A Simple Algorithm for Conservative and Tiled Rasterization

Tomas Akenine-Möller
Lund Institute of Technology

Timo Aila
Helsinki University of Technology
Hybrid Graphics Ltd.

February 24, 2004

Abstract

Several algorithms that use graphics hardware to accelerate processing require conservative rasterization in order to function correctly. Conservative rasterization stands for either overestimating or underestimating the size of the triangles. Overestimation is carried out by including all pixels that are at least partially overlapped by the triangle, whereas underestimation includes only the pixels that are fully inside the triangle. None or few algorithms for conservative rasterization have been described in the literature, and current hardware does not explicitly support it. Therefore, we present a simple algorithm, which requires only a small modification to the triangle setup when edge functions are used. Furthermore, the same algorithm can be used for tiled rasterization, where all pixels in a tile (e.g. 8×8 pixels) are visited before moving to the next tile.

1 Introduction

With the advent of programmable graphics hardware, lots of engineering and research work has focused on "porting" specific algorithms so that they can be run on graphics hardware. The argument for this is usually that the performance of graphics hardware grows faster than that of CPUs, and that in the long run, superior performance is achieved or can be expected. Several of these methods need to use *conservative rasterization* to report or generate correct results.

Two slightly different methods are commonly referred to as conservative rasterization. An *overestimated* footprint of a triangle includes all pixels that are at least partly overlapped by the triangle, whereas an *underestimated* footprint includes only the pixels that are completely inside by the triangle. Figure 1 shows a comparison between conservative and standard rasterization. Conservative rasterization is not applicable for the actual rendering of a scene. For example, consider two triangles that share an edge. To get the expected result when rasterizing these triangles, one usually considers a pixel to be inside a triangle if the sampling point of the pixel is inside the triangle. This avoids duplicate writes to pixels, and is critical for many techniques, e.g., shadow volume rendering [3] and transparency.

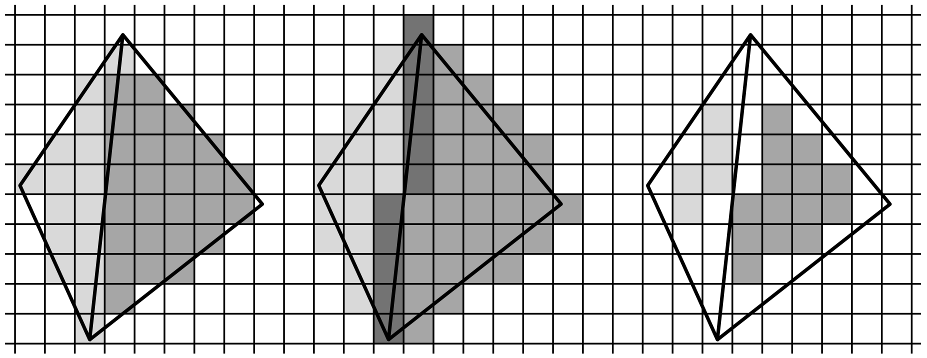

Figure 1: To the left, standard rasterization is shown for two triangles sharing an edge. There is a single sample point at the center of each pixel. In the middle, overestimating conservative rasterization is illustrated. The darkest gray indicates that both triangles have been written to those pixels. To the right, underestimating conservative rasterization is demonstrated; the cracks can be avoided by disabling the conservative rasterization for shared edges.

Still, there are several algorithms that need conservative rasterization in order to function properly. For example, Govindaraju et al. [5] use occlusion queries available in off-the-shelf graphics hardware to compute a potential colliding set of objects, followed by triangle-triangle intersection testing on the CPU. Exact results can only be obtained by using conservative rasterization, whereas a larger resolution can only make the problem less apparent. Koltun et al. [8] use graphics hardware for solving from-region visibility in two dimensions by utilizing a dual ray space. They need to artificially shrink all polygons to compensate the lack of conservative rasterization. Additionally, several other papers utilize conservative rasterization, e.g., Durand et al. [4]. Thus, it should be clear that there is a need for such rasterization algorithms. To our surprise, those algorithms are rarely described in any detail, and furthermore, we are not aware of any graphics hardware that exposes conservative rasterization.

Some authors have implemented conservative rasterization simply by moving the edges of the triangle either inwards or outwards by $\sqrt{2}/2$ pixels. Unfortunately the technique suffers from two problems. First, it fails to find all pixels that are contained by an underestimated triangle. Second, the size of an overestimated triangle is exaggerated, especially at sharp corners.

In this paper, we present the details of a conservative rasterization algorithm based on edge functions [12]. It can be used in both hardware and software. An advantage of this algorithm is that it requires only a small modification to the triangle setup of the rasterizer, and that the remaining parts of the pipeline are left unmodified. Furthermore, we show that the same algorithm can be used for tiled rasterization, which is used to improve memory coherence [10], to do simple forms of culling [2, 11], and for different types of analysis to accelerate rendering [1]. The algorithm allows enabling conservative rasterization separately for each edge.

2 Rasterization using Edge Functions

The majority of rasterizers use edge functions [12] for rasterizing a triangle. The theory of edge functions is briefly reviewed in this section.

Assume that a triangle $\Delta\mathbf{pqr}$ shall be rasterized, and that the points $\mathbf{p}$, $\mathbf{q}$, and $\mathbf{r}$ are two-dimensional points in screen space. Each edge of the triangle defines an edge function $e(\mathbf{s})$, which is simply a line equation in implicit form. The edge function for the edge $\mathbf{pq}$ is

$$e_{\mathbf{pq}}(\mathbf{s}) = (q_y - p_y, -(q_x - p_x)) \cdot (\mathbf{s} - \mathbf{p}) = \mathbf{n} \cdot (\mathbf{s} - \mathbf{p}) = \mathbf{n} \cdot \mathbf{s} + c, \tag{1}$$

where $c = -\mathbf{n} \cdot \mathbf{p}$, and $\mathbf{n} = (n_x, n_y)$ is the normal vector of the line. A point $\mathbf{s}$ is inside the triangle if $e(\mathbf{s}) \leq 0$ for all edge functions.[1] This assumes that the vertices are in counter-clockwise order.

An advantageous property of edge functions is that testing the neighboring pixels is inexpensive during rasterization. If we have tested a pixel at $\mathbf{s} = (s_x, s_y)$, then $e(\mathbf{s})$ has already been evaluated. To test the pixel to the right of $\mathbf{s}$, we want to evaluate $e(s_x + 1, s_y)$, which is done as follows:

$$e(s_x + 1, s_y) = \mathbf{n} \cdot (\mathbf{s} + (1,0)) + c = e(\mathbf{s}) + n_x. \tag{2}$$

As can be seen, the edge function can be updated using a single addition. Similar updates are possible for traversing in the negative x-direction, and in both y-directions. When rasterizing a triangle, one needs to store at least one scalar for each of $e_{\mathbf{pq}}$, $e_{\mathbf{qr}}$, and $e_{\mathbf{rp}}$, and also the normal vectors seen in Equation 1 for each edge.

3 Modification of Triangle Setup

In this section, we describe how the triangle setup can be modified in order to enable conservative and tiled rasterization. A tile is a block of $w \times h$ pixels, and a pixel can be seen as a small tile.

In tiled rasterization, a tile is excluded if the entire tile is outside at least one of the three edge functions or if the tile is entirely outside the axis-aligned bounding rectangle of the triangle. First, we notice that to test a three-dimensional axis-aligned bounding box against a plane, it suffices to test the two corners of the box that define a line segment, which most closely aligns with the normal vector of the plane [6]. Second, Hoff [7] has suggested that one needs to test only one of these points to determine if the entire box is either in the positive or negative half-space. Similarly, our algorithm is based on the fact that we need to evaluate only one corner of the tile per edge function in order to determine if the tile is outside that edge function.

We want to evaluate the edge function, $e(\mathbf{s}) = \mathbf{n} \cdot \mathbf{s} + c$, at exactly one corner of the tile. For overestimating rasterization the correct corner is the one whose dot product with the normal vector of the edge is the smallest, as illustrated in Figure 2. However, our technique uses a more efficient method, and for that we define the following:

[1]This is not entirely true since a pixel center $\mathbf{s}$ can lie exactly on an edge shared by two triangles. McCool et al. [9] present a simple solution to this problem based on the signs of n_x and n_y.

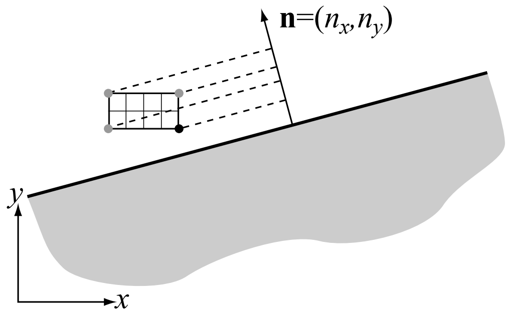

Figure 2: In this example of overestimating rasterization, the tile consists of 4×2 pixels. To determine at which corner of the tile the edge function should be evaluated, the four corners are projected onto the normal vector of the edge. The corner that corresponds to the smallest dot product between the normal and the corner is selected. In this case it is the black corner. Note that the gray half-space is potentially inside the triangle.

$$b_k = \begin{cases} 0, & n_k \geq 0 \\ 1, & n_k < 0 \end{cases}, \tag{3}$$

where b_k is a single bit, and $k \in [x, y]$. Assume that we have evaluated the edge function, e, for the lower left corner of the tile. Depending on the value of $\mathbf{n}$, we might need to add n_x and/or n_y to e. Revisit Figure 2 for an example: since $n_y > 0$, one of the lower two corners must be used. Furthermore, $n_x < 0$, and thus it can be concluded that the lower right corner must be used.

Given that e has been evaluated at the lower left corner of a tile, we use the following equation to compute a correct initialization of e for tiled rasterization:

$$e := e + b_x \times (wn_x) + b_y \times (hn_y) = e + b_x \times t_x + b_y \times t_y, \tag{4}$$

where w is the width, and h is the height of the tile in pixels. The $\times$-operator should be interpreted as follows: $b \times n = 0$ if $b = 0$, and $b \times n = n$ if $b = 1$. Equation 4 is very inexpensive to evaluate as it requires only two additions and multiplexing for determining whether to add the term t_k or not. Furthermore, the extra computations are performed only once per triangle.

The presented technique can also be used to determine the pixels or tiles that are fully inside a triangle. This underestimated footprint of the triangle is obtained by simply selecting the "diagonally opposite" corner of the tile when initializing the edge function.

There are many different ways of traversing the pixels/tiles of a triangle, and our technique can be used with any of those as long as edge functions are involved, and therefore we omit a discussion of any particular traversal method.

4 Discussion

The presented algorithm is so simple, and requires so few modifications to an existing rasterizer that we hope the algorithm will find its way to a hardware implementation. The only modification needed into an OpenGL-alike API is the support for selecting one of the two conservative rasterization modes, and ideally a possibility for enabling the selected mode separately for each edge.

With overestimating rasterization, one needs to be careful not to sample depth and other attributes outside the triangle. One solution for this is to silently clamp the barycentric coordinates to be inside the triangle before proceeding with pixel shading.

It is possible that current graphics chips use the presented technique internally for z_{min}/z_{max}-culling, but no document seems to verify that hypothesis. After we developed and implemented our algorithm in software, it came to our knowledge that a similar algorithm has been briefly mentioned in the US Patent no. $6,480,205$ "Method and apparatus for occlusion culling in graphics systems" by Greene and Hanrahan.[2] Their presentation lacks the details presented here, and furthermore, it is not at all well-known in the computer graphics community.

Acknowledgement Thanks to Jaakko Lehtinen, Lauri Savioja and Jacob Ström for proofreading.

References

[1] Aila, Timo, Ville Miettinen, and Petri Nordlund, "Delay Streams for Graphics Hardware," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 792–800, 2003.

[2] Akenine-Möller, Tomas, and Jacob Ström, "Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 801–808, 2003.

[3] Crow, Franklin C., "Shadow Algorithms for Computer Graphics," *Computer Graphics (SIGGRAPH '77 Proceedings)*, vol. 11, no. 2, pp. 242–248, July 1977.

[4] Durand, Frédo, George Drettakis, Joëlle Thollot and Claude Puech., "Conservative Visibility Preprocessing using Extended Projections," *Proceedings of ACM SIGGRAPH 2000*, pp. 239–248, 2000.

[5] Govindaraju, Naga K., Stephane Redon, Ming C. Lin, and Dinesh Manocha, "CULLIDE: Interactive Collision Detection between complex Models in Large Environments using Graphics Hardware," *Graphics Hardware 2003*, pp. 25–32, July 2003.

[6] Haines, Eric, and John Wallace,"Shaft Culling for Efficient Ray-Traced Radiosity" in P. Brunet and F.W. Jansen, eds., *Photorealistic Rendering in Computer Graphics (Proceedings of the Second Eurographics Workshop on Rendering)*, Springer-Verlag, pp. 122–138, 1994.

[2]They did not patent conservative rasterization but rather an algorithm for occlusion culling.

[7] Hoff., Kenneth E., III,"A Faster Overlap Test for a Plane and a Bounding Box, 1996. http://www.cs.unc.edu/ hoff/research/vfculler/boxplane.html

[8] Koltun, Vladlen , Daniel Cohen-Or, and Yiorgos Chrysanthou, "Hardware-Accelerated From-Region Visibility Using a Dual Ray Space," *12th Eurographics Workshop on Rendering*, pp. 204–214, 2001.

[9] McCool, Michael D., Chris Wales, and Kecin Moule, "Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization," *Graphics Hardware 2001*, pp. 65–72, 2001.

[10] McCormack, Joel, and Robert McNamara, "Tiled Polygon Traversal using Half-plane Edge Functions," *Graphics Hardware 2000*, pp. 15–21, 2000.

[11] Morein, Steve, "ATI Radeon HyperZ Technology," *Workshop on Graphics Hardware, Hot3D Proceedings*, August 2000.

[12] Pineda, Juan, "A Parallel Algorithm for Polygon Rasterization," *Computer Graphics (SIGGRAPH '88 Proceedings)*, vol. 22, no. 4, pp. 17–20, August 1988.