Discontinuity Edge Overdraw

Pedro V. Sander
Harvard University
http://cs.harvard.edu/~pvs

Hugues Hoppe
Microsoft Research
http://research.microsoft.com/~hoppe

John Snyder
Microsoft Research
johnsny@microsoft.com

Steven J. Gortler
Harvard University
http://cs.harvard.edu/~sjg

Abstract

Aliasing is an important problem when rendering triangle meshes. Efficient antialiasing techniques such as mipmapping greatly improve the filtering of textures defined over a mesh. A major component of the remaining aliasing occurs along discontinuity edges such as silhouettes, creases, and material boundaries. Framebuffer supersampling is a simple remedy, but 2×2 supersampling leaves behind significant temporal artifacts, while greater supersampling demands even more fill-rate and memory. We present an alternative that focuses effort on discontinuity edges by overdrawing such edges as antialiased lines. Although the idea is simple, several subtleties arise. Visible silhouette edges must be detected efficiently. Discontinuity edges need consistent orientations. They must be blended as they approach the silhouette to avoid popping. Unfortunately, edge blending results in blurriness. Our technique balances these two competing objectives of temporal smoothness and spatial sharpness. Finally, the best results are obtained when discontinuity edges are sorted by depth. Our approach proves surprisingly effective at reducing temporal artifacts commonly referred to as "crawling jaggies", with little added cost.

Additional Keywords: antialiasing, supersampling, triangle mesh rendering, antialiased line rendering.

1. Introduction

Computer graphics has long dealt with the issue of creating discrete images without aliasing [7]. For the hardware-accelerated triangle rendering pipeline, four forms of aliasing can be identified:

- Aliasing within triangle interiors (undersampling of shading function). One such example is aliasing due to texture undersampling, which can be efficiently handled using mipmaps [30] or higher-quality anisotropic filtering. Other shading variations, like pinpoint specular highlights, can exhibit high frequencies that are more difficult to predict and bandlimit.
- Aliasing at triangle edges (appearance discontinuities [7]). We categorize these *discontinuity edges* into *silhouette edges* which limit the extent of the projected surface, and *sharp edges* which mark shading discontinuities due to material boundaries or discontinuities in material attributes like normals and colors.
- Aliasing among triangles (subpixel-sized triangles, also known as the "small object" problem [7]). This problem is partially helped by level-of-detail control. Robust solution requires adequate supersampling or analytic antialiasing.
- Aliasing at triangle intersections (where one triangle passes through another). Static intersections can be preprocessed to yield explicit sharp edges using polyhedral CSG. Dynamic intersections are difficult to antialias without supersampling.

(a) aliased original (b) overdrawn edges (c) final result

Figure 1: Reducing aliasing artifacts using edge overdraw.

With current graphics hardware, a simple technique for reducing aliasing is to supersample and filter the output image. On current displays (desktop screens of ~1K×1K resolution), 2×2 supersampling reduces but does not eliminate aliasing. Of course, a finer supersampling resolution further reduces aliasing but rapidly becomes impractical.

One can implement 2×2 supersampling either by increasing the framebuffer by a factor of 4, or by accumulating 4 subpixel-offset images of the same scene for each frame [16]. Both approaches are costly. The first requires 4 times the framebuffer memory and 4 times the fill-rate. The second requires 4 times the geometry processing, 4 times the fill-rate, and the addition of an accumulation buffer. The impact is that fill-rate-bound rendering becomes up to 4 times slower, and memory capacity is consumed that could otherwise be devoted to storing texture maps or caching geometry.

Much of the aliasing in current hardware rendering occurs along discontinuity edges. Perhaps most objectionable are the "crawling jaggies" that appear near discontinuity edges as a model moves [7]. Such artifacts are perceptible even at high display resolutions where static spatial aliasing is less obvious, and are observable even with 2×2 supersampling. Since discontinuity edges typically cover only a small fraction of pixels, supersampling *every* pixel seems a brute-force solution.

Our approach is to reduce aliasing artifacts along discontinuity edges by overdrawing them as antialiased lines — a feature commonly available in hardware. The z-buffer is used to resolve visibility between the mesh triangles and the overdrawn edges. The number of discontinuity edges is typically much smaller than the number of triangles or pixels, so the overall frame time overhead is small. For improved quality, the discontinuity edges should be sorted, increasing this overhead marginally.

The result of edge overdraw differs from traditional antialiasing methods like supersampling in that one side of each discontinuity edge is "bloated" by a fraction of a pixel. However, the approach succeeds in greatly reducing crawling jaggies and improving rendering quality, as shown in Figure 1.

2. Previous work

Many general techniques to reduce aliasing have been used in computer graphics, including uniform supersampling [9][16], adaptive supersampling [32], analytic prefiltering [5][7][11][14][17][29], and stochastic sampling [6]. Like our approach, adaptive supersampling attempts to focus computation on troublesome areas such as discontinuity edges. However, adaptive supersampling is difficult to make robust and implement in hardware. Prefiltering approaches bandlimit the continuous signal corresponding to a geometric primitive (such as a constant-colored polygon fragment), before actually point-sampling it. They require expensive visibility determinations over areas rather than points. Stochastic sampling methods convert aliasing to less objectionable noise (rather than "jaggies"), but still require oversampling to acceptably reduce aliasing artifacts.

Coverage bitmask approaches [1][4][12][23][27][28] supersample only coverage rather than full *r,g,b,z* samples. These are effective at reducing artifacts at discontinuity edges but fail to eliminate aliasing at triangle intersections, much like our scheme. Like traditional uniform supersampling, they are brute-force solutions since a coverage bitmask must be computed and stored at every pixel (typically 16-32 extra bits). Moreover these schemes maintain a list of fragments projecting onto each pixel.

OpenGL offers a "polygon antialiasing" feature, available on some high-end graphics workstations, that renders polygons with antialiased boundaries [20]. It uses a special blending mode (source_alpha_saturate) and only works when the polygons are sorted front-to-back. A similar feature is also exposed in Microsoft's DirectX API.

Another approach is to only antialias discontinuity edges. Crow [7] proposes tagging discontinuity edges and antialiasing them using prefiltering convolution in a scanline renderer. Bloomenthal [3] infers discontinuity edges in an aliased image as a postprocess. Pixels near discontinuities are then modified to account for coverage of the inferred edges. This method gets confused at texture discontinuities, ignores temporal aliasing, and is likely too expensive to perform at interactive rates.

In silhouette clipping [25], a coarse mesh is clipped to the exact silhouette of a detailed mesh using the stencil buffer. By transferring the stencil to the alpha buffer and redrawing silhouette edges as antialiased lines, the *external* silhouette is antialiased. The current paper borrows two ideas from this work: efficient runtime silhouette extraction and rendering of antialiased lines. However, we avoid using the stencil or alpha buffers, and reduce aliasing at both internal and external silhouettes, and more generally all discontinuity edges.

Sauer et al. [26] sketch a two-pass software rendering approach for antialiasing silhouette edges. The second pass bloats foreground pixels near silhouettes by computing edge coverage at each pixel. Their method handles only silhouettes and detects these by exhaustive search. Their paper lacks details on how polygons are rasterized or how the two passes are composited. Donovan [10] describes a hardware approach that transfers the aliased framebuffer contents into texture memory, and uses this texture to overdraw antialiased edges in a second pass.

Wimmer [31] describes an approach similar to ours in his downloadable viewer (View3DX). He tries overdrawing antialiased lines, but reports that his approach fails without software sorting of all polygons. The DirectX documentation also mentions the use of edge overdraw to achieve antialiasing [18]:

> Redrawing every edge in your scene can work without introducing major artifacts, but it can be computationally expensive. In addition, it can be difficult to determine which edges should be antialiased. The most important edges to redraw are those between areas of very different color (for example, silhouette edges) or boundaries between very different materials. Antialiasing the edge between two polygons of roughly the same color will have no effect, yet is still computationally expensive.

In this paper we describe an edge overdraw approach that effectively reduces aliasing by properly ordering the rendering and using suitable z-buffer settings. We make the approach practical by efficiently detecting and rendering just the discontinuity edges, and introduce methods to maintain temporal smoothness, spatial consistency, and spatial sharpness. We measure performance on a suite of models and demonstrate the resulting quality.

Our method exploits existing hardware capable of rendering antialiased lines, long a subject of computer graphics research [2][8][13][15][19][21][33][34].

3. Approach

Our approach is to first render the triangle mesh (Figure 1a) and then overdraw its discontinuity edges as antialiased lines (Figure 1b-c). In this section, we discuss issues related to rendering the triangle mesh, determining the discontinuity edges, shading these edges, and rendering them.

3.1 Rendering the triangle mesh

The model is rendered as a standard opaque triangle mesh. For efficiency, it is specified as a display list of triangle strips. The z-buffer is used to resolve occlusion, and is saved for use during edge overdraw.

3.2 Determining the discontinuity edges

Recall that the overdrawn discontinuity edges are the union of sharp edges (which mark shading discontinuities) and silhouette edges (which limit the extent of the projected surface).

Because sharp edges demarcate shading discontinuities, this set of edges is static. Therefore, they are collected during a preprocess, and overdrawn at every frame. Fortunately, the number of sharp edges is typically a small fraction of the total number of edges.

Silhouette edges are based on the viewpoint. An edge is a silhouette edge if one of its adjacent faces is frontfacing and the other backfacing. For many meshes, the average number of silhouette edges per view is only $O(\sqrt{n})$, where n is the number of mesh edges. So, typically only a small fraction of mesh edges needs to be overdrawn as silhouette edges.

Collecting the silhouette edges can of course be done in a brute-force manner by checking all mesh edges in $O(n)$ time. To accelerate this process, we use a fast silhouette extraction algorithm whose average running time is proportional to the number of output silhouette edges [24]. During a preprocess, the algorithm constructs a search hierarchy in which nodes represent clusters of mesh edges. Then, for a given viewpoint at runtime, it traverses the hierarchy and is able to quickly skip entire subtrees that contain no silhouette edges.

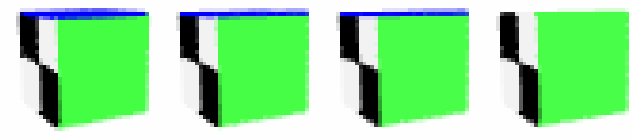
For a closed object, silhouette edges that are *concave* (having an outer dihedral angle ≤ 180 degrees) are always occluded. Therefore, such concave edges need not be entered into the search structure. This typically reduces the number of edges in the structure by 40%. Furthermore, since sharp edges are always

overdrawn, they too are omitted, resulting in an additional reduction of about 10%.

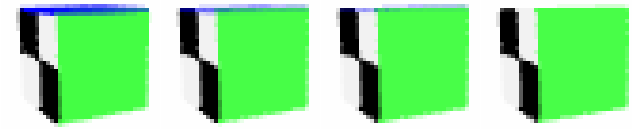
3.3 Shading the discontinuity edges

To shade each discontinuity edge, we use shading parameters (e.g. normals, colors, textures, texture coordinates) taken from the edge's neighboring faces, denoted the *left* and *right* faces. How the shading parameters of the left and right face are combined depends on the category of the discontinuity edge.

We first treat silhouette edges. The case of a non-sharp silhouette edge is simple since the shading parameters of the two adjacent faces agree. At a sharp silhouette edge, the shading parameters of the two faces are different, and the edge must be shaded using the parameters of the *frontfacing* adjacent face. Note that depending on object orientation, a given sharp edge may appear on the silhouette with either the left face frontfacing, or the right face frontfacing.



(a) without edge blending



(b) with edge blending

Figure 2: Unless sharp edges are blended, they can approach the silhouette with the wrong shading, resulting in popping as evident in (a) between the third and fourth panels as the darker top line disappears.

Temporal smoothness: Sharp edge blending

The troublesome case is that of a sharp edge not on the silhouette. To maintain temporal continuity, the edge must somehow smoothly transition to the shading parameters of either the left face or the right face as it approaches the silhouette. Otherwise, abruptly switching the shading parameters from one face to the other would result in a "popping" artifact (see Figure 2).

To solve this problem, for intermediate views where both adjacent faces are frontfacing we shade the edge as a combination of the two faces' shading states. We compute a blend parameter β based on the inner products of the viewing direction with the two adjacent face normals, via

V = eye – edge.midpoint

dotL = V · edge.leftFace.faceNormal

dotR = V · edge.rightFace.faceNormal

β = dotR / (dotL + dotR) .

Shading is then blended using

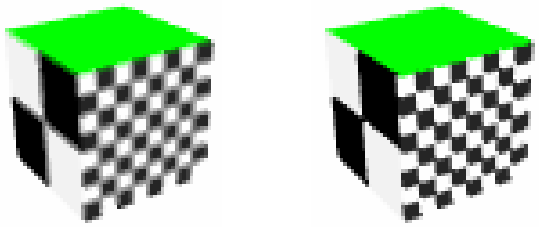
(1 - β) leftShading + (β) rightShading.

To achieve this blending, we have explored two alternate schemes, *blended-draw* and *double-draw*. We will describe both. Note that we prefer the second for its implementation simplicity.

Edge blended-draw. This scheme renders the edge once, as a blended combination of the two shading functions. Ideally, the blending is performed with post-shaded color values. For texture-mapped meshes, this is achieved using hardware multitexturing to blend the two adjacent textures. For Gouraud-shaded surfaces, current hardware does not permit blending of post-shaded results (without resorting to shading on the host CPU). Future hardware supporting programmable shading will permit post-shaded blending. For now, we resort to interpolating the shading attributes (e.g. normals and colors) prior to hardware shading. One drawback is that blending of normals can cause false highlights on sharp crease edges.

Edge double-draw. This scheme renders the antialiased edge twice, once using the shading function of the left face, and once using that of the right face. An opacity value (alpha) is specified for compositing each edge "over" the framebuffer. At least one of the edge renderings must use alpha=1 to prevent the aliased background pixels from showing through. Moreover, the backface shading must be attenuated to zero as the edge approaches the silhouette, to avoid popping. If this backface shading edge is the one drawn with alpha=1, there is no way to eliminate its contribution by rendering the second antialiased line over it (due to the antialiased line's partial coverage). We therefore use a simple order-switching algorithm. Specifically, if $\beta < .5$, we first render with left face shading and alpha=1, followed by right face shading and alpha=β. Otherwise, we first render with right face shading and alpha=1, followed by left face shading with alpha=1-β. Although this results in a slight discontinuity at the $\beta = 0.5$ transition, it is imperceptible in practice.

For blending, we prefer the edge double-draw scheme because it does not require multitexturing and does not exhibit false highlights due to pre-shaded blending. All examples in the paper use this double-draw scheme.



(a) symmetric blending (b) asymmetric blending

Figure 3: Simple symmetric blending blurs discontinuity edges.

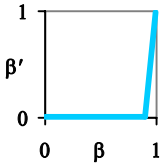
Spatial sharpness: Asymmetric blending

Although blending is needed to avoid temporal popping, it tends to blur the discontinuity edge (see Figure 3), because the shading of the blended edge agrees with neither of the adjacent faces. To compromise between the competing goals of temporal smoothness and spatial sharpness, we adopt a hybrid approach that uses the parameters from a single face (the *left* face) as much as possible, while still avoiding objectionable pops.

We map β through the asymmetric transfer function

$$\beta' = \begin{cases} 0 & \text{, if } \beta \le \tau \\ (\beta - \tau)/(1 - \tau) & \text{, otherwise,} \end{cases}$$

and blend using the resulting β'. We find that with τ set to 0.9, edge transition are still temporally smooth, but the fraction of blended sharp edges drops from about 30% to 2% on average. In addition to restoring edge sharpness and saving blending operations, asymmetric blending allows most of the edge geometry to remain static, possibly cached on the graphics card. We tried to exploit this by first rendering all sharp edges as a display list and then the few blended edges, but did not find any speedup.



Using asymmetric blending, a non-silhouette sharp edge is usually drawn using the shading parameters of the left face. This has the drawback of shifting the proper material boundary by half a pixel. We find that this is less objectionable than the extra softening that occurs when using symmetric blending (Figure 3).

Spatial consistency: Sharp edge orientation

When referring earlier to the left/right faces of an edge, we assumed an edge orientation. If a shading discontinuity consists of several sharp edges along a path and the orientation of each edge in the path is selected independently, then the asymmetric blending bias results in staggered-looking discontinuities (Figure 4a). The solution is to orient discontinuity edges consistently using a simple preprocessing algorithm.

We first concatenate sharp edges together into sharp paths. More precisely, two adjacent sharp edges are placed in the same path if their shared vertex has no other adjacent sharp edges. For each path, we assign an orientation to one edge, and then locally propagate this orientation along the entire path.

If each sharp path is oriented independently, some regular structures appear non-uniform. For example, some patches in Figure 4b appear larger than others. We resolve this using a simple global heuristic. We pick two arbitrary orthogonal vectors, such as $g_1=(2,5,1)$ and $g_2=(2,1,-9)$. For each sharp path, we determine a representative edge as the one whose midpoint is farthest along the vector g_1. We then assign the orientation of this edge based on the sign of the dot product between the edge vector and the vector g_2. Given this first edge orientation, we locally propagate along the sharp path as before. The result is shown in Figure 4c.

(a) staggered discontinuity edges (b) per-path orientation (c) global heuristic orientation

Figure 4: Orienting discontinuity edges. In this close-up of the Cessna, note that the windows have staggered edges in (a) and non-uniform sizes in (b).

3.4 Overdrawing the discontinuity edges

Once shading parameters are determined, edges are rendered into the framebuffer as antialiased lines. Alpha blending is configured so that the lines are drawn using the "over" operation. The z-buffer test is enabled to avoid drawing occluded edges. The z-buffer write is disabled so that chains of antialiased edges do not have gaps at the shared endpoints between individual edges.

Sorting Because the "over" operation is non-commutative, we can occasionally get artifacts when silhouettes lie in front of other discontinuity paths. As described next, we can solve this by sorting the edges in back-to-front order prior to rendering them.

Of all sharp edges, only those on or near the silhouette[1] can occlude other discontinuity edges. Thus, only these need be included in the sort along with the other silhouette edges. The remaining sharp edges are simply drawn first.

We sort the edges according to the distance from the viewpoint to the edge midpoint. Although this midpoint depth-sort heuristic occasionally gives an incorrect sort, artifacts are rare and comprise only a few isolated pixels. By comparison, traditional back-to-front *polygon* rendering requires correct occlusion-based ordering since mistakes there are much more evident.

The sorting step incurs some cost, and is only necessary when there are many discontinuity edge crossings. We therefore report timings both with and without sorting, and demonstrate examples of both on the video.

3.5 Review of final algorithm

Preprocess

```
Collect sharp edges Sharp in scene;
Assign consistent orientations to Sharp;
Construct silhouette extraction tree (excluding sharp & concave);
```

Runtime (given *viewpoint* for each frame)

```
Render scene;
S = ∅;      // set of discontinuity edges to sort
for edge e in Sharp
   d_left = dot(e.f_left.normal , e.midpoint - viewpoint);
   d_right = dot(e.f_right.normal , e.midpoint - viewpoint);
   if d_left < 0 and d_right < 0 then continue; // backfacing
   e.β = d_right / (d_left + d_right);
   if 0.1 < e.β < 0.9 then
      Render e with α = 1.0 using e.f_left shading;
   else
      S = S U {e};
Extract silhouette edges Sil given viewpoint;
S = S U Sil;
Sort S in back-to-front order;
for edge e in S
   if e ∈ Sil then
      Render e with α = 1.0 using e.f_front shading;
   else if e.β < 0.9 then
      Render e with α = 1.0 using e.f_left shading;
   else
      e.β' = (e.β – 0.9) / (1.0 – 0.9);
      if e.β' < 0.5 then
         Render e with α = 1.0 using e.f_left shading;
         Render e with α = e.β' using e.f_right shading;
      else
         Render e with α = 1.0 using e.f_right shading;
         Render e with α = 1.0 – e.β' using e.f_left shading;
```

[1] An edge is declared to be near the silhouette if it has $\beta < 0.1$ or $\beta > 0.9$.

4. Implementation and results

Our software is written using OpenGL. It has been implemented and tested on a Pentium III 800MHz PC with an NVIDIA GeForce2 graphics card. (We have also verified that the method works on an SGI Octane.)

Implementation details. We use `glEnable(GL_POLYGON_OFFSET_FILL)` to perturb z-buffer values of triangles behind those of lines. This is necessary so that antialiased lines pass the z-buffer test to cover the jaggies. For edges adjacent to triangles with high depth slope, we sometimes observe remaining aliasing artifacts, suggesting that the `glPolygonOffset()` feature is not pushing the triangles back sufficiently. The presence of these artifacts varies with the particular graphics hardware.

For efficiency, we only enable `GL_BLEND` for rendering lines. The lines are rendered using the default `glLineWidth(1.0f)`.

When edge sorting is enabled, we use `qsort()`. A faster algorithm like bucket sort could further improve the timing results when rendering high-resolution models.

Results. We tested our system on six models. The preprocessing bottleneck is the creation of the silhouette tree, which is currently unoptimized and can take several minutes on large models. Collecting the sharp edges and assigning them consistent orientations takes only a few seconds.

Runtime results are shown in Table 1. Note that the extracted silhouette edges do not include silhouette edges that are sharp or concave. Rendered edges excludes backfacing sharp edges. The ship example has a higher performance overhead because it is geometry-bound and has a high number of discontinuity edges.

Model	man	plane	stoneh	dino	ship
Faces	1,586	8,000	1,380	43,866	85,068
Edges	2,379	12,000	2,070	65,799	127,602
Sharp edges	354	2,085	1,250	900	19,769
Edge statistics averaged over 100 viewpoints					
Extracted sil. edges	94	393	22	365	7,122
Rendered edges	373	1,727	952	1,894	21,980
Sorted edges	309	1,212	661	1,240	16,448
Blended edges	6	23	10	23	266
Rendering time per frame (in milliseconds)					
No edge overdraw	7.2	9.8	9.6	18.9	40.1
Unsorted edge overdraw	7.7	10.3	10.7	20.0	88.4
Sorted edge overdraw	7.7	10.8	10.7	23.3	121.2

Table 1: Results.

Figure 6 compares traditional aliased rendering, our approach, and 2x2 supersampling. Note that edge overdraw achieves better results than supersampling at edges that are nearly vertical or horizontal. The difference between sorted and unsorted edge overdraw is slight, but is most visible in the second row. The effect of sorting is more prominent in animations (rather than still images), where it reduces some temporal aliasing artifacts. The texture-mapped cube demonstrates the behavior at boundaries between textures.

Figure 7 demonstrates our technique on more complex meshes, using unsorted edge overdraw. The strongest benefit of our approach is its ability to reduce temporal aliasing artifacts, commonly referred to as "crawling jaggies". Unfortunately this cannot be conveyed using static images, so please refer to the accompanying video.

5. Discussion

Surface boundaries. Our scheme easily generalizes to the case of meshes with boundaries. A boundary edge can be thought of as a smooth edge with an outer dihedral angle of 360 degrees. Thus it is reported as a silhouette edge for all viewpoints. Obviously, the edge is shaded using the attributes of its one adjacent face.

With surface boundaries, however, the mesh interior may become visible, so some of our optimizations must be disabled. Concave edges can no longer be omitted from the silhouette search structure, and sharp edges must be drawn even if they are backfacing.

Bloating. Overdrawing edges with antialiased lines extends triangles by a fraction of a pixel along discontinuities (Figure 5). At silhouette edges, this essentially enlarges the foreground object slightly at the expense of the background. This is necessary since the framebuffer lacks information about what lies behind the foreground object at partially covered pixels drawn in the foreground.

For non-silhouette sharp edges, we do have simultaneous shading information for both adjacent faces. Therefore, it should be possible to produce a reasonably antialiased result, given an appropriate but currently unavailable "antialiased double line" hardware primitive.

(a) original aliased (b) 2×2 supersampled (c) edge overdrawn

Figure 5: Comparison of antialiasing at a discontinuity edge. Note that hardware-antialiased edge overdraw often achieves smoother edge filtering than simple 2x2 supersampling.

Bloating is most evident on small features such as thin cylinders, which appear wider and with darker silhouettes. The effect can be reduced through level-of-detail techniques that approximate small geometric features using lines and points [22][24].

Per-object overdraw. For a scene with many objects, edges can be overdrawn after all objects are rendered. Alternatively, edge overdraw can be applied after the rendering of each object. In that case, the objects must be rendered in back-to-front order if one desires correct behavior at object silhouettes.

6. Summary and future work

We describe edge overdraw, an effective method for reducing discontinuity edge artifacts for use in z-buffer hardware rendering. For typical models having a small proportion of discontinuity edges, edge overdraw can be performed with little added cost. While the method is designed for spatial antialiasing, its most striking benefit is the reduction of "crawling jaggies" as demonstrated on the video.

Future work includes finding efficient methods for extracting silhouettes from dynamic meshes, such as view-dependent level-of-detail representations and animated shapes. To solve the "small object" aliasing problem, LOD methods that utilize line and point primitives [22][24] may prove useful.

References

[1] ABRAM, G., WESTOVER, L., AND WHITTED, T. Efficient alias-free rendering using bit-masks and look-up tables. SIGGRAPH 1985, pp. 53-60.

[2] BARKANS, A. High speed high quality antialiased vector generation. SIGGRAPH 1990, pp. 319-326.

[3] BLOOMENTHAL, J. Edge inference with applications to antialiasing. SIGGRAPH 1983, pp. 157-162.

[4] CARPENTER, L. The A-buffer, an antialiased hidden surface method. SIGGRAPH 1984, pp. 103-108.

[5] CATMULL, E. A hidden-surface algorithm with antialiasing. Computer Graphics 12(3), January 1980, pp. 23-34.

[6] COOK, R., PORTER, T., AND CARPENTER, L. Distributed ray tracing. SIGGRAPH 1984, pp. 137-145.

[7] CROW, F.C. The aliasing problem in computer generated images. Communications of the ACM, v. 20, November, 1977.

[8] CROW, F.C. The use of grayscale for improved raster display of vectors and characters. SIGGRAPH 1980, pp. 1-5.

[9] CROW, F.C. A comparison of antialiasing techniques. IEEE Computer Graphics and Applications, v. 1, January, 1981.

[10] DONOVAN, W. Method and apparatus for performing post-process antialiasing of polygon edges. U.S. Patent 6,005,580, December 1999.

[11] FEIBUSH, E., LEVOY, M., AND COOK, R. Synthetic texturing using digital filters. SIGGRAPH 1980, pp. 294-301.

[12] FIUME, E., FOURNIER, A., AND RUDOLF, L. A parallel scan conversion algorithm with anti-aliasing for a general-purpose ultracomputer. SIGGRAPH 1983, pp. 141-150.

[13] FUCHS, H., AND BARROS, J. Efficient generation of smooth line drawings on video displays. Computer Graphics 13(2), August 1979, pp. 260-269.

[14] GUENTER, B., AND TUMBLIN, J. Quadrature prefiltering for high quality antialiasing. ACM Transactions on Graphics, October 1996.

[15] GUPTA, S., AND SPROULL, R.F. Filtering edges for gray-scale displays. Computer Graphics, 15(3), August 1981, pp.1-5.

[16] HAEBERLI, P., AND AKELEY, K. The accumulation buffer: hardware support for high-quality rendering. SIGGRAPH 1990, pp. 309-318.

[17] MCCOOL, M. Analytic antialiasing with prism splines. SIGGRAPH 1995, pp. 429-436.

[18] MICROSOFT. DirectX SDK Documentation. http://msdn.microsoft.com/library/psdk/directx/imover_0lk4.htm.

[19] NAIMAN, A. Jagged edges: when is antialiasing needed? TOG, Oct. 1998.

[20] NEIDER, J., DAVIS, T., AND WOO, M. OpenGL programming guide. Addison-Wesley, 1993.

[21] PITTEWAY, M, AND WATKINSON, D. Bresenham's algorithm with grey scale. Communications of the ACM, 23(11), November 1980.

[22] POPOVIC, J., AND HOPPE, H. Progressive simplicial complexes. SIGGRAPH 1997, pp. 217-224.

[23] RIVARD, B., WINNER, S., KELLEY, M., PEASE, B., AND YEN, A. Hardware accelerated rendering of antialiasing using a modified a-buffer algorithm. SIGGRAPH 1997, pp. 307-316.

[24] ROSSIGNAC, B., AND BORREL, P. Multi-resolution 3D approximations for rendering complex scenes. In *Modeling in Computer Graphics*, B. Falcidieno and T.L. Kunii, Eds. Springer-Verlag, 1993, pp. 455-465.

[25] SANDER, P., GU, X., GORTLER, S., HOPPE, H., AND SNYDER, J. Silhouette clipping. SIGGRAPH 2000, pp. 327-334.

[26] SAUER, F., MASCLEF, O., ROBERT, Y., AND DELTOUR, P. Outcast: programming towards a design aesthetic. 1999 Game Developers Conference, pp. 811-827. (Also available at http://www.appeal.be/products/page1/Outcast_GDC/outcast_gdc_7.htm.)

[27] SCHILLING, A. A new, simple, and efficient antialiasing with subpixel masks. SIGGRAPH 1991, pp. 133-142.

[28] TORBORG, J., AND KAJIYA, J., Talisman: Commodity realtime 3D graphics for the PC, SIGGRAPH 1996, pp. 353-364.

[29] TURKOWSKI, K. Antialiasing through the use of coordinate transformations. ACM Transactions on Graphics, 1(3), July 1982, pp. 215-234.

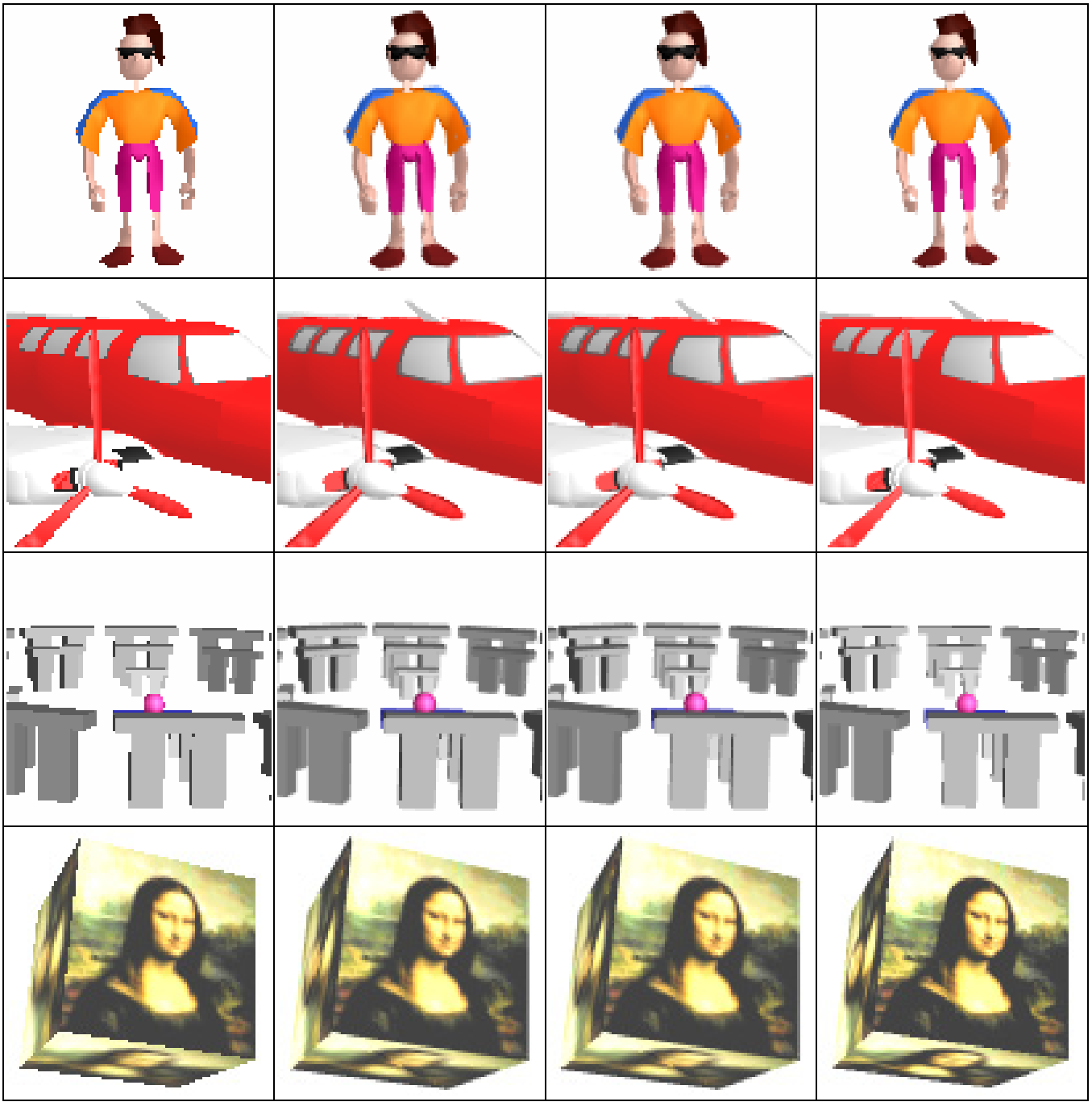
[30] WILLIAMS, L.J. Pyramidal parametrics. SIGGRAPH 1983, pp. 1-12.

[31] WIMMER, M. View3DX software, 1997. http://www.cg.tuwien.ac.at/~wimmer/view3dx .

[32] WHITTED, T. An improved illumination model for shaded display. Communications of the ACM, v.23, June, 1980.

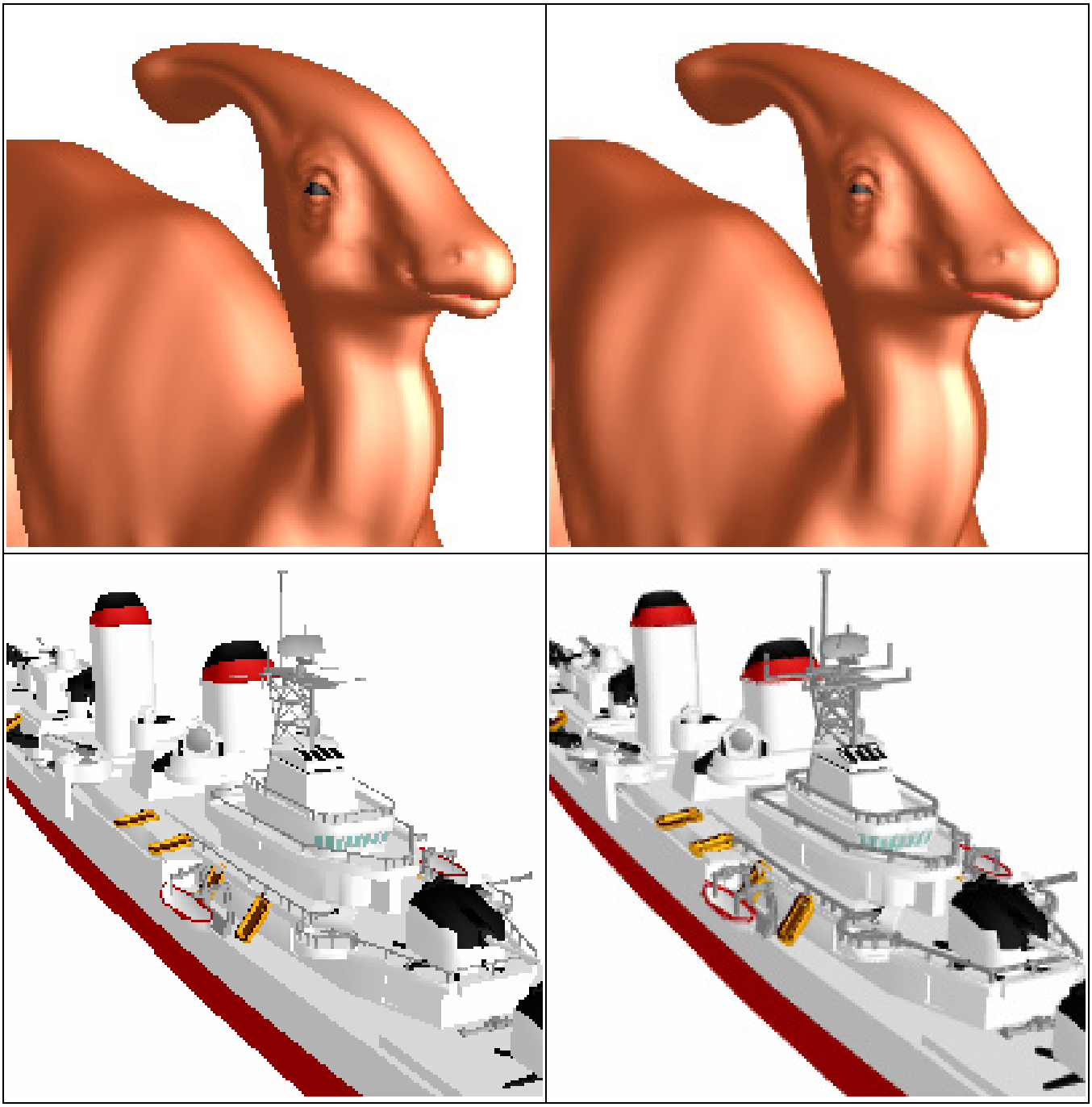
[33] WHITTED, T. Antialiased line drawing using brush extension. SIGGRAPH 1983, pp. 151-156.

[34] WU, X. An efficient antialiasing technique. SIGGRAPH 1991, pp. 143-152.



(a) original aliased mesh (b) with unsorted edge overdraw (c) with sorted edge overdraw (d) 2x2 supersampling

Figure 6: Edge overdraw results. Images (a-c) are rendered at 100x100 resolution, while (d) is rendered at 200x200 and averaged down.



(a) original aliased mesh (b) with unsorted edge overdraw

Figure 7: Results of edge overdraw on more complex meshes, at 200x200 resolution.