Real-time Cast Shadow Contours

Péter Barabás*
Supervised by: László Szécsi†

Computer Graphics Group
Department of Control Engineering and Information Technology
Budapest University of Technology and Economics
Budapest / Hungary

Abstract

This paper presents a real-time algorithm for drawing shadow contours in non-photorealistic rendering. We use the straightforward idea of intersecting shadow volumes with shadow receiver surfaces, proposing a practical scheme for accelerating the process on the GPU. Real-time operation is achieved by building a 2D bounding volume hierarchy (BVH) that relies on implicit spatial coherence in triangle mesh models.

Keywords: NPR, outline rendering, shadow volumes

1 Introduction

Photo-realism has been in the focus of rendering systems for decades. Photo-realistic rendering aims at creating images that are indistinguishable from real-world photographs, which is made possible by the precise simulation of physics laws during the rendering process. How accurately physics is applied in the rendering algorithm determines the level of realism of the result.

Computer graphics also tries to mimic artistic expression and illustration styles [6, 16, 18]. Such methods are usually vaguely classified as *non photo-realistic rendering* (NPR). While the fundamentals of photo-realistic rendering are in optics that are well understood, NPR systems simulate artistic behavior that is not mathematically founded and often seems to be unpredictable. Therefore, the first step of NPR is to model the artist by establishing a mathematical model describing his style, and then solve this model with the computer. The result will be acceptable if our model is close to the not formally specified artistic behavior. During the history of NPR, many individual styles were addressed. Many of those styles employ pen lines, pencil lines, or brush strokes to build an image. These elements are often used to draw outlines.

Outline visualization is extensively used in a wide range of applications, from CAD systems to stylized rendering. It can clarify the shape of a complex 3D object or may highlight essential features. The human visual system processes seen images by identifying shapes separated by discontinuities. Outline rendering provides strong cues for shape separation, substituting for subtle and expensively rendered real-world cues like scattered lighting and shadows, and providing a stronger visual language in stylistic rendering. Cartoon shading, in particular, relies on edge visualization to convey shape information, in lieu of realistic shading.

This paper proposes a stylized rendering method where outlines are drawn to emphasize the contours of shadows, and describes a GPU-based real-time implementation. The organization of the paper is as follows. In Section 2 we summarize the related previous work on NPR, and outline rendering in particular. We explain why cast shadow contours received little attention, and evaluate the fitness of existing methods for this purpose. Section 3 introduces our approach. A detailed description of the final algorithm, and the discussion of results and future work conclude the paper.

2 Previous work

There are two well known approaches to outline rendering. The first one works in image space with the use of color, normal, and depth maps [15]. Edge pixels—those that lie near discontinuities in these maps—can be found using edge detection filters. What level of image-space discontinuity warrants outline edges must be adjusted by fine-tuning filter parameters and applying mask textures [17]. Object-space consistency of outlines during animations is also subject to those parameters. Cast shadow contours can easily be drawn if edges are detected on an untextured but shadowed rendering of the scene. The main problem with this approach is the excessive texture access bandwidth and the absence of real scalability in line features.

The other approach works in world space and generates new triangle strip geometry to visualize the outlines. In the following, we discuss methods in this category in greater detail.

There are two basic classes of outlines that are always drawn in line art, both indicating some kind of perceived

*medve9213@gmail.com
†szecsi.laszlo@gmail.com

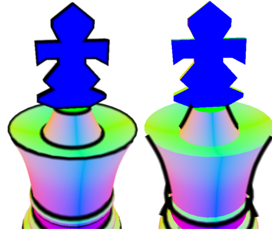


Figure 1: Crease (left) and silhouette (right) outlines.

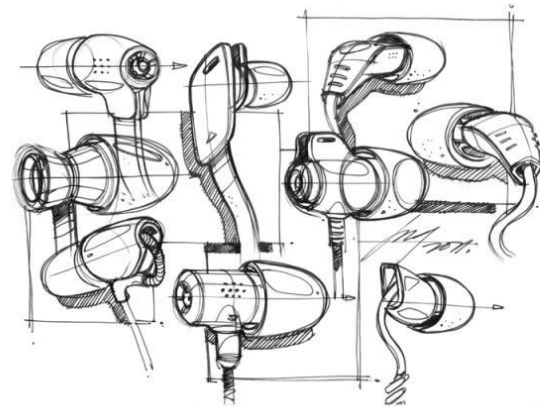
discontinuity (see Figure 1). *Silhouettes* appear at discontinuities in image space, where a continuous object surface appears to end. For manifold surface models this can happen only where the surface folds behind itself, meaning that outlines are located on the border of the visible (camera-facing) and not visible (back-facing) part of the object surface. The other class of displayed outlines—called *creases* or feature lines—indicate discontinuities in the surface normals, and they are defined by the topology of the mesh itself, independent of the view direction or the camera settings. In this paper, we take some ideas from conventional silhouette identification approaches, and discuss which are useful in finding shadow contours. Our presented results also include conventional outline rendering in addition to the newly proposed shadow contour outlines.

In addition to the silhouettes and creases discussed above, outline drawings may feature further lines. Suggestive contours [4] and apparent ridges [10] define outlines based on surface curvature characteristics. While these can provide superior visual cues, especially in absence of additional shading, they are less fit if we aim at minimal-overhead real-time rendering [3].

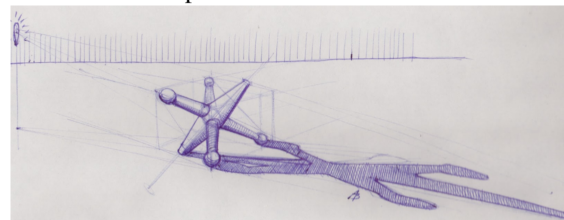
Cast shadow contours are yet another class of outlines that can appear in drawings. They have received less attention in research, both for artistic and technical reasons.

On the artistic side, cast shadow contours are relatively rarely drawn in technical or artistic images. Quite often, under natural illumination, shadows are supposed to have soft edges, and it is undesirable to draw attention to discontinuities in brightness due to cast shadows. When shadows need to appear hard, they are often rendered in solid black, making outlines not very prominent, even if drawn. However, where shadows need to be emphasized, especially in architectural or artistic sketches (Figure 2), shadow outlines are often drawn. Even in paintings, some strokes aligned on cast shadow contours are present (Figure 3).

On the technical side, cast shadow contour generation is theoretically straightforward, and less prone to artifacts than silhouette outlines. Eisemann et al. [5] described the process of intersecting shadow volumes with the meshed shadow receiver surface. It requires the identification of the shadow caster silhouette as seen from the light source, and projecting it onto shadow receiver surfaces. We discuss these two phases in the following subsections.

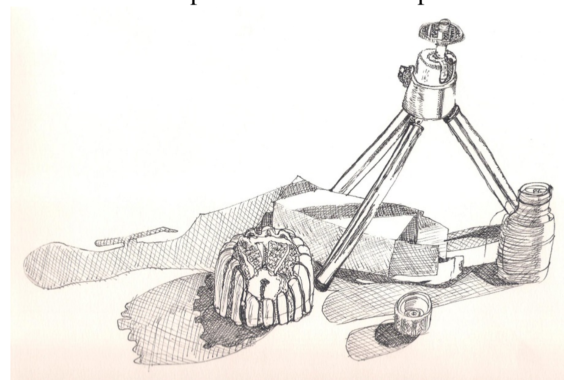


source: https://u.osu.edu/idvisualization/



source: http://tightline-sketchblog.blogspot.hu/

source: http://www.anfitrion.co/p/2682/



source: https://alison512480.wordpress.com

Figure 2: Architectural or artistic sketches with cast shadow contours.

Figure 3: The Night Café in Arles by Vincent van Gogh, watercolor.

2.1 Shadow volume generation

Extraction of the shadow caster silhouette is a well-known operation in *shadow volume* computation used for *stencil shadows*, which can be implemented in a GPU shader pass [2].

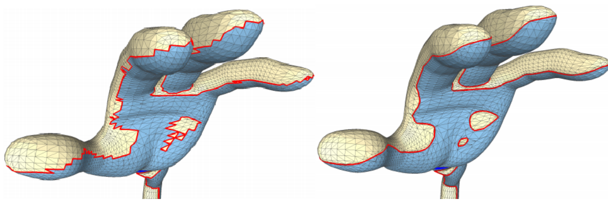


Figure 4: Markosian [14] (left), and Herzmann–Zorin silhouettes [9] (right) seen from a viewpont different from the camera position used for silhouette extraction. Figure taken from Benard et al. [1].

There are two basic ways to define silhouettes on triangle meshes. The approach by Markosian et al. [14] operates on the discrete triangle mesh geometry itself, selecting those edges as silhouette edges which separate front-facing and back-facing triangles. When rendered in solid color, the silhouette looks smooth, but—as shown in Figure 4—it is often ragged on the object surface. In the image plane, the edge loop can even turn back along the object silhouette multiple times, which becomes apparent if strokes are rendered semi-transparently. The definition by Hertzmann and Zorin [9] avoids this problem, as it considers the smooth surface instead of the triangulated one, reconstructing silhouettes from the vertex normals. For a given vertex $\mathbf{a}$ with normal $\mathbf{n_a}$ and vector $\mathbf{c_a}$ to the camera, we define the scalar field $f(\mathbf{a}) = \mathbf{n_a} \cdot \mathbf{c_a}$, extending f to triangle interiors by linear interpolation. Silhouettes are taken to be the zeroset of f, yielding clean, closed polylines whose segments traverse faces in the mesh (rather than following edges, as in the Markosian method).

For stencil shadows, the Markosian-style silhouettes are extruded, as they provide artifact-free self-shadowing, and overly complex or back-tracking shadow volume boundaries do not influence the quality of the projected shadows. However, for the purposes of cast shadow contour rendering, these problems are just as relevant as for straightforward silhouette rendering. Therefore, Hertzmann-and-Zorin-style silhouettes should be preferred.

When rendering silhouettes, hidden outlines should be removed. This is expensive to solve geometrically, thus screen-space methods are preferred. Depth testing is quite unreliable, and ID buffers were more successfully used [13]. For cast shadow outlines, there is no established practice. Geometric processing of shadow volumes would not be real-time, and adapting the ID buffer method is also not straightforward. Thus, we propose to solve the problem of removing cast shadow outlines due to hidden-from-light silhouettes in screen space (in Section 3.2).

2.2 Intersection

Eisemann et al. [5] described the process of intersecting shadow volumes with the meshed shadow receiver surface. Their purpose for extracting cast shadow contours was to transform 3D objects into 2D clip art. Therefore, real-time performance was not targeted and no acceleration scheme for the intersection was proposed.

Performing intersection in real time, however, is challenging in practice, as it is a crossbar on shadow volume and surface mesh faces, resulting in a naive algorithm of $\mathcal{O}(n^2)$ time complexity. Intersection tests can be accelerated using spatial subdivision schemes, but in dynamic scenes the cost of constructing those may be prohibitive.

In this paper, we show that we get a reasonably tight bounding volume hierarchy over the shadow volume faces, if we apply the *object median* subdivision scheme on the primitive stream generated by a contour-extruding geometry shader, without any additional ordering or cost heuristics. We compare intersection performance with that of a proper top-down object median subdivision scheme with object sorting to show that there is no significant performance penalty incurred.

3 New method

Generating cast shadow contours is a simple and straightforward problem in theory. By using the geometry of shadow volumes, we can easily find the intersections between shadow volume faces and shadow receiver faces. This would mean, however, that we would need to check every shadow volume face against every shadow receiver face. Even a moderately complex scene would impose a prohibitively large computation time, if we were to utilize this naive approach.

3.1 Intersection acceleration

We assume a perspective pinhole camera for a point light source, or an orthographic camera for a directional light source. Shadow caster silhouette edges—which are also faces of the shadow volume, when extruded—appear as 2D line segments when projected on the camera's image plane. Our method relies on building a 2D *bounding volume hierarchy* (BVH) over these line segments. Traversing the BVH allows us to reduce the number of shadow volume faces we need to check for each receiver face. The contour edges constitute the leaves of the BVH tree, and nodes are *axis-aligned bounding boxes*, or AABBs—rectangles in this 2D case—, enclosing all child leaves. During traversal, we project each shadow receiver face to the camera plane, and check its AABB against the cells of the BVH recursively. If there is no intersection with a cell, its children are not traversed, filtering out most of the shadow volume faces that our shadow receiver face does not intersect.

A BVH is useful if it eliminates costly intersection computations. Thus, BVH cells should be as small as possible. This is often achieved by separating primitives into two locally coherent clusters, and repeating the process recursively to obtain a hierarchy. The clustering can be done by sorting primitives according to one (or more) of their spatial position coordinates, and splitting the sorted list into two parts. Splitting may be based on cost heuristics or simple strategies like the spatial median (similarly sized cells) or the object median (same number of elements in both cells) [8, 19].

Building a BVH that is efficient, however, suffers from the same performance problems that we aim to solve. Various methods for interactively building BVHs with GPU support have been proposed, one of the most relevant being *linear bounding volume hierarchies* (LBVH) [12, 11]. However, even this method requires sorting primitives at least once, with a severe performance impact for our application. In this paper, we investigate the effect of relying on the inherent spatial coherence in typical triangle mesh models, completely forgoing the sorting step. This means that we use the cast shadow contour's edge primitives in the order they are written by the GPU after the silhouette detector shader. We use the object median splitting scheme, to avoid computation of cost heurisics, and obtain a balanced BVH tree, which is both easy to store and efficient to traverse on the GPU. This allows us to create the BVH in a bottom-up fashion, which can be solved efficiently using parallel *reduction* [7].

As triangle mesh geometries are typically modelled with some inherent spatial coherence or even optimized into triangle strips, our intuition was that even if the silhouette detection and the parallel stream processing introduce some randomness, the output contour segments would still exhibit sufficient coherence on a local scale. This would mean that the BVH built using this ordering is only sub-par on a few of the highest levels, compared to one built with proper sorting, introducing a fairly constant, but relatively small overhead.

3.2 Hidden caster silhouette removal

Shadow caster silhouettes hidden from the light source appear on the shadow receiver surface as shadow contours that fall inside already shadowed areas. These inner shadow contours need to be filtered out, otherwise multiple objects casting overlapping shadows or concave shadow casters would cause erroneous contours to appear on receivers. To solve this, we utilized the information already available to us via shadow volume generation—the stencil buffer. In the stencil buffer, each texel has a value corresponding to the amount of shadow volumes it is contained in. This means we can use that information to check if a contour edge is inside a single shadow volume or not. Some ambiguity would be present, as all contours are exactly on the boundary of the shadow volume. In order to avoid the flickering caused by these inaccuracies, we offset all contours towards the inside of the shadowed surface, using the receiver surface normals and the shadow volume face normals.

4 Implementation

The implementation of such an algorithm is inherently multi-pass. The following steps provide an overview of what an implementation entails:

- shadow volume generation,
- rendering shadow volume faces and computing axis aligned bounding rectangles in the light source camera's screen space,
- building the bounding volume hierarchy,
- checking for shadow volume–shadow receiver intersections using the bounding volume hierarchy.

Each step is implemented as a separate shader pass. Since we want to implement the bounding volume hierarchy builder using parallel reduction, using the GPU and a very short compute shader is a natural choice. Normally, only the result in the stencil buffer is used when implementing shadow volume based shadows, the geometry is thrown away. For our purposes, we need to access the geometry of the shadow volume in later passes. For this we use the stream output functionality of GPUs, emitting the world positions of the vertices composing the extruded faces of the volume.

Once we have the shadow volume faces, we feed the contents of the stream output buffer back into the pipeline for a pass rendered from the light source. The light source requires its view and projection matrices to be set-up—similar to the shadow mapping technique—since we are rendering from the viewpoint of the light. The shader run

in this pass is very straightforward, we just transform the primitives to the space of the light source, project them, and create the AABBs we need in the next step. The end results are streamed out as well, which gives us the actual order of the leaves later in the bounding volume hierarchy.

4.1 Building the BVH

We set up the buffer that will contain the bounding volume hierarchy using a representation that is similar to the array representation of the heap data structure. This helps us to easily calculate indexes of child nodes. The bounding volume hierarchy is created using a compute shader, which processes each level above the leaf level. The buffer containing the hierarchy is filled up from back to front, with leaves being the elements at the end, and their parents occupying the previous elements.

The only guarantee in stream output primitive order is that subsequent draw calls will take up subsequent regions in the stream output buffer. Since the primitive order defines the spatial coherence of the bounding volume hierarchy, we will need to measure performance against a bounding volume hierarchy constructed with proper sorting.

4.2 Tree traversal and edge generation

The final pass is where we find the actual shadow contours. A geometry shader runs on the shadow receiving meshes. As visible cast shadow outlines appear on faces that appear as front faces as seen both from the camera and from the light source, back faces in either aspect are culled. Then, we generate the light-screen-space axis aligned bounding rectangle of the primitive, and traverse the bounding volume hierarchy while checking against the axis aligned bounding rectangles of its nodes. For each traversed leaf we precisely calculate if there actually is any intersection with the primitive. If there is, we have to determine whether the contour is completely inside the primitive, or just partially, and calculate the actual intersection points accordingly.

Traversing a tree-like data structure is usually achieved by recursion, which is forbidden in shader code. Therefore, we implemented recursion using a small local stack. At each intersected node of the tree we push one child node on our self-managed stack and evaluate the other. After a traversal branch terminates because of a non-overlapping AABB or because of reaching a leaf, we pop a node from the top of the stack. Traversal is complete when no nodes remain on the stack. We also skip faces with normals facing away from our light source to eliminate intersections on sides opposite from the light, and to save performance.

There is a practical limitation with regards to using the geometry shader. The buffer we output vertices into is limited in size, so depending on the amount of data we want to stream out—position, in our case—we can only output a limited number of vertices. With a single position tuple of four floats, we can currently output 256 vertices, which means 128 contour edges for each triangle. This limitation could require us to increase polygon count on shadow receiving geometry to prevent the GPU from discarding contour edges. To lower the impact of this limitation, we quantize the positions and discard contour segments which have no length after quantization. We achieve this by defining a grid in 3D world space with a small enough resolution to be unnoticeable—usually around 1/10 of the contour stroke width—and then each contour segment point is snapped onto the nearest grid point. If the length of the contour segment is zero after quantization, we do not render it.

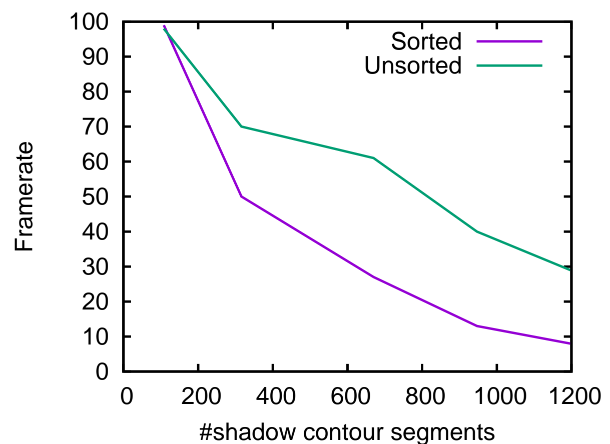


Figure 5: Overall performance of shadow outline rendering with our GPU bottom-up bounding volume hierarchy construction without sorting, and with the CPU top-down solution with sorting, on an NVidia 970 GTX and an i7 3770K. The shadow receiver had 2182 faces.

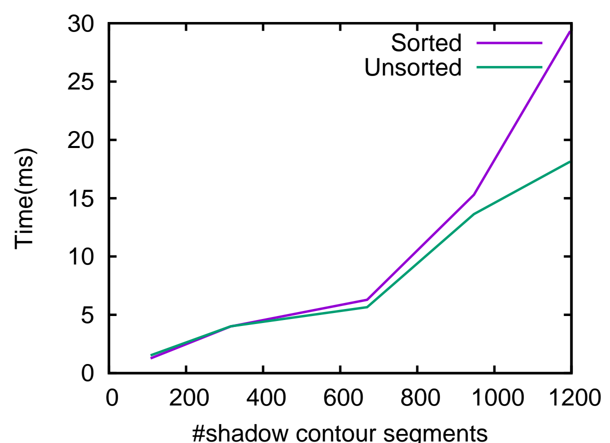


Figure 6: Tree traversal times for trees constructed with our GPU bottom-up approach without sorting, and with the CPU top-down solution with sorting. Surprisingly, the unsorted tree often outperformed the sorted one.

5 Evaluation of unsorted bounding volume hierarchy performance

For the purpose of comparison, we needed to implement a bounding volume hierarchy building algorithm that does not exploit pre-existing locality, but sorts the silhouette segments before subdividing them. Numerous such algorithms exist, with different schemes used for partitioning objects (Sulaiman [19] offers an overview). These may have significant differences in performance. We opted for a simple one that is close to our method in that it builds a balanced tree, and does not use expensive cost estimates to improve partitioning: the object median method. This is a top-down method, sorting the objects according to a coordinate axis, and splitting them so that the two partitions have the same number of elements. We implemented this method on the CPU. There is little doubt that comparison with a more sophisticated bounding volume hierarchy construction scheme (e.g. with surface area heuristics) could provide better traversal statistics—at least in theory. In practice, traversing unbalanced trees would require a larger local stack and a less direct tree representation, both of which would impact GPU performance, which is why we focused on the most practical object median method.

We measured our bottom-up GPU implementation against the top-down CPU algorithm. Not surprisingly, the overall frame rate was much more favorable with the GPU algorithm, especially as the face count of the shadow volume increased (Figure 5). This is easy to explain, as CPU sorting made the application CPU-limited, and bounding volume hierarchy construction stalled the rendering process. This could be somewhat mitigated by parallel sorting on the GPU—making the solution much more complex and difficult to implement—but tree construction times will always remain relatively high.

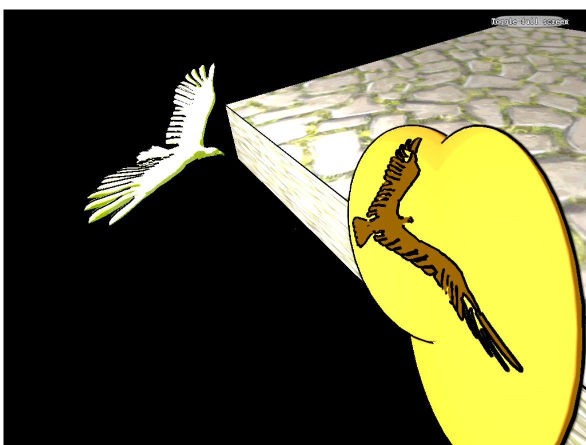


Figure 7: Eagle shadow caster test scene with cast shadow contours on double ellipsoid receiver.

More interestingly, we measured the tree traversal times for the two methods. Test scenes are shown in Figures 7, 8, 9, 10, 11. Table 1 shows scene characteristics and

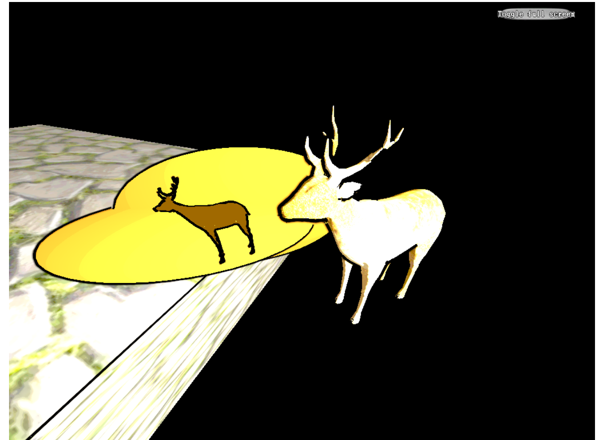


Figure 8: Deer shadow caster test scene with cast shadow contours on double ellipsoid receiver.

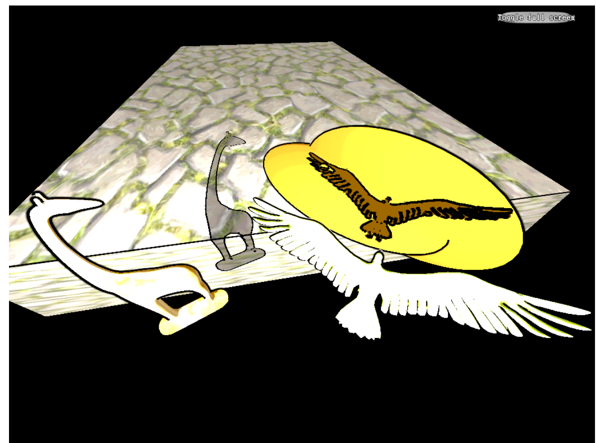


Figure 9: Eagle and giraffe shadow casters test scene with cast shadow contours on double ellipsoid receiver.

times for tree traversal with the sorted and unsorted trees. We expected the sorted tree to perform better, but of course not so much as to validate construction overhead. Surprisingly, we found that more often than not the unsorted tree performed even better than the sorted one. Thus, we conclude that not sorting the objects is perfectly sound in this application.

6 Conclusion

We have shown that during shadow contour rendering, it is unnecessary to include an expensive sorting step, when building an acceleration hierarchy over the contour edges. We have presented an algorithm for rendering cast shadow contours exploiting this fact. Comparison with the object median split scheme using sorting revealed that traversal times remain similar, while tree construction times are much lower, allowing for real-time operation for scenes of about thirty thousand triangles.

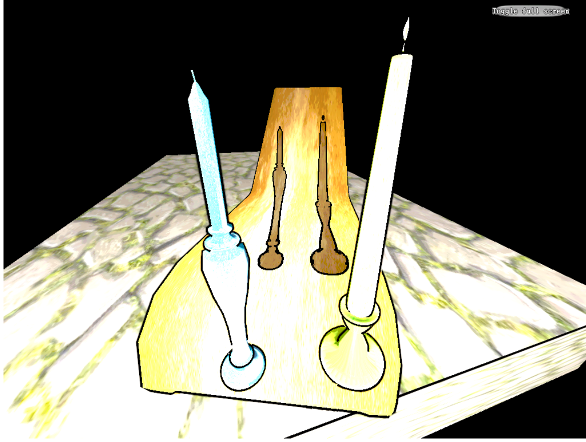


Figure 10: Candle shadow casters test scene with cast shadow contours on deck chair.

Table 1: Tree traversal times (in ms) for the test scenes. Shadow caster polygon counts and the number of shadow contour edges are given for every scene.

Test scene	polys	lines	sorted	unsorted
Eagle	21328	707	18.66	19.13
Deer	28434	797	30.25	25.23
G&E	25930	954	55.23	50.69
Candles	21594	512	28.3	19.24
Heart	20468	293	5.11	6.93

7 Future work

Stylization of the shadow contour strokes should be improved to mimic actual artistic work.

Cast shadow outlines should integrate smoothly with other kinds of strokes in artistic rendering, offering a special tool for emphasizing shadows. Therefore, we need to integrate our method with NPR techniques other than outline rendering. Filling the shadows with hatching is the most obvious task. In architectural rendering, researching ways to render outlined shadows with precise hatching strokes may be interesting.

We could further compare the performance of the unsorted bounding volume hierarchy against more sophisticated methods, like the linear bounding volume hierarchy [12, 11]. This could provide some additional insight into the locality requirements characteristics. However, if not sorting works adequately, it is hard to envision a scenario where devoting resources to build a better bounding volume hierarchy could pay off—at least in a dynamic environment where shadow contours change in every frame.

8 Acknowledgements

This work has been supported by OTKA PD-104710.

References

[1] Pierre Bénard, Aaron Hertzmann, and Michael Kass. Computing smooth surface contours with accurate topology. *ACM Transactions on Graphics (TOG)*, 33(2):19, 2014.

[2] Stefan Brabec and Hans-Peter Seidel. Shadow volumes on programmable graphics hardware. In *Computer Graphics Forum*, volume 22, pages 433–440. Wiley Online Library, 2003.

[3] D. DeCarlo, A. Finkelstein, and S. Rusinkiewicz. Interactive rendering of suggestive contours with temporal coherence. In *Proceedings of the 3rd International Symposium on Non-photorealistic Animation and Rendering*, pages 15–145. ACM, 2004.

[4] D. DeCarlo, A. Finkelstein, S. Rusinkiewicz, and A. Santella. Suggestive contours for conveying shape. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 848–855. ACM, 2003.

[5] Elmar Eisemann, Holger Winnemöller, John C Hart, and David Salesin. Stylized vector art from 3d models with region support. In *Computer Graphics Forum*, volume 27, pages 1199–1207. Wiley Online Library, 2008.

[6] Paul Haeberli. Paint by numbers: Abstract image representations. In *ACM SIGGRAPH Computer Graphics*, volume 24, pages 207–214. ACM, 1990.

[7] Mark Harris et al. Optimizing parallel reduction in CUDA. *NVIDIA Developer Technology*, 2(4), 2007.

[8] Vlastimil Havran. *Heuristic ray shooting algorithms*. PhD thesis, Citeseer, 2000.

[9] A. Hertzmann and D. Zorin. Illustrating smooth surfaces. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 517–526. ACM Press/Addison-Wesley Publishing Co., 2000.

[10] T. Judd, F. Durand, and E. Adelson. Apparent ridges for line drawing. In *ACM Transactions on Graphics (TOG)*, volume 26, pages 19–19. ACM, 2007.

[11] Tero Karras. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pages 33–37. Eurographics Association, 2012.

[12] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast bvh construction on gpus. In *Computer Graphics Forum*, volume 28, pages 375–384. Wiley Online Library, 2009.

[13] L. Markosian and J.F. Adviser-Hughes. *Art-based modeling and rendering*. Brown University, 2000.

[14] L. Markosian, M.A. Kowalski, D. Goldstein, S.J. Trychin, J.F. Hughes, and L.D. Bourdev. Real-time nonphotorealistic rendering. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 415–420. ACM Press/Addison-Wesley Publishing Co., 1997.

[15] M. Nienhaus and J. Doellner. Edge-enhancement—an algorithm for real-time non-photorealistic rendering. *Journal of WSCG*, 11(2), 2003.

[16] Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. Real-time hatching. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 581–581. ACM, 2001.

[17] J. Shin. A stylised cartoon renderer for toon shading of 3d character models. Master's thesis, University of Canterbury, UK, 2006.

[18] Thomas Strothotte and Stefan Schlechtweg. *Non-photorealistic computer graphics: modeling, rendering, and animation*. Elsevier, 2002.

[19] Hamzah Asyrani Sulaiman. *Bounding Volume Hierarchies for Collision Detection*. INTECH, 2012.

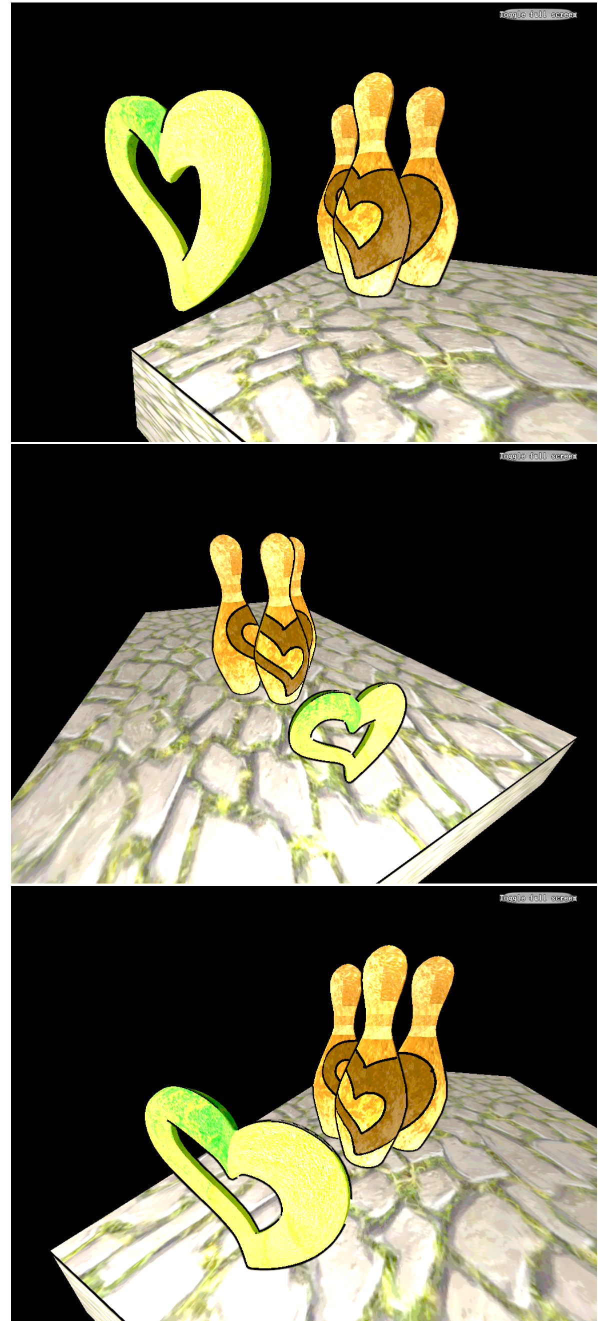


Figure 11: Heart shadow caster test scene with cast shadow contours on three bowling pin receivers.