ← right;
        if this == top {
            // Special case to avoid division by zero.
            new_left ← next;
            new_right ← next;
        } else {
            // Work out whether the edges would fall outside |next|,
            // rounding outward to make sure we detect all limiting cases.
            dx_left ← left.x0 - top.x0;
            dy_left ← left.y - top.y;
            dx_right ← right.x1 - top.x1;
            dy_right ← right.y - top.y;
            x_left ← top.x0 + floor(dx_left/dy_left * dy_next);
            x_right ← top.x1 + ceil(dx_right/dy_right * dy_next);
            if x_left < next.x0 {new_left = next;}
            if x_right > next.x1 {new_right = next;}
        }

        // Compute the vertices for the new quad.  This time, we round
        // inward to make sure we don't cover any non-opaque pixels when
        // the quad is rasterized.
        dx_left ← new_left.x0 - top.x0;
        dy_left ← new_left.y - top.y;
        dx_right ← new_right.x1 - top.x1;
        dy_right ← new_right.y - top.y;
        x_left ← top.x0 + ceil(dx_left/dy_left * dy_next);
        x_right ← top.x1 + floor(dx_right/dy_right * dy_next);
        // Note the +1 on the right-side X coordinates, since the segment
        // X coordinates are inclusive.
        vertices ← list((top.x0,top.y), (top.x1+1,top.y),
                        (x_right+1,next.y+1), (x_left,next.y+1));
        // Shift the X coordinate for obtuse angles 1 pixel inward, again
        // to avoid hitting non-opaque pixels at rasterization time.
        if vertices[0].x > vertices[3].x {
            vertices[0].x ← vertices[0].x + 1;
        } else if vertices[3].x > vertices[0].x {
            vertices[3].x ← vertices[3].x + 1;
        }
        if vertices[1].x < vertices[2].x {
            vertices[1].x ← vertices[1].x - 1;
        } else if vertices[2].x < vertices[1].x {
            vertices[2].x ← vertices[2].x - 1;
        }

        // If the bottom edge has zero or negative width, we can't create
        // any more quads on this branch, so move on to the next branch.
        if vertices[2].x <= vertices[3].x {continue;}

        // Add this quad and its covered area to the candidate set.
        candidates.add((vertices, CountCoveredPixels(vertices, coverage_map)));

        // Recursively look for deeper quads along this branch.
        candidates ← union(candidates, FindOpaqueQuadRecursive(
            coverage_map, opaque_segments, top, new_left, new_right, next));
    }
    return candidates;
}

// Counts the number of not-yet-covered opaque pixels covered by the
// quadrilateral defined by the given vertices.
CountCoveredPixels(vertices, coverage_map) {
    dx_left ← vertices[3].x - vertices[0].x;
    dx_right ← vertices[2].x - vertices[1].x;
    dy ← vertices[3].y - vertices[0].y;
    num_pixels ← 0;
    for y in [vertices[0].y ... vertices[3].y-1] {
        left ← vertices[0].x + round(dx_left/dy * y);
        right ← vertices[1].x + round(dx_right/dy * y);
        for x in [left ... right-1] {
            if coverage_map.value(x, y) > 0 {num_pixels ← num_pixels + 1;}
        }
    }
}
```

```
///////////////////////////////// Section 2.3 /////////////////////////////////

// Reconstructs a scene using boundary and opaque polygons to optimize
// performance, and returns a list of polygon meshes to be rendered in order.
ReconstructScene(elements) {
    graph ← ElementGraph(elements);
    elements ← SortElements(elements, graph);
    AssignDepthValues(elements, graph);
    return GenerateMeshes(elements);
}

// Returns a directed graph representing the front/back relationships of
// the given list of elements.  For two nodes i and j, the graph has an edge
// from i to j if and only if elements[i] obscures elements[j] as rendered.
ElementGraph(elements) {
    // DirectedGraph(n) (not shown in this pseudocode) is the constructor for
    // a directed graph with n nodes labeled 0 through n-1.
    graph ← DirectedGraph(length(elements));
    for i in [1 ... length(elements) - 1] {
        for j in [0 ... i-1] {
            if DoElementsIntersect(elements[i], elements[j]) {
                graph.AddEdge(i, j);  // Element i obscures element j.
            }
        }
    }
    return graph;
}

// Returns whether the given elements would intersect if rendered.
DoElementsIntersect(element1, element2) {
    // Implementation omitted for brevity.  A simple implementation would
    // transform the image bounding boxes based on the elements' parameters
    // and return whether the transformed boxes intersect; this method has
    // the advantage of being fast, but it will falsely claim that two
    // elements intersect when the intersection area consists entirely of
    // transparent pixels.  A more accurate but slower method would be to
    // check the actual image data and only return true if the intersection
    // area contained non-transparent pixels (with an appropriate margin of
    // error for differing rasterization behavior between renderers).
}

// Returns whether the given elements can be rendered as part of the same mesh.
CanShareMesh(element1, element2) {
    // For the purposes of this pseudocode, we assume that the image used is
    // the sole determinant of whether the two elements can share a mesh.
    // In practice, other factors such as rendering parameters and the use
    // of texture atlases will affect this decision.
    return element1.image == element2.image;
}

//////////////////////////////// Section 2.3.1 ////////////////////////////////

// Sorts a list of elements so that, when possible, elements using the same
// image are adjacent to each other, and returns the sorted list.  This
// routine does not change the appearance of the scene.
SortElements(elements, graph) {
    for i in [0 ... length(elements) - 2] {
        j ← i+1;
        while j < length(elements) {
            while not CanShareMesh(elements[i], elements[j]) {
                j ← j+1;
                if (j >= length(elements)) {break;}
            }
            if j >= length(elements) {
                break;  // No more elements that can share the same mesh.
            }
            can_move ← true;
            for k in [i+1 ... j-1] {
                if graph.HasEdge(j, k) {
                    can_move ← false;
                    break;
                }
            }
            if can_move {
                elements.insert(elements.delete(j), i+1);
                break;
            }
        }
    }
}

//////////////////////////////// Section 2.3.2 ////////////////////////////////

// Assigns depth values (Z coordinates) to each element such that the depth
// range used by the scene is kept as small as possible, but front-back
// relationships between overlapping elements are maintained.  This
// pseudocode assumes that zero is the farthest depth value, that
// increasing depth values are closer to the camera, and that the
// granularity of depth values is no greater than 1.
AssignDepthValues(elements, graph) {
    // Walk the element list, assigning each element a depth value greater
    // than all elements behind it.  We assign in steps of 2 so there is
    // room in front of each element for its opaque polygons.
    for i in [0 ... length(elements) - 1] {
        elements[i].position.z = 0;
        for j in [0 ... i-1] {
            if graph.HasEdge(i, j) {
                elements[i].position.z ← max(
                   elements[i].position.z, elements[j].position.z + 2);
            }
        }
    }
}

//////////////////////////////// Section 2.3.3 ////////////////////////////////

// Returns an ordered list of meshes which can be rendered to produce the
// same result as drawing each element of |elements| in sequence, but which
// requires less time to render.
GenerateMeshes(elements) {
    meshes ← list();
    opaque_meshes ← list();

    for i in [0 ... length(elements) - 1] {
        element ← elements[i];
        if i == 0 or not CanShareMesh(element, elements[i-1]) {
            if current_mesh != null and not current_mesh.IsEmpty() {
                meshes.append(current_mesh);
            }
            // TriangleMesh() (not shown) is the constructor for a mesh object
            // whose polygons are all triangles.  We pass in only the image,
            // but generally there will be  additional rendering parameters
            // such as blend type.
            current_mesh ← TriangleMesh(element.image);
            if not current_opaque_mesh.IsEmpty() {
                // Prepend instead of append, so meshes closer to the front
                // of the scene are rendered first.
                opaque_meshes.prepend(current_opaque_mesh);
            }
            current_opaque_mesh ← TriangleMesh(element.image);
        }
        AddPolygon(current_mesh, element.image.boundary_polygon);
        for polygon in element.image.opaque_polygons {
            // These polygons are always quadrilaterals, so we take a shortcut.
            current_opaque_mesh.AddIndexedVertices(polygon, list(0,1,2,2,3,0));
        }
    }
    if current_mesh != null and not current_mesh.IsEmpty() {
        meshes.append(current_mesh);
    }
    if current_opaque_mesh != null and not current_opaque_mesh.IsEmpty() {
        opaque_meshes.prepend(current_opaque_mesh);
    }

    // Some renderers can precompute pixel visibility regardless of render
    // order; in that case, meshes in the |opaque_meshes| list which share
    // the same image can all be merged into a single mesh to further reduce
    // rendering overhead.

    return opaque_meshes + meshes;
}

// Adds the polygon described by |vertices| to |mesh| as a set of indexed
// triangles.
AddPolygon(mesh, vertices) {
    triangles ← list();
    // Make a copy of the vertex list that we can modify.
    vertices_left ← vertices;
    // Keep track of the original vertex indices corresponding to each
    // element of |vertices_left|, for adding to the triangle list.
    vertex_indices ← list(0 ... vertices.length-1);
    while vertices_left.length >= 3 {
        ear_vertex ← FindPolygonEar(vertices_left);
        triangles.append(vertex_indices[ear_vertex-1]);  // May wrap around.
        triangles.append(vertex_indices[ear_vertex]);
        triangles.append(vertex_indices[ear_vertex+1]);  // May wrap around.
        vertices_left.remove(ear_vertex);
        vertex_indices.remove(ear_vertex);
    }
    mesh.AddIndexedVertices(vertices, triangles);
}

// Finds a vertex of the polygon described by |vertices| whose two
// adjacent edges form an ear of the polygon (see [Held]; we use Held's
// CE2 conditions for ear testing), and returns the index into |vertices|
// of that vertex.
FindPolygonEar(vertices) {
    // Every simple polygon has at least two ear vertices, so if we don't
    // find any ears before the last two vertices, we know without checking
    // that they must both be ear vertices.
    for vertex in [0 ... vertices.length-3] {
        P ← vertices[vertex-2];
        A ← vertices[vertex-1];
        B ← vertices[vertex];
        C ← vertices[vertex+1];
        Q ← vertices[vertex+2];

        // Check that this vertex's (B) angle is convex.
        if CrossProduct(C-B, A-B) <= 0 {
            continue;  // Straight or reflex angle means it's not an ear.
        }

        // Check that the adjacent vertices (A and C) are each inside the
        // cone of the opposite vertex.
        if not IsPointInsideCone(A, B, C, Q) {continue;}
        if not IsPointInsideCone(C, P, A, B) {continue;}

        // Check that no other vertices are contained in the triangle (ABC).
        // A vertex is inside the triangle if it is both inside the ear
        // cone and on the same side of the ear diagonal as the central
        // vertex (B).  Note that technically, we need only check that no
        // *reflex* vertices contained in triangle ABC, but since we don't
        // keep track of which vertices are reflex, it's easier (and only
        // insignificantly slower) to just check every vertex; the end
        // result is the same.
        ok ← true;
        for V in vertices {
            if (V != A and V != B and V != C
                and IsPointInsideCone(V, A, B, C)
                and ArePointsOnSameSide(V, B, A, C))
            {
                ok ← false;
                break;
            }
        }
        if (!ok) {continue;}
    }
}

// Returns whether point P is within cone ABC (the region to the left of
// ray BA and to the right of ray BC, in our coordinate system).
IsPointInsideCone(P, A, B, C) {
    return CrossProduct(P-B, A-B) > 0 and CrossProduct(P-B, Q-B) < 0;
}

// Returns whether points P and Q are on the same side of line AB.
// Points on the line are treated as being on neither side.
ArePointsOnSameSide(P, Q, A, B) {
    return sign(CrossProduct(P-A, B-A)) == sign(CrossProduct(Q-A, B-A));
}

////////////////////////////// End of pseudocode //////////////////////////////
```