Production Ray Tracing of Feature Lines

Shinji Ogaki
Solid Angle

Iliyan Georgiev
Solid Angle

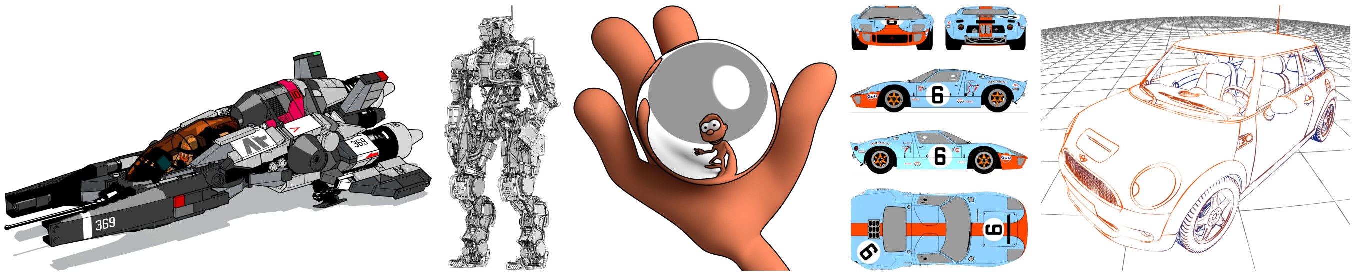

Figure 1: Our image-space algorithm can efficiently render feature lines of complex ray-traced objects. It supports arbitrary camera projections and surface shaders, specular reflection and refraction, and allows for varying the line style across surfaces.

ABSTRACT

Automated feature line drawing of virtual 3D objects helps artists depict shapes and allows for creating stylistic rendering effects. High-fidelity drawing of lines that are very thin or have varying thickness and color, or lines of recursively reflected and refracted objects, is a challenging task. In this paper we describe an image-based feature detection and line drawing method that integrates naturally into a ray tracing renderer and runs as a post-process, after the pixel sampling stage. Our method supports arbitrary camera projections and surface shaders, and its performance does not dependent on the geometric complexity of the scene but on the pixel sampling rate. By leveraging various attributes stored in every pixel sample, which are typically available in production renderers, e.g. for arbitrary output variables (AOVs), feature lines of reflected and refracted objects can be obtained with relative ease. The color and width of the lines can be driven by the surface shaders, which allows for achieving a wide variety of artistic styles.

CCS CONCEPTS

• **Computing methodologies** → **Rendering**; **Ray tracing**;

KEYWORDS

ray tracing, non-photorealistic rendering, contours, feature lines

ACM Reference Format:

Shinji Ogaki and Iliyan Georgiev. 2018. Production Ray Tracing of Feature Lines. In *SIGGRAPH Asia 2018 Technical Briefs (SA '18 Technical Briefs), December 4–7, 2018, Tokyo, Japan.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3283254.3283273

1 INTRODUCTION

Two general approaches exist for drawing feature lines of rendered 3D objects: image-space and object-space based. Image-space methods are compatible with both rasterization and ray tracing and work as a post-process on the pixel samples. Their main advantage is that their performance is independent of the geometry tessellation rate. This approach also allows for drawing contour lines of reflected and refracted objects. One of its drawbacks is that varying the style along the lines is difficult, e.g. for emulating thick-to-thin ink strokes, which requires a distance parameterization along lines.

On the other hand, object-space line drawing is well-suited for controlling stroke styles like scumbling and for rendering motion lines. However, such methods do not easily support arbitrary camera projections and have difficulties with reflection and refraction. While they can handle thin or small objects well, their performance decreases in proportion to the number of polygons being drawn.

The contour rendering method that we present in this paper operates in image space, which is well suited for modern ray-tracing production renderers that must often handle enormous amounts of geometry and support arbitrary shaders and camera projections. Our method can draw outlines of reflected and refracted objects, which is important for film production as well as scientific illustrations. While some commercial renderers have some of these capabilities, it is not clear in existing literature how to vary the line width and color or how to handle intersections of lines with different styles. Our method allows surface shaders to control the stroke style, and renders line intersections accurately via an image-space ray tracing approach. Figure 1 showcases the capabilities of our implementation in the Arnold production ray tracer.

Another limitation of image-space contour drawing is the large amount of samples required to render thin lines without excessive aliasing and/or noise. We address this issue by imposing a minimum image-space line width that preserves the contour appearance.

2 RELATED WORK

The line drawing method of Choudhury and Parker [2009] integrates well into a ray tracing framework, but its use is limited to constant line width and color, and can produce artifacts where several contours meet, as we discuss below. Their feature detection metric also involves evaluating a number of additional “probe” samples for every pixel sample, which adds redundant computation and can become prohibitively expensive with the high pixel sampling rates used in production rendering. Our approach effectively simplifies their metric to a boolean decision, adds support for surface-varying line styles, and avoids the expensive additional probe ray shooting.

Kim et al. [2008] proposed a stroke direction propagation algorithm that uses multi-perspective projections to render reflections and refractions in a line-art style. While this is an interesting approach, it only considers single reflection and double refraction.

Bauer [2017] addressed temporal flickering, which is a common issue in contour drawing. His algorithm accumulates a high-resolution sub-pixel contour image as samples are taken according to the specifics of the rendering system it is tailored to. The feature detection and the progressive refinement nature of the method results in sub-optimal contour fidelity. Moreover, it is not clear how to vary the stroke style or handle contour intersections consistently with this approach. Our method considers all sampling data at once to maximize the quality of the contours and their intersections.

Grabli et al. [2004] introduced a flexible framework that allows for controlling the line style via custom shaders. They adopt the edge extraction algorithm of Hertzmann and Zorin [2000] which operates in object space, making it difficult to support reflection and refraction. Our image-space method allows for varying the line style and also supports ray-traced reflection and refraction.

3 OUR METHOD

Our goal is to augment a ray-traced image with lines along certain features: object silhouettes, intersections between objects, and creases (i.e. regions of high surface curvature). Additionally, we allow surface shaders to define custom features, as described below.

We target integration into a production renderer, which produces a high-quality image by taking a large number of ray samples per pixel. We want to perform the contour drawing in image space as a post-process effect so as to avoid invasive modifications to the renderer. To this end, we take advantage of the dense image sampling and perform feature detection based solely on comparing geometric and shading attributes readily stored in the payloads of pixel samples. Every sample that lies on a feature line has its color changed to the color of the line. All samples are then fed to the pixel filter for final averaging, as usual. We next describe these steps in detail.

3.1 Feature and line attributes

A feature region in the image is one where the visible scene geometry changes rapidly. To detect such changes, we look for variations in the *feature attributes* of adjacent pixel samples. Four such attributes are readily supplied by the renderer: object ID, shader ID, texture coordinate, and surface normal. We also allow shaders to output an additional “feature color” at every surface point. This attribute is quite versatile: for example, a shader can assign a different constant color to every polygon or black/white colors corresponding to the object’s shadowed/lit regions. Our algorithm will then detect the edges in this output and draw corresponding lines, e.g. along the polygon edges (see the ground plane in Fig. 1, right) or the shadow boundaries.

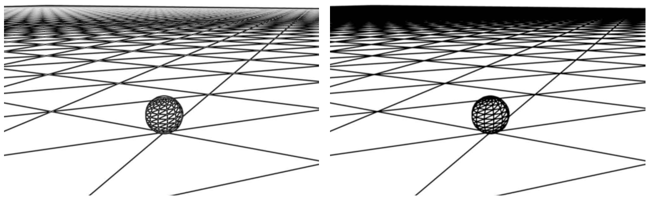
Figure 2: The method of Choudhury and Parker [2009] does not consider line intersections and produces conspicuous aliasing (left). Our method renders these correctly (right).

At every surface point, shaders can also optionally output *line attributes*: color and thickness. These override the user-specified global line attributes and allow for creating style variations along the feature lines, e.g. driven by textures or the illumination (see the car in Fig. 1, right).

3.2 Feature detection and line shading

Image-space methods draw lines by altering the colors of pixel samples that lie near features edges. Traditionally this is done by blending between the input ray-traced color and the line color, based on the feature strength and/or the sample’s image-space proximity to the feature edge [Bauer 2017; Choudhury and Parker 2009]. While good for anti-aliasing, this approach has difficulties handling varying stroke styles and line intersections, and it is also prone to producing artifacts. For example, the edge strength metric of Choudhury and Parker [2009] assumes that a pixel sample can lie near at most one, straight feature edge, causing unnatural brightening in regions with curved edges, or where several edges meet as we illustrate in Fig. 2, left.

Our approach to solving these issues is to treat the feature lines as opaque pieces of geometry that, even though defined and drawn in image space, are attached to the 3D scene objects and can therefore intersect and occlude each other. A pixel sample can then “see” only the topmost of potentially several overlapping lines. We test every sample for line intersection by performing a ray-tracing-like operation in image space.

Given a sample, henceforth referred to as *target sample*, we look for potential feature line intersections by comparing the target’s five feature attributes to those of every other sample in its vicinity, henceforth called *neighbor samples*. The neighborhood is a disk-shaped stencil, illustrated in Fig. 3, left, with diameter equal to the maximum possible line width, which is specified by the user. If a neighbor sample has an attribute sufficiently different from that of the target, according to a user-specified threshold, then the target sample potentially lies on a feature line. For texture coordinates we threshold the texture-space distance, for normals the angle, and the feature color difference is thresholded per color channel.

To obtain a valid line intersection, the image-space distance to the neighbor sample must be smaller than half the width of the

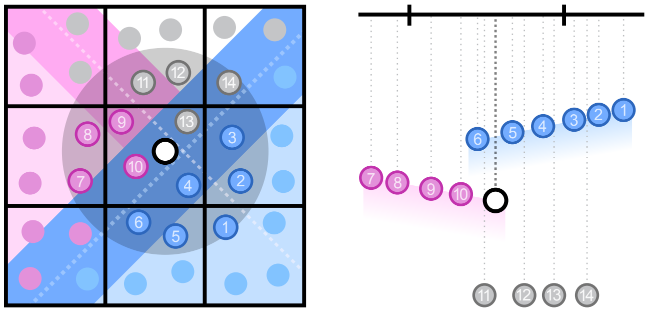

Figure 3: We test every pixel sample for feature line intersection by inspecting the samples in a circular neighborhood around it (left) in front-to-back order (right). In this example, the target sample (in white) hits the pink object but lies on the silhouette line of the blue object; the gray samples do not intersect any geometry.

tested line. The target and the neighbor samples will generally have different line widths defined by their corresponding surface shaders. One of the two samples should take priority in specifying the line attributes, and by default this is the one with the shorter ray hit distance. This ensures that we draw the correct silhouette lines of every object, as styled by its associated shader, on top of the objects behind it.

Since we only need to find the closest line intersection at the target sample, we iterate over the target-neighbor pairs in order of decreasing priority, i.e. by the depth of the priority sample in every pair, as illustrated in Fig. 3, right. This ensures that we test the lines in front-to-back order. If two or more pairs have the same priority, they are checked in order of increasing target-neighbor distance in image space. As soon as a valid intersection is found, we terminate the search and set the color of the target sample to the line color of the pair's priority sample. For artistic flexibility we also allow the priority to be manually overridden, as we discuss next.

3.3 Line priority

When two objects with different line styles intersect, the above described algorithm will draw half of each line style on the two sides of the intersection edge. This can result in a dual-color line, which is not always desired. To avoid this, and for better artistic control, we allow the user to set explicit priorities to objects. These take precedence over the default depth-based criterion for determining sample priority. When set, only the line of the higher-priority object will be drawn, with full width. An example is shown in Fig. 4, left, where by default double lines are drawn along object intersection edges. Setting object priorities results in a more pleasant look in Fig. 4, middle.

3.4 Reflection and refraction

Extending our method to draw feature lines in specular reflections and refractions is relatively straightforward. To that end, we store the tree of reflection and refraction events associated with every pixel sample, and record feature and line attributes for every tree node. The leaves of the tree are the non-reflective/refractive ray hits where the ray recursion terminates. (For samples that hit such surfaces directly from the camera, the ray tree is made of just a single leaf node.)

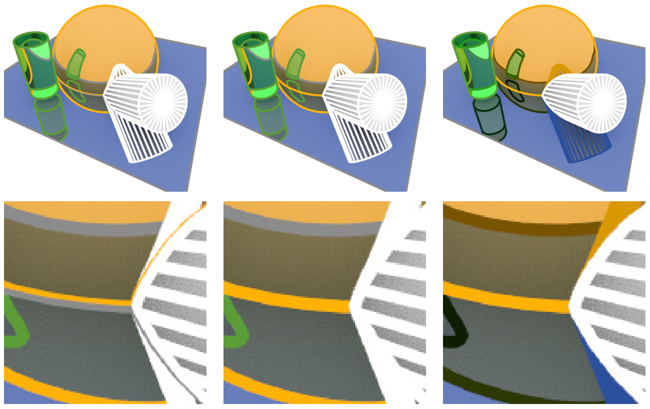

Figure 4: Line priority is by default determined by the distance along the ray, where the shorter wins (left). Per-object user-set priorities are also supported (middle); here the cylinders have the highest priority and the ground plane has the lowest. Additionally tinting the line colors of reflected and refracted objects gives a more realistic result (right).

We first run our feature detection algorithm on the attributes at the root node of every sample, which correspond to the primary ray hits. Then, we identify the samples that have not yet intersected a line and run the same process again only for those samples, this time considering the primary reflection attributes. Samples that have not hit a reflective surface (i.e. have null reflection attributes) are not considered. We then repeat the process by considering the refraction attributes, and so on. Thus, the attribute trees of all samples are traversed in sync, in breath-first order, such that the lines of reflected objects cover those of refracted objects.

While feature detection is performed solely in image space, by caching the throughput of the corresponding ray paths in the tree nodes, the line colors can be additionally tinted to make them appear reflected or refracted. This gives more realistic results, as seen in Fig. 4, right.

3.5 Handling thin lines

In order to draw thin contour lines without excessive noise, we impose a minimum line width in image space, relative to the pixel size. This technique is typically used to prevent aliasing of thin hair or grass curves by expanding their width so that they cover at least a specified distance across a pixel, and compensating for this expansion by making the curves proportionately more transparent [Cook et al. 2007; Georgiev et al. 2018]. The same approach is directly applicable to our contour line drawing. Excessive widening of the lines changes the look, so instead of exposing the minimum width as a user parameter, we fix it to half the pixel size (see Fig. 5).

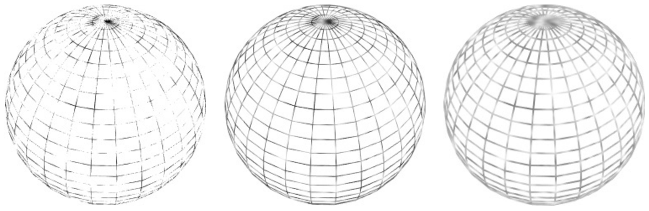
Figure 5: Lines with width driven by a noise function, at 9 samples/pixel. The minimum line width (MLW) is, from left to right: 0, 0.5, and 1 pixel. We fail to capture thin lines with MLW=0, and setting MLW=1 changes the look too much.

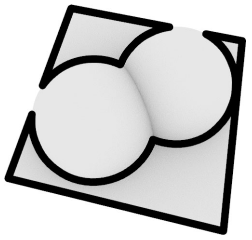
Figure 6: A difficult case for our method, which fails to draw the silhouette of the rectangle in regions where it is covered by the no-silhouette sphere.

4 RESULTS, LIMITATIONS, FUTURE WORK

We have implemented our feature line drawing method in the Arnold production ray tracer [Georgiev et al. 2018]. Being an image-space filter, our algorithm can handle complex geometry and arbitrary camera projections. We are also free to use any surface shader, and purpose-built shaders can provide additional feature and line-style attributes. Figure 1 showcases our implementation, where in the image on the right the line style is driven by the illumination.

Besides the general limitations of image-space contour drawing, our method can suffer from a few other issues. First, lines can be missed when a no-line object is placed in front of a thick-line object and their silhouettes overlap, as seen in Fig. 6. The issue could be addressed by computing the intersections of multiple objects along every ray instead of just the closest one [Wald et al. 2018]. Storing several sets of feature attributes per sample, one set for every intersected object, would enable edge detection for non-directly visible objects.

Another issue is the performance drop with large numbers of neighbor samples in the stencil due to thick lines and excessively high pixel-sampling rates. Fortunately, this is not a critical issue in practice, since lines are typically at most a few pixels wide. High sampling rates also increase the memory consumption of the per-sample attribute caching. To ameliorate this and keep memory usage low, we render and post-process the image in small tiles.

Our method is incompatible with blurry effects like glossy reflection/refraction, depth of field, and motion blur. In the presence of such effects, neighbor samples can have very different attributes and our algorithm ends up finding features everywhere. In fact, meaningfully defining contours for such effects remains an open problem. We have seen some demand for blurring edges when rendering with depth of field and would like to investigate how to do this. One option for motion blur could be to use deterministic time sampling. This would result in a stroboscopic effect that might be preferred, on top of which motion lines can be additionally drawn.

Drawing dashed lines or lines with distance-varying style requires the ability to identify the lines' endpoints and to measure distances along them. This is something we would like to look into.

Lastly, for stereoscopic viewing we render the left- and right-eye images completely independently. While we have not yet experienced serious issues with this simple approach, it is prone to causing viewing discomfort, which can be mitigated by drawing the feature lines in a stereo-consistent way [Bukenberger et al. 2018].

ACKNOWLEDGMENTS

The authors thank Lee Griggs for rendering the images in Fig. 1: The space ship model is from mecabricks.com, the robot was modeled by Stas Gorshenin, and the Pepe model was created by Daniel Martinez Lara.

REFERENCES

Andreas Bauer. 2017. A New Contour Method for Highly Detailed Geometry. In *ACM SIGGRAPH 2017 Talks (SIGGRAPH '17)*. ACM, New York, NY, USA, Article 71, 2 pages. https://doi.org/10.1145/3084363.3085052

Dennis R. Bukenberger, Katharina Schwarz, and Hendrik P. A. Lensch. 2018. Stereo-Consistent Contours in Object Space. *Comput. Graph. Forum* 37, 1 (2018), 301–312. https://doi.org/10.1111/cgf.13291

A. N. M. Imroz Choudhury and Steven G. Parker. 2009. Ray Tracing NPR-style Feature Lines. In *Proceedings of the 7th International Symposium on Non-Photorealistic Animation and Rendering (NPAR '09)*. ACM, New York, NY, USA, 5–14. https://doi.org/10.1145/1572614.1572616

Robert L. Cook, John Halstead, Maxwell Planck, and David Ryu. 2007. Stochastic Simplification of Aggregate Detail. *ACM Transactions on Graphics* 26, 3 (2007).

Iliyan Georgiev, Thiago Ize, Mike Farnsworth, Ramón Montoya-Vozmediano, Alan King, Brecht Van Lommel, Angel Jimenez, Oscar Anson, Shinji Ogaki, Eric Johnston, Adrien Herubel, Declan Russell, Frédéric Servant, and Marcos Fajardo. 2018. Arnold: A Brute-Force Production Path Tracer. *ACM Trans. Graph.* 37, 3, Article 32 (Aug. 2018), 12 pages. https://doi.org/10.1145/3182160

Stéphane Grabli, Emmanuel Turquin, Frédo Durand, and François X. Sillion. 2004. Programmable Style for NPR Line Drawing. In *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques (EGSR'04)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 33–44. https://doi.org/10.2312/EGWR/EGSR04/033-044

Aaron Hertzmann and Denis Zorin. 2000. Illustrating Smooth Surfaces. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 517–526. https://doi.org/10.1145/344779.345074

Yongjin Kim, Jingyi Yu, Xuan Yu, and Seungyong Lee. 2008. Line-art Illustration of Dynamic and Specular Surfaces. In *ACM SIGGRAPH Asia 2008 Papers (SIGGRAPH Asia '08)*. ACM, New York, NY, USA, Article 156, 10 pages. https://doi.org/10.1145/1457515.1409109

Ingo Wald, Jefferson Amstutz, and Carsten Benthin. 2018. Robust Iterative Find-Next-Hit Ray Traversal. In *Eurographics Symposium on Parallel Graphics and Visualization*, Hank Childs and Fernando Cucchietti (Eds.). The Eurographics Association. https://doi.org/10.2312/pgv.20181092