Ray Tracing NPR-Style Feature Lines

A.N.M. Imroz Choudhury*
Scientific Computing and Imaging Institute

Steven G. Parker†
NVIDIA Corporation
Scientific Computing and Imaging Institute

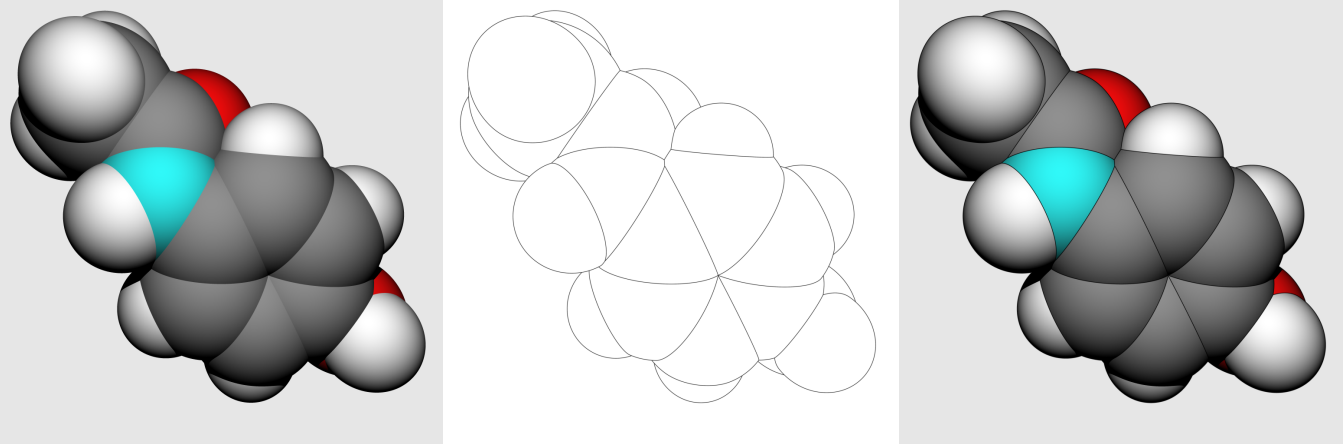


Figure 1: *Acetaminophen molecule rendered with Lambertian shading (left), using our NPR line renderer (middle), and a composite of both renderings (right). Between any two bonded atoms, a line marks the intersection; around the outer edge of the molecule, silhouette edges are visible.*

Abstract

We present an algorithm for rendering high-quality line primitives of controllable on-screen width within a ray tracing framework, which can render simple NPR-style feature lines, including silhouette edges, crease lines, and primitive intersection lines. The algorithm is based on a variant of cone tracing, which measures distances in screen space and is used to detect and render feature lines. This technique opens ray tracing up to previously difficult or impossible styles of rendering, such as mesh visualization, as well as a variety of NPR techniques, such as apparent ridges.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms

Keywords: ray tracing, lines, creases, silhouettes, intersections

1 Introduction and Background

Though the goal of ray tracing and other physically-based rendering techniques is ultimately to produce photorealistic images, it is often helpful to use non-photorealistic rendering techniques [Gooch and Gooch 2001; Strothotte and Schlechtweg 2002] to illustrate or highlight certain features, such as the implied molecular bonds in Figure 1. In NPR techniques, *line primitives* are often used to more clearly express a particular idea or quality. For example, the abstract quality of *confidence* in architectural renderings can be expressed with lines, using sketchier lines to indicate areas in a model requiring further discussion and design [Potter et al. 2009]. Lines are also often used to highlight geometric qualities. Carefully placed lines can express the shape of a complex model [Judd et al. 2007], even in cases when physically-based lighting and material models might obscure subtle details. More generally, lines can be combined with photorealistic rendering to enhance geometric features such as silhouettes and creases [Saito and Takahashi 1990], acting as an additional cue to structure and shape. In these cases and others, line primitives make the abstract more concrete, and the subtle more obvious, and as such they present a definite advantage in rendering systems.

Drawing lines in a raster graphics setting is a primitive operation that uses line rasterization algorithms (e.g., the Bresenham algorithm [Bresenham 1965]). The lines thus produced can be used to highlight features such as sharp corners and silhouette edges. Furthermore, they are drawn on-screen with a particular pixel-width, regardless of the scale of the scene or the current viewing projection. Because three-dimensional geometry is also rasterized to the screen, 3D primitives can be freely mixed with line primitives to produce a variety of illustrative effects.

However, in ray tracing there is no obvious way to "rasterize" a line; instead, all primitives are detected by intersecting camera rays with scene geometry. Because lines are infinitely thin, they are troublesome for the ray tracing algorithm, which operates on "physical" primitives that have at least two dimensions. It is possible to represent lines with, e.g., long, thin pipe-like primitives; however, such primitives have world-space thickness, and therefore change their appearance on-screen as the camera position or zoom level changes.

This paper presents a method for ray tracing feature lines within scenes, using a scale-independent, user-controllable width. The lines thus produced behave much like rasterized lines, maintaining their apparent width when zooming in on a feature, for instance.

*email: roni@cs.utah.edu
†email: sparker@nvidia.com

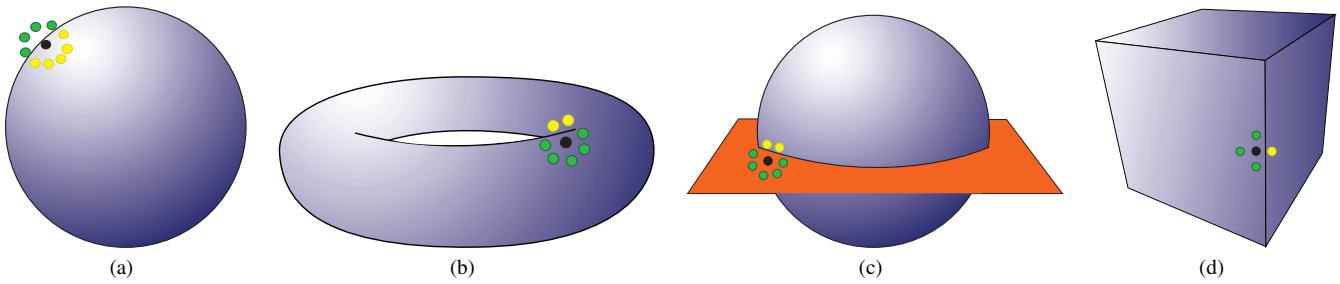


Figure 2: *The different types of feature lines our algorithm captures, illustrated schematically. (a) Silhouette edges, where the edge of an object lies against the background (or a more distant object). (b) Self-occluding silhouettes, where an edge of an object lies against a farther portion of the same object. (c) Intersection lines, where two objects intersect. (d) Crease edges, where an object's normal changes discontinuously.*

Our method demonstrates how a variant of line rasterization can be included in a ray tracer, avoiding the problems of approximating line primitives with physical ones, thus allowing for the inclusion of NPR-style enhancements. Including feature lines in a rendering gives the viewer additional cues to relative positions of objects within the scene, and also enhances particular features within objects, such as sharp corners.

1.1 Feature Lines

Feature lines are linear manifolds that denote geometrically interesting features of objects. The ones this paper focuses on are

- *silhouette edges*, where a primitive's normal vector is perpendicular to the viewing direction;
- *intersection lines*, marking the curves along which two primitives intersect;
- and *crease edges*, indicating curves along which there is a discontinuity in a primitive's normal field (e.g. the sharp corners of a box).

Silhouette edges appear on the "boundaries" of objects, where their surfaces turn away from the viewer and become occluded. They are important visually because they set objects apart from one another. Similarly, intersection lines also set objects apart by marking places where two primitives "run into" each other, creating a seam between them. Our technique does not use the normal and viewing directions to compute silhouette edges, instead relying on the fact that whenever two different primitives occupy a neighborhood of the screen, either a silhouette edge or an intersection line must be present in that neighborhood, and we do not distinguish between the two types of line. Crease edges act more as a cue to an object's internal structure, indicating salient features such as the sharp edges of a box. Such lines enhance the effect of shading on the faces of a box, for instance, to indicate such edges and corners.

The visual cues offered by these feature lines are important when understanding the relative positions of objects in a scene is the primary goal, such as in technical illustrations [Dooley and Cohen 1990], molecular graphics [Tarini et al. 2006], or particle data visualization [Bigler et al. 2006b].

1.2 Ray Tracing, Cone Tracing, and Screen Space

Ray tracing is an image synthesis technique in which a camera shoots rays through an image plane into a scene; each camera ray interacts with scene geometry and may shoot one or more secondary rays to determine the effect of shadows, refraction, reflection, etc.

A ray $\mathbf{R}$ with origin $\mathbf{O}$ and direction $\mathbf{D}$ is defined by the set of points

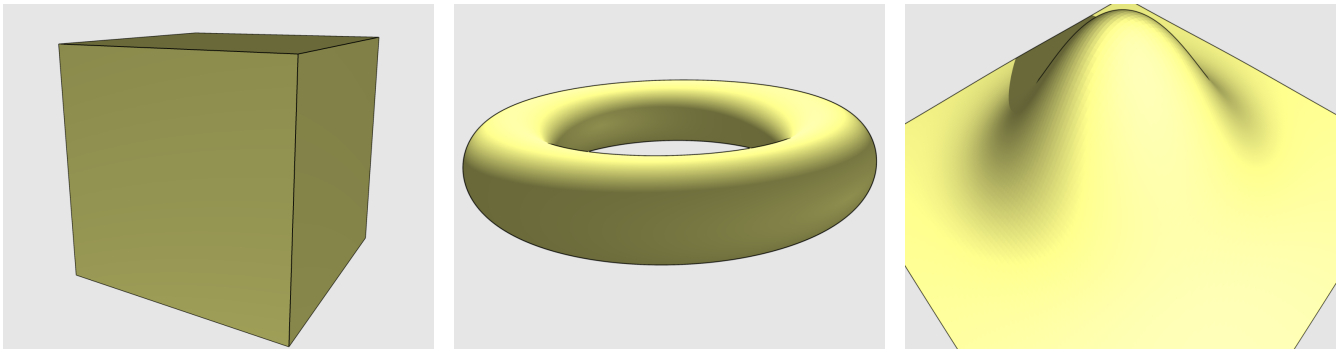
$$\mathbf{R}(t) = \mathbf{O} + t\mathbf{D}, t \in [0, \infty). \quad (1)$$

The parameter t is called the *propagation* or *parametric* distance. When the direction $\mathbf{D}$ has unit length, $\mathbf{R}(t)$ represents a point at distance t units along the ray (starting at the origin point $\mathbf{O}$). Image generation proceeds by constructing a ray to go from the camera into the scene through each image pixel, computing which object in the scene the ray strikes *first* (i.e. which object's intersection with the ray admits the smallest t-value), then computing a color for that pixel by using a shading model, the object's normal vector at the intersection point, and other possible effects such as shadowing, refraction, and reflection. Further details about the ray tracing algorithm can be found in several sources (e.g. [Whitted 1980]).

Ray tracing has several advantages over traditional raster graphics: advanced shading effects such as shadows, specular reflection, and refraction through transparent materials arise as a natural consequence of how rays are traced; its computational complexity scales sublinearly with the number of primitives, making it an attractive choice for very large scenes such as those that can occur in scientific visualization [Stephens et al. 2006]; and in principle it is an "embarrassingly parallel" application that scales well on large machines in practice. The major drawback to ray tracing is the difficulty of producing high-quality images quickly. However, there is much current work on improving the real-time performance of ray tracing [Ize et al. 2007; Wald et al. 2006; Wald et al. 2008].

Furthermore, as discussed above, ray tracing relies on primitives with actual thickness, and as such, a "line primitive" of infinitesimal physical width is impossible. Instead of drawing lines as explicit primitives, however, we use a variation of *cone tracing* [Amanatides 1984] to scout the area in screen space for characteristics of the various feature lines we wish to draw. The basic idea behind cone tracing is to generalize a ray into a cone with its apex at the camera position, widening in such a way that its intersection with the image plane is a circle inscribed within the target pixel. Using a cone instead of a ray gives information about what is happening across the pixel, instead of only in the center of the pixel, and thus can be used to anti-alias the scene, and to compute glossy reflections, translucency, and soft shadows.

We are interested in how a given ray's "neighbors" of some distance in screen space behave. Depending on these rays' behavior, we



(a) (b) (c)

Figure 3: *Examples of crease lines and self-occluding silhouette edges. (a) The shading on the cube faces hints at a discontinuous normal field; the crease lines highlight the location of the discontinuity. (b) The shape of a torus is more strongly expressed when the self-occluding silhouettes extending from the central hollow are shown. (c) A gaussian function is rendered with a self-occluding silhouette, emphasizing the peak.*

may detect a feature line. For example, in Figure 2(c), the center ray (black) has struck the orange quad. But of the surrounding rays (representing the ray's neighbors), some of them (green) also strike the quad but others (yellow) strike the sphere, indicating that the black ray has sampled the scene near an *intersection line*. Because the width of the cone is measured in screen space rather than world space, any lines thus rendered will be drawn with the same width regardless of the nature of the current view of the scene.

Because we are not interested in a general cone that can intersect scene geometry and produce all of the effects mentioned above, we take more of a brute-force approach: we approximate a cone by constructing a *ray stencil* about a particular sample ray which samples a disc in screen space (Figure 4).

1.3 Motivation and Contributions

As ray tracing is quickly becoming a viable rendering method, it pays to investigate its capacity for certain non-photorealistic effects. Though the effects presented in this paper can be achieved using standard image processing techniques, we believe the method is useful to include directly in a ray tracer for several reasons. First, it keeps the implementation simple by eliminating the need to create extended framebuffers and then pass them to other phases of a toolchain. Second, by creating the images first and passing them downstream to another tool, we are stuck with prefiltered framebuffers which may not be as suitable for edge-finding and rendering. As we will show in this paper, our line rendering algorithm takes advantage of the ray tracing framework to produce high-quality lines at "run time," which is more difficult to do if using external tools. Finally, we believe it is valuable in itself to demonstrate that the ray tracing framework is fully capable of rendering non-physical primitives such as rasterized lines.

The central contributions of this paper stem from a novel ray tracing algorithm that is able to detect and render feature lines, with no modification to the underlying scene geometry. We summarize some of the features of this algorithm, with detailed discussion to follow:

- The feature lines drawn are *high-quality* because they are naturally anti-aliased.
- They are of *controllable on-screen pixel-width*, as the user can tune the screen space size of the ray stencil used to detect feature lines.
- Feature line drawing opens ray tracing up to new styles of rendering, e.g. *mesh visualization* (Figure 7), that were not possible before.
- The general aspects of the approximate cone-tracing allows for *porting other NPR techniques* to a ray tracer. As proof of concept, we have implemented *apparent ridges* [Judd et al. 2007].

2 Related Work

The computer graphics literature shows a wealth of line rendering techniques, mostly within the domain of non-photorealistic rendering. Here we review several examples in order to give a flavor of the types of techniques, and to motivate their usefulness in general.

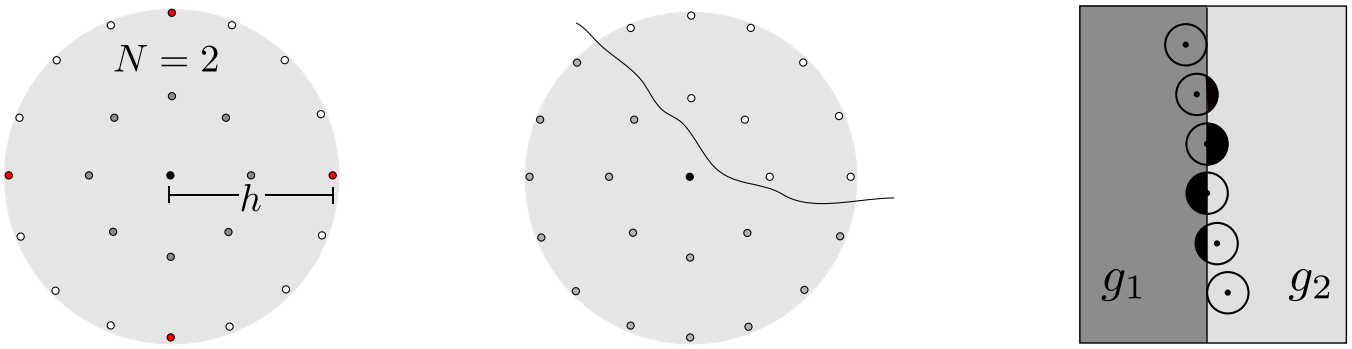
2.1 Wireframe Rendering

Perhaps the simplest way to include NPR-style lines in a rendering is to use a simple two-pass rendering algorithm: first solid geometry is rendered, then the geometry wireframes are overlaid. For certain geometries, this process yields both crease and silhouette edges (as for cubes or hexahedra). Refinements of this basic technique exist, such as a single-pass, fragment-shader based approach that produces high-quality lines with little to no loss in performance [Bærentzen et al. 2006], but such techniques cannot capture other interesting features, such as intersection lines. In addition, for most objects (e.g. tesselated spheres) simply rendering a wireframe will show lines that are not feature lines.

2.2 Feature Line Drawing

When simple wireframe rendering fails, it is necessary to process the scene geometry to compute where the feature lines lie. One general technique for doing so is to collect depth and other values into extended framebuffers, then compute edges, creases, and silhouettes by applying standard image processing techniques [Saito and Takahashi 1990]. An alternative approach is to compute edges directly from the geometry, by finding those edges that separate back-facing from front-facing triangles [Raskar and Cohen 1999].

Several interesting approaches also make use of surface curvature from scene geometry. Suggestive contours [DeCarlo et al. 2003] are silhouette lines that may appear in a slightly different view of



(a) (b) (c)

Figure 4: *The details of how ray stencils work. (a) The ray stencil $D_h^N(s)$ samples a disc of radius h around the sample ray position s (black). The samples lie in N concentric circles (dark gray, light gray/red) about s, where N is the* quality parameter *(in this example, $N = 2$ for two rings of samples). The largest circle has radius h. The red samples indicate a finite difference stencil that can be used to measure gradients in searching for crease edges. (b) A ray stencil being used to measure foreign primitive area. s strikes some primitive below the black line, and a different primitive lies above the line. The stencil (which excludes s) contains twenty four rays ($M = 24$), and nine of them strike a foreign primitive ($m = 9$). Using the linear edge strength metric (Equation 2), which measures how close to* half *of the samples strike a different primitive, we have $e_s = e_M(9) = 62.5\%$. (c) In the limit as $N \to \infty$, a ray stencil becomes a circular disc, minus its center point. The disc moves across a primitive boundary, from geometry ID g_1 to g_2, acting as an* area indicator *for the portion of the filter that lies on a foreign primitive (shaded black, the foreign primitive is g_2 for the top three discs, and g_1 for the bottom three). For the six "snapshots" shown in this example, the disc moves a distance in screen space of $2h$, where h is the disc radius. The foreign area increases from zero to one-half, "flips" to the left side as the center of the filter crosses the boundary, and then decreases back to zero. The foreign primitive area is used to define an edge strength, which in turn is used to render feature lines.*

geometry, and thus may express shape information in the current view; ridge-valley lines [Ohtake et al. 2004] are lines along which the surface curvature is extremal; and apparent ridges [Judd et al. 2007] are similar to ridge-valley lines, but they are drawn where the view-dependent surface curvature is extremal.

2.3 Approaches for Ray Tracing

Silhouette edges can be useful for enhancing geometric features in glyph-based scientific visualization of particle data sets. One approach [Bigler et al. 2006b] adds silhouettes to a ray traced image by first creating a depth buffer (made up of the rays' t-values) and then convolving the depth values with a Laplacian kernel to detect "edges" in the depth image (similarly to the approach of Saito and Takahashi); finally, these values are compared with a threshold to decide where to place the edges on top of the rendered image. By varying the threshold, the silhouettes will capture either groups of objects that are close together, or each object by itself.

In constrast, some approaches use ray tracing directly to compute NPR images, such as in simulation of copper plate images [Leister 1994], in which objects are rendered with their silhouette edges, and their interiors are hatched instead of shaded. The ray intersection points are used to decide how to hatch such objects, with options for combining several hatching styles onto one object. In this case, the authors chose ray tracing as the rendering engine, as it was suited to the underlying algorithm. Our work, on the other hand, specifically strives to integrate line drawing and other NPR effects directly into ray tracing.

Bigler et al.'s approach to rendering silhouettes in a ray traced image is notable as one of the few techniques that combines a raster-style algorithm with a ray traced image. The method presented in this paper aims to demonstrate that, in general, such algorithms can be adapted to work fully within a ray tracing framework. The wealth of NPR techniques for raster graphics shows their usefulness in many situations: we wish to incorporate the general machinery underlying these techniques into the framework of ray tracers.

3 Computing and Drawing Feature Lines

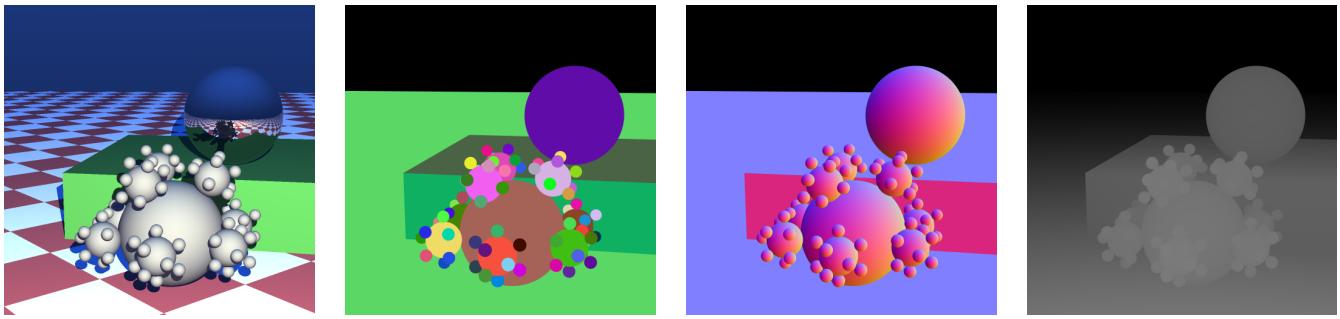
Generally, a ray tracer shoots several *sample rays* from a camera through an image plane into the scene being rendered. In the simplest case, one sample ray is shot through the center of each pixel of the target image. Each sample ray computes a color by interacting with scene geometry (possibly shooting shadow or other secondary rays); the sample colors are combined to yield colors for each pixel in the final rendered image.

3.1 Tracing Ray Stencils

Detecting whether a sample ray strikes the scene near a feature line requires knowing what happens in some neighborhood of the sample ray, i.e., how some cone about the sample ray interacts with the scene. To approximate the cone for a sample ray s, a disc-shaped *ray stencil* $D_h^N(s)$ is constructed about s with radius h in screen space, and using a quality parameter N. The parameter N determines the number of rays M in the stencil (note that M does not count the sample ray itself). Figure 4(a) describes the construction of $D_h^N(s)$.

Within the disc sampled by the ray stencil, feature lines can be found as follows. s strikes some primitive in the scene, or else it strikes the background: in either case, it is associated with a *geometry ID* g_s describing what it has struck,[1] and a parametric distance

[1] In the simplest setup, the geometry ID is identical with the "primitive ID" of the primitive the ray has struck. However in some cases it may be more meaningful to associate a more complex object made up on simpler primitives (e.g. a ray striking a meshed object will actually strike a triangle primitive, but the geometry ID can instead reference the mesh itself), hence the use of the term *geometry ID*. Both primitive IDs and geometry IDs can usually be identified with the unique pointer value specifying the appropriate object in memory. Geometry IDs, when organized into a full image (as



(a) (b) (c) (d)

Figure 5: *A scene rendered (a) normally, (b) with geometry IDs, (c) with colormapped normals, and (d) with depths (i.e. t-values). The method presented in this paper essentially searches for zeroth and first order discontinuities in these images to compute feature lines.*

t_s to the intersection point. Each stencil ray $r \in D_h^N(s)$ is traced and associated with its own geometry ID g_r and parametric distance t_r.

Let m be the number of rays r in $D_h^N(s)$ for which $g_r \neq g_s$ (i.e. the number of rays striking geometry *different* from that struck by s). Depending on the value of m, one of the following situations results:

1. *The stencil straddles different geometry IDs* ($0 < m \leq M$). Some of the stencil rays strike different objects than the sample ray does so the sample ray is near either a silhouette edge or an intersection line (Figure 1 demonstrates both types).

2. *All the stencil rays strike the same geometry as the sample ray* ($m = 0$). When the entire ray stencil strikes the same object, the possible feature lines are crease edges and self-occluding silhouettes. A crease edge occurs when the surface normal changes discontinuously (Figure 3(a)). Numerically, we define a crease edge occurring on any sample whose ray stencil indicates a very large magnitude normal gradient. The normal gradient is computed using the finite difference stencil contained withing the ray stencil (Figure 4(a)). If the gradient is larger than some threshold, then the sample lies near a crease edge.

   If a crease edge is not found, then the sample must be checked for a self-occluding silhouette (Figure 3(b,c)). These occur when some of the stencil rays strike the object at a significantly farther parametric distance than the sample ray does. Define d to be the number of stencil rays r for which $|t_r - t_s| > T$ (where T is a threshold that depends on the particular scene and primitive, but can usually be set to some fraction of t_s, allowing for some view dependence on how the silhouette is drawn). If $d > 0$ then the sample lies near a self-occluding silhouette.

When the feature type is determined, an *edge strength metric* is used to compute how dark the associated feature line should be drawn. The stronger the edge, the darker it will be drawn. An edge strength metric is a function e_M that maps some count of stencil rays to an edge strength lying between 0 and 1. The edge strength e_s for the sample ray s is $e_M(m)$ if the feature is a silhouette or intersection, $e_M(d)$ if it is a self-occluding silhouette, and 1 for a crease edge.

A simple example of an edge strength metric, which is used for the examples in this paper, is

$$e_M(i) = 1 - \frac{|i - \frac{1}{2}M|}{\frac{1}{2}M}. \tag{2}$$

This function rises linearly with i from zero to one for $i \in [0, \frac{1}{2}M)$, and then decreases linearly back to zero for $i \in (\frac{1}{2}M, M]$. It reflects the fact that when a ray stencil is situated with half its area in one geometry region and half in another, it is measuring the strongest possible edge between two regions (Figure 4(c)). In this case, $m = \frac{1}{2}M$, and $e_M(m) = 1$. The value of this function is used to blend the sample color c_s with black, thus rendering feature lines.

The result of tracing and shading a sample ray, and then tracing the associated ray stencil, is therefore a sample color c_s and an edge strength e_s.

3.2 Feature Line Rendering Algorithm

To find and draw feature lines, we use a modified version of the ordinary ray tracing algorithm. For a particular sample ray, the algorithm runs as follows: the sample ray s is traced and shaded as normal with color c_s. A ray stencil $D_h^N(s)$ is constructed about s and the stencil rays are traced until they strike the scene geometry. Depending on m, one of the two cases in Section 3.1 is triggered, and an edge strength e_s is computed. The sample color c_s is blended with black, using the edge strength as an interpolation factor, yielding a darkened color $c_s(1 - e_s)$, and this color replaces the original sample color. For full edge strength, the sample will be black and for zero edge strength it will be shaded as normal. Between these extremes lies a spectrum of darkened colors, usually serving as a "halo" for the darkest part of the line. By changing the radius of the stencil h, the width of the line can be varied (demonstrated in Figure 10). Furthermore, because edge strength determines darkness, the lines are inherently anti-aliased via prefiltering (in much the same way as the wireframe technique discussed above [Bærentzen et al. 2006]).

4 Discussion

4.1 Ray Stencils and Image Filtering

The idea of a "ray cone" and its approximation with ray stencils is the central concept of our method. By arranging the ray stencil in screen space with a fixed radius, and allowing the rays the compute

in Figure 5(b)) are similar to the *ID reference image* used to compute surface visibility in certain applications [Kowalski et al. 1999; Northrup and Markosian 2000].

scene information, a projection of the scene onto the disc sampled by the stencil is accomplished. Through the stencil ray t-values, geometry IDs, and normal vectors, the projection actually produces three images: a geometry identification image, a depth image, and a normal image (Figure 5).

In computing the silhouette and intersection lines, the counting process described in Section 3.2 is essentially an *area estimation*, taking the area of the stencil covering "foreign" geometry IDs to be a measure of how strong of an edge lies within the disc. The SUSAN edge detector from image processing [Smith and Brady 1997] works in a similar manner, finding edges in an image at the centers of circular regions in which the pixel intensity across the circle varies significantly from the intensity at the circle-center pixel.

As constructed, the ray stencils contain a first-order finite difference stencil. This is used to compute the gradient of the normal image, discontinuities of which indicate crease edges. First-order discontinuities in the normal are related to second-order discontinuities in the depth image (i.e. sudden changes in the normal direction are accompanied by sudden changes in the rate of change of depth values with respect to some viewing direction).

For each type of feature line our algorithm detects, the action of the ray stencil can be explained in terms of searching for zeroth, first, or second order discontinuities in one function or another, along the same lines as discussed in, e.g., [Saito and Takahashi 1990]. Our method demonstrates that general image filtering techniques are possible, fully within the ray tracing framework.

4.2 Anti-Aliasing

One particular advantage of our method is that the lines drawn are naturally anti-aliased. This arises from the nature of the area estimation performed by the ray stencils. Ideally, as a ray stencil moves across a line-like feature in image space, the foreign geometry ID area increases from zero to fifty percent, and then falls back to zero again. The edge strength metric derived from these values, when used to determine darkness, produces a line that is dark in the middle and smoothly lightens toward white at distances equal to the radius of the stencil. By increasing the quality parameter N in the stencil construction, the area estimation becomes finer and more levels of smoothness can be used at the cost of tracing more stencil rays.

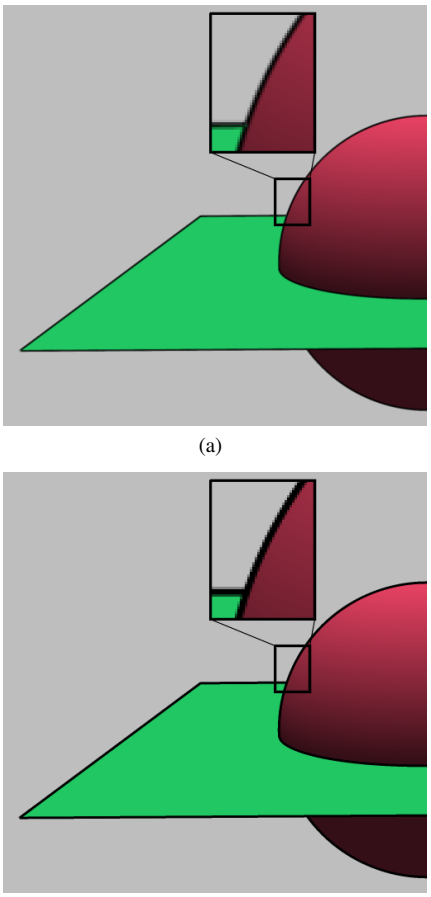
If lines with different qualities are needed, different edge strength metrics can create different kinds of lines. For instance, exponentiating the second term of Equation 2 will produce an edge strength metric with a faster transition from light to dark, giving bolder, sharper-edged lines (Figure 6):

$$e_M(i) = 1 - \left(\frac{|i - \frac{1}{2}M|}{\frac{1}{2}M} \right)^{10}. \quad (3)$$

The stencil rays can also be used for scene anti-aliasing. Because the stencil rays are used for essentially an image processing task that depends on visibility and depth, they are intersected with scene geometry but not shaded. However, traversing and intersecting rays with the scene is the dominant cost in tracing them; for a small extra cost the stencil rays can be shaded and used for multisampling, in addition to computing feature lines (Figure 10(a,c,e)).

5 Results

The technique presented in this paper opens ray tracing to new styles of rendering. In this section, we review several examples of the technique.



(a)

(b)

Figure 6: *The same scene rendered using a (a) linear edge strength metric and (b) exponential edge strength metric. Note that the exponential line is bolder, with a faster transition from clear to black. Because of the quicker transition, these lines may require supersampling of the scene to achieve high quality (both of these scenes are rendered with 9 samples per pixel).*

5.1 Primitive joints

Figure 1 shows a space-filling model [Corey and Pauling 1953] of an acetaminophen molecule, with NPR lines shown by themselves in the middle panel. In the space-filling model, spheres representing atoms always intersect to show atomic bonds; marking the lines along which the atoms intersect can make the structure subtly more apparent, aiding in the understanding of such images. Techniques for approximating global illumination, such as ambient occlusion, have been shown to be useful for certain visualization applications, including particle data [Gribble and Parker 2006] and molecule rendering [Tarini et al. 2006]. These methods are especially helpful in understanding subtle three-dimensional placement. One of the effects of using ambient occlusion for a molecular model is to *darken* the atomic joints; in this case, drawing the intersection lines serves as a non-photorealistic way to indicate the darkened regions, evoking some of the core effect of the ambient occlusion renderings. Rendering intersection lines directly generalizes the darkening effect by drawing lines even for primitives that intersect at shallow angles, in cases where ambient occlusion would *not* significantly

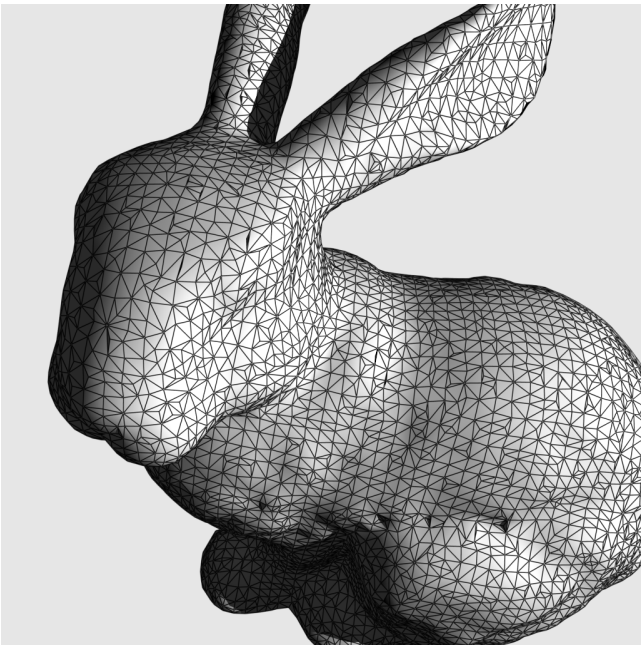


Figure 7: *Our method reveals intersection lines between the triangles in the bunny model, allowing for the visualization of meshes.*

darken the joint between them.

5.2 Mesh visualization

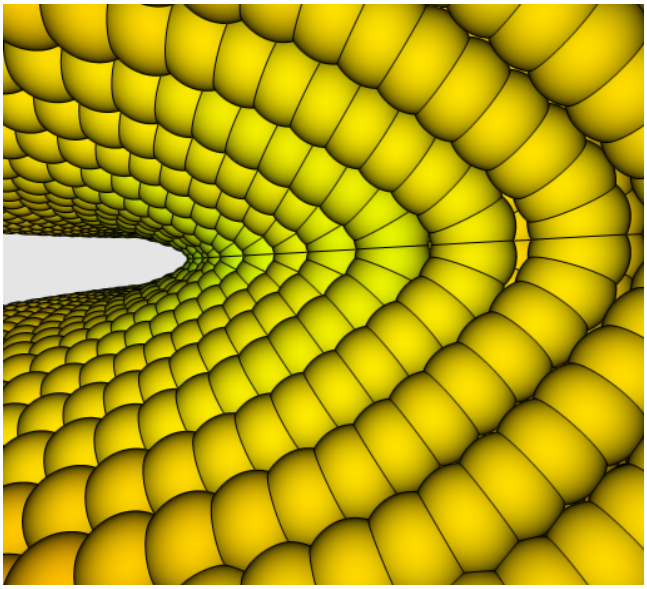
Triangle meshes are a standard and widespread way to represent three-dimensional geometry. In a raster graphics setting, wireframe techniques can reveal the mesh structure by showing the triangle joints. This can be useful for many purposes, such as debugging mesh model geometry or evaluating triangle quality during design and construction.

When ray tracing a mesh model (Figure 7), our technique offers two choices for the stencil ray geometry IDs: either the ray can associate with the object representing the whole object, or with the individual triangle primitive within the model. In the former case, the technique treats the mesh as a single, whole object, and it will show the silhouette of the meshed object.[2] In the latter case, however, the joints between the triangles will also be drawn, revealing the mesh structure and connectivity.

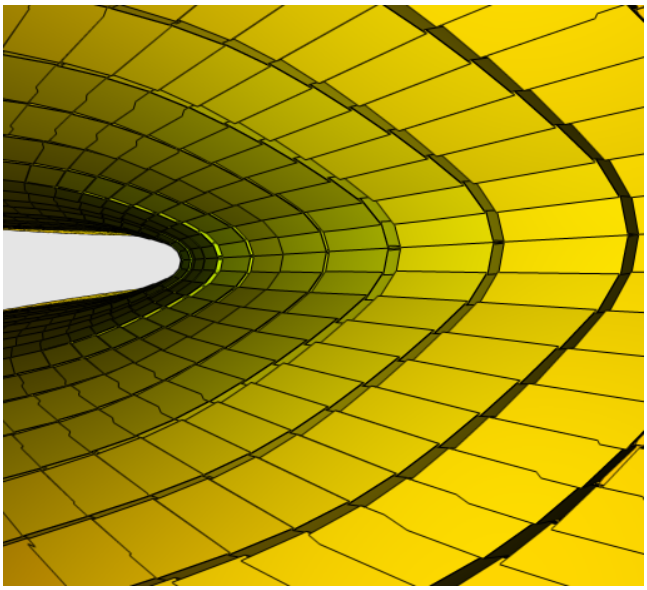
5.3 Particle data sets

Advanced shading models and NPR techniques have been shown to be useful in examining particle data sets produced by the Material Point Method (MPM) [Bigler et al. 2006b]. MPM simulates solid mechanics by treating objects as collections of particles, each of which represents a small piece of material; the particles move in response to applied forces, deforming the objects they compose [Sulsky et al. 1995]. The data sets are usually visualized using a glyph to represent each particle; important insights come from understanding how the particles are arranged and how their arrangement changes over simulated time.

In such data sets, silhouette edges can act as a cue to structure. Figure 8 shows two renderings of a particle data set, each using a different glyph geometry. By using the technique described in this paper, we highlight not only silhouette edges but also places where particles intersect (Figure 8(a)), while crease edges enhance the individual hexahedral glyph shapes (Figure 8(b)). Seeing where particles overlap is especially important for MPM data, as it usually indicates error or instability in the simulation. As with the molecule rendering (Figure 1), the NPR lines elaborate the primitives' physical relationship to each other, which is important to understand in a setting like scientific visualization.



(a)



(b)

Figure 8: *A particle data set rendered using (a) spheres and (b) hexahedral elements. In both images, our NPR technique is used to render intersection lines, silhouette edges, and crease edges. The intersection lines make clear the relative positions of the glyphs. In particular, (a) each sphere is seen to overlap its upper and lower neighbor, while (b) the hexahedral glyphs intersect very slightly where the boundary lines appear not to be straight, indicating misaligned glyphs and perhaps error in the simulation.*

[2] If the normals are interpolated across the triangle faces for shading purposes, then the technique will not pick up sharp corners at the triangle edges.

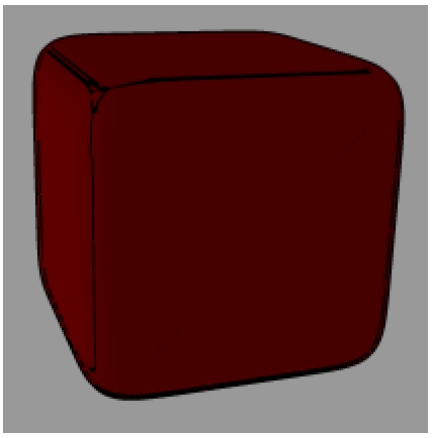


Figure 9: *A rounded cube overlaid with apparent ridges, as computed by a ray tracer using ray stencils.*

	No lines	$N = 1$	$N = 2$	$N = 3$
No anti-aliasing	24.3	3.1	1.9	1.0
Stencil AA	N/A	2.1	0.9	0.4
True multisampling	N/A	0.4	0.1	0.02

Table 1: *Frame rates for single-thread runs over several quality parameters and choices of anti-aliasing strategies.*

	No lines	$N = 1$	$N = 2$	$N = 3$
No anti-aliasing	147.0	30.1	14.3	7.8
Stencil AA	N/A	16.7	6.8	3.5
True multisampling	N/A	3.5	0.6	0.2

Table 2: *Frame rates for eight-thread runs over several quality parameters and choices of anti-aliasing strategies.*

5.4 Other NPR Techniques

To demonstrate that our framework of ray stencils is useful for other NPR algorithms as well, we have implemented *apparent ridges* [Judd et al. 2007]. Figure 9 shows a "rounded cube" (actually a superellipsoid) on which apparent ridges, which are the loci of points at which the "view dependent curvature" is locally maximal, can be seen.

The view dependent curvature is described by a rectangular matrix Q that depends both on an object's surface curvature, and the viewing projection. It transforms vectors in screen space to normal perturbation vectors in world space: in terms of ray stencils, this means measuring the normal gradient in screen space is sufficient to construct Q. The singular value decomposition of Q yields the view dependent curvature for each sample ray, which in turn can be used to render the apparent ridges.

This example demonstrates that the ray tracing framework, together with ray stencils, is flexible enough to allow for "porting" NPR techniques from a raster graphics setting to a ray tracer. Given the large number of techniques discussed in Section 2, the apparent ridges example shows that it is possible to use NPR methods within ray tracers.

6 Performance and Interactivity

We have implemented our line rendering algorithm within Manta [Bigler et al. 2006a], an interactive ray tracer whose flexible design supports various ray tracing algorithms. Tables 1 and 2 show some performance data to give a sense of the possible levels of interactivity in the line renderer. The numbers in Table 1 show frame rates from a single-thread run on a 2.8 GHz Intel Core 2 Duo machine, while Table 2 represents an eight-thread run on a Dual 3 GHz Quad-core Intel Xeon; in both cases, the acetaminophen molecule shown in Figure 1 is being rendered. Each column represents a fixed quality parameter N over three different runs: one with no anti-aliasing performed, one in which the stencil rays double as extra scene samples, and one in which an equivalent number of multisamples is cast and used to anti-alias.[3]

For the purposes of this discussion, we define an *interactive frame rate* to be one that is above 1fps. Though somewhat arbitrary, this threshold is chosen with the idea that frame rates falling below it quickly make it difficult to even reposition the camera reliably, hence putting the user into a *non-interactive* regime.

We note that for the common case of a single-threaded run on common desktop hardware (Table 1), the base example of a ray-traced scene with no anti-aliasing runs at an interactive, as do four of the other examples, while for combinations of true multisampling with $N \geq 2$, performance falls into non-interactive frame rates. These numbers strongly reflect the primary tradeoff in using our line renderer: achieving high-quality images requires large amounts of computation. However, because lower-fidelity sessions run interactively, it is possible to prototype images and experiment with camera views and scene composition before settling on a set of parameters and producing the high-quality final image. In addition, the returns in line quality diminish quickly for increasing values of N (the images in this paper are rendered for the most part with $N = 1$). Large values of N may only be necessary when the lines being drawn are very thick, or if rapidly changing edge strength metrics (such as the "exponential" lines defined in Equation 3) are used.

On the other hand, if even a few more computing resources are available, the situation becomes brighter. For the eight-thread run on a dual quad-core machine (Table 2), all of the frame rates improve, pushing four more examples into interactive frame rates. Generally speaking, desktop machines are able to run our algorithm at interactive rates, with more computing time required for only the highest quality images. The ability to run interactively is important, as much of the utility of our method comes from the added understanding possible from seeing feature lines.

7 Conclusions and Future Work

We have presented a method for computing and rendering feature lines, using only the machinery of a ray tracing engine. The method searches for indications of such feature lines in the screen space vicinity of a sample ray, allowing for the ray tracer to render custom-width lines, emulating methods that already exist in the raster graphics literature. The technique is useful for drawing attention to specific geometric features, such as the relative placement and grouping of primitives, which can promote better comprehension of certain images.

One major area of future work for our method lies in searching for optimizations. Currently, we trace stencil rays independently; however, because stencil rays are, by definition, more likely to be coherent, there should be ways to leverage that coherence to compute their scene intersections more quickly. In addition, because of the stencil rays' geometric regularity, there may be opportunities

[3] e.g. $N = 1$ gives eight stencil rays plus one sample ray, so the "True multisampling" case uses nine samples per pixel; in general, a quality parameter N uses $(2N + 1)^2$ rays (including the sample ray).

for re-using results from previous instantiations of ray stencils as well.

We may also be able to reduce computation and increase the accuracy of our results by analytically intersecting a cone with the scene and noting how it interacts with scene primitives. This would be a major generalization of our use of stencils and it is not clear whether there is a good way to do so.

Currently we use a numerical threshold for deciding whether a large-magnitude normal gradient represents a discontinuity. Because of the all-or-nothing nature of this operation, we are forced to set the edge strength for all detected crease lines to 1. It is not clear how to make the threshold criterion continuous in a reasonable manner, but doing so would increase the quality of crease edges.

It remains an open question for us whether our method should render feature lines in reflected and refracted images; currently we have opted not to extend the method to secondary rays in order to reduce computation. However, if such effects are need, it is a straightforward extension of the same algorithm described in Section 3 to secondary stencils all striking the same reflective or refractive object.

Ray tracing is a useful technology, but until now the techniques of the NPR community have largely been alien to it. By demonstrating feature lines and the possibility of expressing traditional NPR algorithms in a ray tracer, we hope to see both the ray tracing and NPR communities benefit.

References

AMANATIDES, J. 1984. Ray tracing with cones. *Computer Graphics 18*, 3 (July), 129–135.

BÆRENTZEN, A., NIELSEN, S. L., GJØL, M., LARSEN, B. D., AND CHRISTENSEN, N. J. 2006. Single-pass wireframe rendering. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, ACM, New York, NY, USA, 149.

BIGLER, J., STEPHENS, A., AND PARKER, S. 2006. Design for parallel interactive ray tracing systems. 187–196.

BIGLER, J., GUILKEY, J., GRIBBLE, C. P., HANSEN, C. D., AND PARKER, S. G. 2006. A case study: Visualizing material point method data. In *Proceedings of Euro Vis 2006*, 299–306, 377.

BRESENHAM, J. E. 1965. Algorithm for computer control of a digital plotter. *IBM Systems Journal 4*, 1, 25–30.

COREY, R. B., AND PAULING, L. 1953. Molecular models of amino acids, peptides, and proteins. *The Review of Scientific Instruments 24*, 8 (August), 621–627.

DECARLO, D., FINKELSTEIN, A., RUSINKIEWICZ, S., AND SANTELLA, A. 2003. Suggestive contours for conveying shape. *ACM Trans. Graph. 22*, 3, 848–855.

DOOLEY, D., AND COHEN, M. F. 1990. Automatic illustration of 3d geometric models: lines. In *SI3D '90: Proceedings of the 1990 symposium on Interactive 3D graphics*, ACM, New York, NY, USA, 77–82.

GOOCH, B., AND GOOCH, A. 2001. *Non-Photorealistic Rendering*. A.K. Peters.

GRIBBLE, C. P., AND PARKER, S. G. 2006. Enhancing interactive particle visualization with advanced shading models. In *APGV '06: Proceedings of the 3rd symposium on Applied perception in gr aphics and visualization*, ACM Press, New York, NY, USA, 111–118.

IZE, T., SHIRLEY, P., AND PARKER, S. 2007. Grid creation strategies for efficient ray tracing. In *IEEE Symposium on Interactive Ray Tracing, 2007*, 27–32.

JUDD, T., DURAND, F., AND ADELSON, E. H. 2007. Apparent ridges for line drawing. *ACM Trans. Graph. 26*, 3, 19.

KOWALSKI, M. A., MARKOSIAN, L., NORTHRUP, J. D., BOURDEV, L., BARZEL, R., HOLDEN, L. S., AND HUGHES, J. F. 1999. Art-based rendering of fur, grass, and trees. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 433–438.

LEISTER, W. 1994. Computer generated copper plates. *Computer Graphics Forum 13*, 1 (February), 69–77.

NORTHRUP, J. D., AND MARKOSIAN, L. 2000. Artistic silhouettes: a hybrid approach. In *NPAR '00: Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, ACM, New York, NY, USA, 31–37.

OHTAKE, Y., BELYAEV, A., AND SEIDEL, H.-P. 2004. Ridge-valley lines on meshes via implicit surface fitting. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, ACM, New York, NY, USA, 609–612.

POTTER, K., GOOCH, A., GOOCH, B., WILLEMSEN, P., KNISS, J., RIESENFELD, R., AND SHIRLEY, P. 2009. Resolution independent npr-style 3d line textures. *Computer Graphics Forum 28*, 1, 52–62.

RASKAR, R., AND COHEN, M. 1999. Image precision silhouette edges. In *Proceedings of the 1999 symposium on Interactive 3D graphics*, 135–140.

SAITO, T., AND TAKAHASHI, T. 1990. Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph. 24*, 4, 197–206.

SMITH, S. M., AND BRADY, J. M. 1997. Susan—a new approach to low level image processing. *International Journal of Computer Vision 23*, 1 (May), 45–78.

STEPHENS, A., BOULOS, S., BIGLER, J., WALD, I., AND PARKER, S. G. 2006. An application of scalable massive model interaction using shared memory systems. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*.

STROTHOTTE, T., AND SCHLECHTWEG, S. 2002. *Non-Photorealistic Computer Graphics. Modelling, Animation, and Rendering*. Morgan Kaufmann.

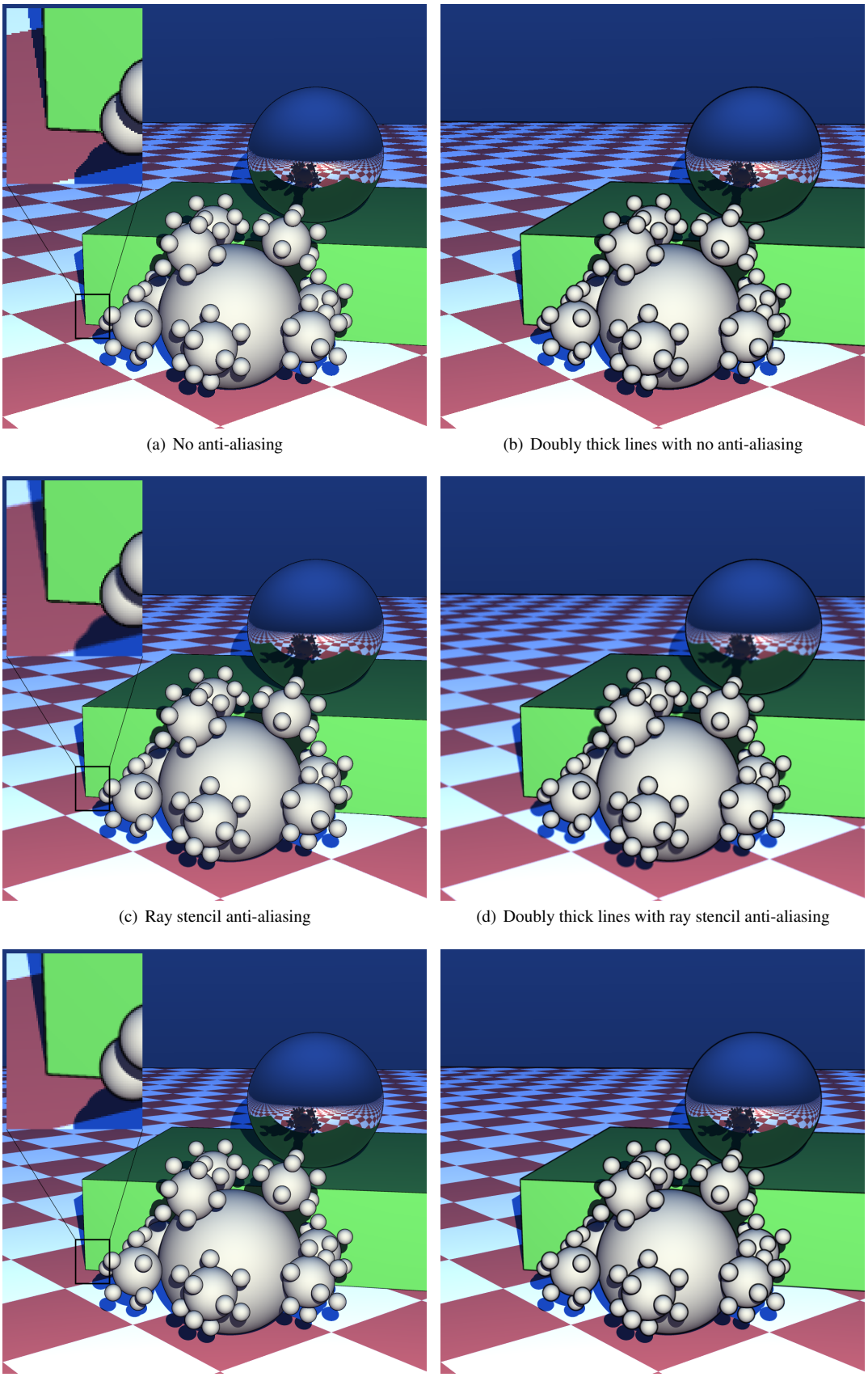
SULSKY, D., ZHOU, S.-J., AND SCHREYER, H. L. 1995. Application of a particle-in-cell method to solid mechanics. *Computer Physics Communications 87*, 1–2 (May), 236–252.

TARINI, M., CIGNONI, P., AND MONTANI, C. 2006. Ambient occlusion and edge cueing for enhancing real time molecular visualization. *IEEE Transactions on Visualization and Computer Graphics 12*, 5, 1237–1244.

WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. G. 2006. Ray tracing animated scenes using coherent grid traversal. *ACM Trans. Graph. 25*, 3, 485–493.

WALD, I., IZE, T., AND PARKER, S. G. 2008. Fast, parallel, and asynchronous construction of bvhs for ray tracing animated scenes. *Computers and Graphics 32*, 1 (February), 3–13.

WHITTED, T. 1980. An improved illumination model for shaded display. *Commun. ACM 23*, 6, 343–349.



(a) No anti-aliasing

(b) Doubly thick lines with no anti-aliasing

(c) Ray stencil anti-aliasing

(d) Doubly thick lines with ray stencil anti-aliasing

(e) Multisampling (nine samples per pixel)

(f) Doubly thick lines with multisampling (nine samples per pixel)

Figure 10: *Demonstrations of different effects within our line renderer. The right column is the same as the left column, except the lines are set to be drawn twice as thick. The second row shows how we can anti-alias the scene using the rays in the ray stencil as additional scene samples. For comparison, the third row shows scene anti-aliasing using a traditional multisampling strategy. Though the multisampled images are of higher quality than the stencil anti-aliased images, they take significantly longer to render.*