DOI: 10.1111/cgf.12061
EUROGRAPHICS 2013 / I. Navazo, P. Poulin
(Guest Editors)
Volume 32 (2013), Number 2

Analytic Visibility on the GPU

T. Auzinger[‡], M. Wimmer and S. Jescke

Vienna University of Technology, Austria

Abstract

This paper presents a parallel, implementation-friendly analytic visibility method for triangular meshes. Together with an analytic filter convolution, it allows for a fully analytic solution to anti-aliased 3D mesh rendering on parallel hardware. Building on recent works in computational geometry, we present a new edge-triangle intersection algorithm and a novel method to complete the boundaries of all visible triangle regions after a hidden line elimination step. All stages of the method are embarrassingly parallel and easily implementable on parallel hardware. A GPU implementation is discussed and performance characteristics of the method are shown and compared to traditional sampling-based rendering methods.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Antialiasing I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Visible line/surface algorithms

1. Introduction

An essential task in rendering a 3D scene to a 2D image is the determination of the (partial) visibility of the scene objects. Object visibility exhibits discontinuities at object silhouettes, causing infinitely high frequencies in the output. In order to suppress the resulting aliasing artifacts when rendering to an image of finite resolution, it is necessary to apply low-pass filtering on the scene data. To address this issue, one choses a suitable filter and performs a convolution with the scene data. This is especially relevant in current and future rendering and visualization tasks, due to increasing model complexity.

The standard method in depth-buffered rasterization is to approximate the convolution integral by choosing for every output pixel a single or multiple sampling locations. A weighted sum of the sample values gives the final pixel value. If both the visibility and scene data are evaluated at each sample point, we have common *supersampling*. If we just sample the visibility data at each sampling location and choose a lower number of scene samples, we obtain *multisampling* [Ake93]. In both methods, the choice of the sample locations and weights is crucial for the visual quality [RKLC*11].

A different approach to the aliasing problem is to compute the filter convolution *analytically*, i.e., to obtain a symbolic formula for the result and supply the scene data as parameters. A sketch of such a system for CPUs was developed by Catmull already in 1984 [Cat84]. In recent years we saw works on the exact computation of the convolution of polygonal data with box filters by Manson and Schaefer [MS11] and of polygonal and polyhedral data with radial filters by Auzinger et al. [AGJ12]. Both methods, while efficiently implementable on GPUs, do not address scene visibility.

In this paper, we try to close this gap and present two new algorithms that allow for the efficient computation of analytic visibility on parallel hardware. Together with the methods above, fully analytic anti-aliased 3D scene display is enabled and, by utilizing the improved programmability of recent graphics hardware, interactive frame rates can be achieved.

2. Related Work

Analytic visibility methods were developed early in the history of computer graphics, with the first hidden line/surface elimination algorithms by Appel [App67] and Robert [Rob63]. Sutherland et al. [SSS74] presented an excellent survey of other methods and their close relationship to sorting [SSS73].

‡ thomas.auzinger@cg.tuwien.ac.at

Plenty of early algorithms exist for the elimination of hidden lines [Gal69, Hor82] or general curves [EC90]. Line drawings depend in general on these techniques, with the first halo rendering presented by Appel et al. [ARS79], and recent development covered in the course by Rusinkiewicz et al. [RCDF08].

Extensions to analytic hidden surface elimination were conducted by Weiler et al. [WA77], who clips polygonal regions against each other until trivial depth sorting is obtained, by Franklin [Fra80], who uses a tiling and blocking faces to establish run-time guarantees, and by Catmull [Cat78, Cat84], who sketches a combination of analytic visibility and integration for CPUs. An early parallelization is described by Chang et al. [CJ81]. McKenna was the first to rigorously show a worst-case optimal sequential algorithm with complexity $O(n^2)$ in the number of polygons [McK87]. Improvements were made by Mulmuley [Mul89] and Sharir et al. [SO92] with the goal of output-sensitive algorithms, i.e. to base the run-time complexity on the actual number of intersections between the polygons. One of the first parallel algorithms was the terrain visibility method by Reif and Sen [RS88], later improved by Gupta and Sen [GS98]. The general setting was treated by Randolph Franklin et al. [RFK90] but with worst case asymptotics independent of processor count. Once hardware limitations disappeared, the focus of the community moved to approximate, sampling-based visibility. Raytracing and its variants rely on object space visibility, e.g. space partitioning hierarchies, while rasterization mainly uses the z-buffer methodology; an overview is given in the course by Durand [Dur00].

The computational geometry community showed continued interest in analytic visibility, and in recent years Dévai gave an easier-to-implement optimal sequential algorithm and an optimal parallel algorithm that runs in $\Theta(\log n)$ using $n^2/logn$ processors in the CREW PRAM model (Concurrent Read Exclusive Write Parallel Random Access Machine) [Dév11]. However, such optimal algorithms use intricate data structures and are highly non-trivial to implement on actual GPU hardware.

As our method provides the correct input for analytic anti-aliasing methods, we give a short overview of this field here. The aliasing problem in computer graphics was rigorously treated for the first time by Crow [Cro77] and over the years various filters have been proposed by Turkowski [Tur90], Mitchell and Netravali [MN88] and others, but simple box filtering still stays relevant for current analytic methods such as wavelet rasterization by Manson et al. [MS11]. While different approaches to (semi-)analytic anti-aliasing have been proposed throughout the years, e.g. by McCool [McC95] and Guenter and Tumblin [GT96], sampling is still the preferred method, either in its stochastic variant originated by Dippe et al. [DW85] or with (semi)-regular sampling patterns. Especially the latter is still a research focus in stratified Monte-Carlo methods and GPU rasterization; the course by Keller et al. [KPRG12] give an overview. Analytic methods show an increase in popularity in the field of *motion blur* and *depth of field* rendering; Gribel et al. use exact visibility along cuts through the scene in depth direction for point [GDAM10] and line [GBAM11] samples for stochastic sampling on the CPU. Auzinger et al. [AGJ12] use analytic integration to obtain anti-aliased sampling of 2D and 3D scenes on the GPU.

3. Analytic Visibility

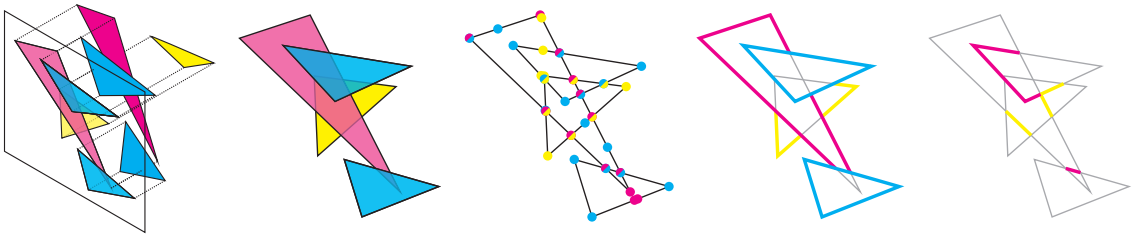
Adapting a visibility algorithm to massive SIMD architectures has many objectives. Ideally it allows the workload to be split into a predictable and large number of small and independent subtasks. Furthermore it should accommodate simple data structures that can be accessed in coalesced parallel fashion. Additionally, the actual computations should rely on basic data types such as integral or floating point values. These restriction rule out methods that internally use arbitrary polygons or generate complex graphs, as well as arbitrary precision arithmetics. Furthermore, acceleration data structures are needed to avoid the $O(n^2)$ complexity of intersecting all triangles with each other. Visibility algorithms generally contain a sorting routine, which has to be mapped to efficient methods on the GPU.

Our visibility algorithm takes the normalized device coordinates of all 3D triangles as input and outputs a list of all 2D line segments, which constitute the borders of all visible polygonal regions of the scene when projected onto the view plane. It is assumed that the triangles are *consistently orientated* (to allow backface culling) and *non-intersecting* in 3D. Note that *cyclic overlap* of triangles is permitted. The result gives a complete description of the boundaries of all visible regions and matches the input requirements of analytic anti-aliasing approaches [MS11, AGJ12].

Our method performs the visibility computation independently for each edge, which yields an embarrassingly parallel workload. Building on a method of Dévai [Dév11], we present a new edge-triangle intersection method that allows replacing the theoretical 'black hole' treatment of Dévai with a novel, implementation-friendly boundary completion stage. Both algorithms and the additional hidden line elimination stage are presented in the next sections.

3.1. Edge Intersections

To determine the visible parts of the scene edges, we first project them onto the view plane and compute the intersection in this space. Given a projected edge e and a projected triangle t, we can assume that an occlusion, if it exists, is a connected set of e, i.e. a line segment in the view plane, due to the convexity of triangles. This well-defined output enables efficient parallel calculations of all occlusions between edges and triangles. A high-level view of this method is given in algorithm 1 with an example in figure 2.



(a) View plane projection (b) 2D view (c) Edge intersections (d) Hidden line elimination (e) Boundary completion

Figure 1: *Overview of our analytic visibility method. The scene triangles are projected onto the view plane (a) since the* edge intersection *phase (see section 3.1) operates mostly in 2D (b). It determines for each edge the (possible) intersections with all other triangles (c). The intersection data is used in the second phase to determine the* visible line segments *(see section 3.2). A* hidden line elimination *algorithm gives the visible segments of each edge (d) and a final* boundary completion *(e) completes all visible line segments. These segments are the boundaries of the visible regions of the scene triangles.*

Algorithm 1 Edge intersection phase

Input: T is the set of all scene triangles. E is the set of all their edges.

Output: *Data* is the resulting intersection data, where $Data(e)$ gives the intersection data for edge e. Each entry of $Data(e)$ is the 3-tuple $(p, flag, type)$ consisting of an intersection point p, a flag that indicates if p is *starting* or *ending* a line segment, and a *type* describing the relative depth relation between e and p (i.e. one of *occluding*, *occluded*, or *self*).

```
 1: procedure EdgeIntersections(E, T)
 2:     Data ← {}
 3:     for all edge e ∈ E do in parallel
 4:         Data(e) ← {}
 5:         for all triangle t ∈ T, e ∉ t do in parallel
 6:             if AreOverlapping(e, t) then
 7:                 (p0, p1) ← Intersect(e, t)
 8:                 if IsBehind(e, p0, p1) then
 9:                     type ← occluded
10:                 else
11:                     type ← occluding
12:                 end if
13:                 AppendTo(Data(e), (p0, starting, type))
14:                 AppendTo(Data(e), (p1, ending, type))
15:             end if
16:         end for
17:         type ← self
18:         (p0, p1) ← EndPoints(e)
19:         AppendTo(Data(e), (p0, starting, type))
20:         AppendTo(Data(e), (p1, ending, type))
21:     end for
22:     return Data
23: end procedure
```

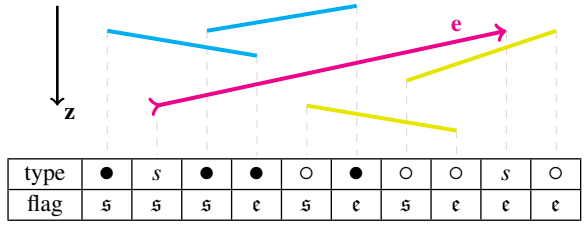
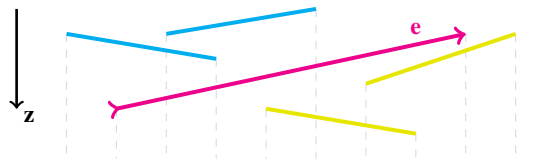


Figure 2: *Example result of the* EdgeIntersections *procedure (see algorithm 1) for a single edge* **e** *(in magenta). The plane which contains* **e** *and extends into z direction intersects two triangles in front of e (in blue) and two triangles behind e (in yellow). All four triangles overlapping e and the resulting intersection points are the endpoints of the associated line segments. Each point is associated with a type, denoting if e is occluded by its triangle (●), if it is an endpoint of e (s), or if its triangle is occluded by e (○). Additionally, a flag stores if the intersection points starts (𝔰) or ends (𝔢) an occlusion.*

The core routines Intersect and IsBehind need to be further discussed. The intersection routine Intersect reduces to a geometrical computation in 2D, as both its input parameters, edge e and triangle t, are projected onto the view plane. AreOverlapping is a conservative test to discard triangles that do not intersect e. In general we expect the intersection to consist of a whole line segment; Intersect outputs the segment's end points in 3D, where the depth coordinate is chosen such that the point lies on the plane of t. It should be noted that we discard single-point intersections in this phase, as zero-length line segments do not contribute to the final image. Additionally, each endpoint is marked as either *starting* or *ending* the occlusion.

A hidden line elimination algorithm considers only the occlusion of edge e [App67]. In our case, as we want full hidden surface removal, we have to take into account all the triangles that are occluded by e, too. IsBehind compares



type	●	*s*	●	●	○	●	○	○	*s*	○
flag	𝔰	𝔰	𝔰	𝔢	𝔰	𝔢	𝔰	𝔢	𝔢	𝔢
init	1	-1	1	-1	✗	-1	✗	✗	1	✗
scan	1	0	1	0	✗	-1	✗	✗	0	✗
out	✗	✗	✗	✗	✗	✓	✗	✗	✓	✗

Figure 3: *Example result of the* HIDDENLINEELIMINATION *procedure (see algorithm 2) for the scene in figure 2. The table gives type and flag information after the edge intersection phase and at various stages of the hidden line elimination algorithm. The state of V after the initialization (after line 20) is shown in row* init*, while the state of S after the inclusive scan (after line 21) is given by* scan*. The entry ✗ denotes a removed index. The check marks ✓ in row* out *show which intersection points are reported as visible edge segments.*

the relative depth of an edge e and the line segment given by p_0 and p_1. Since we assume non-intersecting triangles as input, their depth ordering can be determined and we store this information in *type*.

Great care has to be taken to ensure the robustness of such geometrical calculations on the used hardware. Fixed precision or floating point arithmetics lead to round-off errors and can cause erroneous results in binary geometric decisions (e.g. is a point on a line?). *Exact geometric computation* is the most commonly used technique to solve this problem. But since it relies on a combinatorial analysis of the geometric calculation at hand and on the use of arbitrary precision numbers, it is not suited for practical implementation on graphics hardware. We chose an ε-prefiltering approach by Frankel et al. [FNS04] and compute parallelism queries of lines in double precision. This enables us to reliably determine the correct intersections for geometrically complex models (see figure 5).

Algorithm 2 Determine the visible parts of all scene edges

Input: The *sorted* output of EDGEINTERSECTIONS (see algorithm 1).

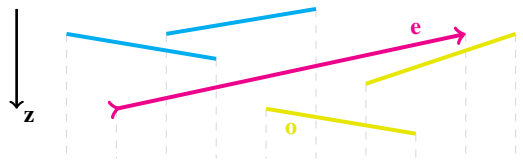
Output: REPORTSEGMENT outputs the visible line segments of all edges.

```
1:  procedure HIDDENLINEELIMINATION(E)
2:      for all e ∈ E do in parallel
3:          I ← Data(e)
4:          for n ← 1,|I| do in parallel
5:              (∼,∼,type) ← I(n)
6:              if type = occluding then
7:                  REMOVEINDEX(I,n)
8:              end if
9:          end for
10:         for n ← 1,|I| do in parallel
11:             (∼,flag,type) ← I(n)
12:             if flag = starting then
13:                 V(n) ← 1
14:             else
15:                 V(n) ← −1
16:             end if
17:             if type = self then
18:                 V(n) ← −V(n)
19:             end if
20:         end for
21:         S ← INCLUSIVESCAN(V)
22:         for n ← 1,|I| do in parallel
23:             v ← V(n), s ← S(n)
24:             if (v = 1 ∧ s > 0) ∨ (v = −1 ∧ s > −1) then
25:                 REMOVEINDEX(I,n)
26:             end if
27:         end for
28:         for n ← 1,n ≤ |I|,n ← n+2 do in parallel
29:             (p0,∼,∼) ← I(n)
30:             (p1,∼,∼) ← I(n+1)
31:             REPORTSEGMENT(p0,p1)
32:         end for
33:     end for
34: end procedure
```

3.2. Visible Line Segments

Having obtained the intersections for each scene edge, we employ two procedures to determine all visible line segments necessary for the final integration stage. We present this as two separate methods. The first is a simple *hidden line elimination* algorithm (see figure 1d) while the second method completes the boundaries of the visible regions of each triangle and thus provides full *hidden surface elimination* (see figure 1e).

Hidden line elimination is a standard technique in line rendering and was first introduced by Appel [App67]. We adapt a recent result of Dévai [Dév11] on the optimal runtime of parallel hidden surface algorithms for our method. We use a scan-based approach which walks along the line of a given edge e and determines for each intersection point if it is an endpoint of a visible segment of e. This requires an ordering of the intersections along e, which is achieved by a *intermediate sorting step*. An overview of hidden-line elimination assuming a sorted input is given by algorithm 2 and an example in figure 3. For each edge in parallel, we first remove all data from intersections that originate from triangles that lie behind the edge (lines 4-9). Depending on the type of each intersection point, we assign a value (0 or ± 1) to a list V (lines 10-20). The following INCLUSIVESCAN creates a list S that holds a measure on how many triangles occlude e



type	●	*s*	●	●	○	●	○	○	*s*	○
flag	𝔰	𝔰	𝔰	𝔢	𝔰	𝔢	𝔰	𝔢	𝔢	𝔢
vis	F	F	F	F		T			T	
front							T			T
init	0	0	0	0	-1	-1	1	1	1	-1
scan	1	1	1	1	0	-1	0	1	2	1
out	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗

Figure 4: *Example result of the* BOUNDARYCOMPLETION *procedure (see algorithm 3) for the scene in figure 2. The table shows type and flag information after the edge intersection phase, and the program's state at various stages for a selected iteration step in n, such that the occluded line segment* **o** *(see figure above) is given by the intersections with indices n_{start} and n_{end}. The boolean values of Vis after its initialization (after line 6) are given by the row* vis*, while the results of the* INFRONT *check can be found in row* front*. The last three rows show the resulting values of V, S and* REMOVEINDEX *(consult figure 3).*

at a given intersection. We remove all occluded edge segments (line 25) and report the remaining ones as endpoints of visible edge segments. The criterion for removal (line 24) reflects the fact that for a given intersection index n a visible edge segment starts at n, if $V(n) = -1$ and $S(n) = 1$. The end of such a segment is given by $V(n) = 1$ and $S(n) = 0$.

To complete the boundary of each visible region, we propose an extension to the parallel hidden line elimination above. In conjunction, they give a full hidden surface elimination. Intuitively, we want to determine which triangles lie 'on the other side' of the visible edge segments (figure 1e shows the missing segments that complete the visible edge segments of figure 1d to yield the boundaries of the visible regions in figure 1b). This is a harder problem than hidden line elimination, as we are interested in the triangles that are right behind a given edge e (as opposed to just knowing that e lies behind one or more triangles). Hence our algorithm 3 executes a modified hidden line elimination for every line segment o that is occluded by e (lines 7-39). The initialization of V is changed in such a way that the visible parts of e act as anti-occlusions, i.e. all segments o are occluded unless they lie behind a visible part of e (see figure 4). Each intersection along e is assigned a value (0 or ± 1) (lines 21-24), perform an inclusive scan on the list (line 25), remove invisible line segments (lines 27-32), and report the remaining segments (lines 33-37).

In the initialization phase we have to resolve the relative depth layering of the occluded line segments, too. Our algorithm achieves this by pairwise comparison of the line seg-

Algorithm 3 Determine the missing boundaries of the visible regions

Input: The *sorted* output of EDGEINTERSECTIONS (see algorithm 1).

Output: REPORTSEGMENT outputs the missing line segments.

```
1:  procedure BOUNDARYCOMPLETION(E)
2:      for all e ∈ E do in parallel
3:          I ← Data(e)
4:          for n ← 1,|I| do in parallel
5:              Vis(n) ← INTERSECTIONISVISIBLE(I,n)
6:          end for
7:          for n ← 1,|I| do
8:              (∼,flag,type) ← I(n)
9:              if type = occluding ∧ flag = starting then
10:                 ID ← GETTRIANGLEID(T,e,n)
11:                 (n_start,n_end) ← GETINDICES(I,ID)
12:                 for k ← 1,|I| do in parallel
13:                     if k ≠ n_start ∧ k ≠ n_end then
14:                         (∼,∼,type_k) ← I(k)
15:                         if type_k = occluding then
16:                             ID_k ← GETID(T,e,k)
17:                             F(k) ← ISINFRONT(ID_k,ID)
18:                         end if
19:                     end if
20:                 end for
21:                 for k ← 1,|I| do in parallel
22:                     (∼,flag_k,type_k) ← I(k)
23:                     V(k) ← INIT(k,n_start,n_end,flag_k,type_k,Vis(k),F(k))
24:                 end for
25:                 S ← INCLUSIVESCAN(V,1)
26:                 I_n ← I(n)
27:                 for k ← 1,|I| do in parallel
28:                     v ← V(k),s ← S(k)
29:                     if (v = 1 ∧ s > 0) ∨ (v = −1 ∧ s > −1) then
30:                         REMOVEINDEX(I_n,k)
31:                     end if
32:                 end for
33:                 for k ← 1,k ≤ |I|,k ← k+2 do in parallel
34:                     (p_0,∼,∼) ← I_n(k)
35:                     (p_1,∼,∼) ← I_n(k+1)
36:                     REPORTSEGMENT(p_0,p_1)
37:                 end for
38:             end if
39:         end for
40:     end for
41: end procedure
```

```
42: procedure INIT(n, n_start, n_end, flag, type, visible, inFront)
43:     if n = n_start then return −1 end if
44:     if n = n_end then return 1 end if
45:     if type = occluding ∧ inFront = true then
46:         if flag = ending then
47:             return −1
48:         else [flag = starting]
49:             return 1
50:         end if
51:     end if
52:     if visible = true then
53:         if (type = self ∧ flag = starting) ∨ (type = occluded ∧ flag = ending) then
54:             return −1
55:         end if
56:         if (type = self ∧ flag = ending) ∨ (type = occluded ∧ flag = starting) then
57:             return 1
58:         end if
59:     end if
60:     return 0
61: end procedure
```

ments which are subsets of their respective triangles. This is a standard geometric computation and is denoted by the procedure ISINFRONT in line 17. As already mentioned, we only report line segments that are only occluded by e. This requires the knowledge of all visible line segments of e, which is exactly the result of HIDDENLINEELIMINATION. We abbreviate it with the call of INTERSECTIONISVISIBLE in line 5.

4. Implementation

Our analytic visibility method targets massively parallel SIMD architectures with current GPUs as a prime example. We make use of their large amount of moderately sized SIMD units by implementing a software rendering pipeline on NVidia hardware using the CUDA C programming language [NVI]. We give a short review of this environment and continue with a detailed explanation of our design choices.

4.1. Hardware

We use NVidia's nomenclature and refer to the SIMD units as *warps* and assume their size to be 32 threads. They are grouped into *thread blocks* which enables the use of a processor's fast on-chip memory as *shared memory* by all the block's threads. The much larger and considerably slower global memory allows data transfer and synchronization across thread blocks, where the latter is enabled by atomic memory accesses. The amount of registers and shared memory is limited and the excessive use of one resource can decrease the amount of warps that can run in parallel, leading to device underutilization. While our design can also be implemented on other parallel hardware, such as multi-core CPUs, our algorithm benefits from the huge amount of threads that are kept active by a GPU, thus hiding memory latency.

4.2. Design Considerations

Our algorithm shows a distinct two-level parallelism in all its core procedures (see algorithm 1-3). The outer loop iterates over all edges in parallel. Since the number of intersections per edge varies greatly, we cannot employ a SIMD model at this level without incurring a severe under-utilization penalty. Therefore, we assign edges to separate SIMD units and parallelize the inner loops across their threads.

In the edge intersection phase (see section 3.1) we assign a given edge to a warp and fetch a triangle per thread until the pool of relevant triangles is depleted. Both the assignment of edges to warps and the computations of the offset into the output array are done with atomic memory functions on global counters. In our tests the large number of active warps efficiently hide the memory latency associated with the triangle fetches.

The sorting of intersections along the edges is executed for all edges in parallel by using a key-value radix sort. The key of each intersection holds a reference to its edge and a parameter that increases along the edge. Knowing the number of intersections per edge allows efficient retrieval of the sorted intersection for each edge.

The hidden line elimination and boundary completion are executed similar to the edge intersection. The edges are assigned to warps and the inner loops parallelized across their threads.

4.3. Analytic Visibility Pipeline

In this section we will provide an overview of the actual pipeline we implemented and essential details of the adaptation to the CUDA framework as well as vital optimizations. Our algorithm to intersect edges with triangles has an outer loop over all edges and an inner loop over all triangles. A quadratic complexity in the number of triangles will become prohibitively costly for large scenes and thus we assign the scene triangles to subsets of the view plane – in our case quadratic *bins*. This preserves spatial coherence in the memory accesses and enables load balancing by prioritizing bins with a high number of assigned triangles. We generate a list of overlapped bins per triangle and then use a fast radix sort [MG11] to obtain a list of triangles per bin (similar to the work on voxelization by Pantaleoni [Pan11]). Our use of fixed bins can be improved upon by employing an adaptive scene subdivision to enhance the load balancing.

Furthermore we accelerate the procedure AREOVERLAPPING (see algorithm 1 line 6) by assigning an axis-aligned

bounding box to each projected scene triangle. By quantizing the box coordinates we compress the full description into 32 bits. A fast rejection of non-overlapping triangles can be executed with just a few compare operations.

As each thread handles the intersection of an overlapping triangle with the warp's edge, we obtain either two or zero intersections per thread. The final storage address in global memory is the sum of two offsets. A global offset which is acquired on a per-warp basis via global memory atomics and a per-thread offset which is obtained by computing a warp-wide scan of the number of intersections. Sorting the intersections along each edge is achieved by a key-value sort which arranges all intersections according to edge index and position along their respective edge. The edge index (a 32 bit integer) and the position along the edge (a 32 bit float) of each intersection is combined into a 64 bit radix sortable key. We use the radix sort method of the *thrust* library [BH11] in our implementation.

The core parts of both algorithm 2 and 3 are the initialization, execution and evaluation of INCLUSIVESCAN. For a given edge e with n intersections, the number of values which have to be scanned can be up to n. A first approach would be to store these values in fast shared on-chip memory and execute the scan over the whole array. Shared memory is a very limited resource ($\sim$ 48kB per SM) and thus only a limited amount of values can be stored before the number of warps per SM, which can be launched in parallel, is significantly reduced. Storing the intermediate values in the much larger global memory is prohibitively expensive in terms of memory access times. Our solution is to conduct the inclusive scan in parallel for chunks of warp size and execute the chunks sequentially. This allows the full utilization of the GPU's SIMD parallelism with a small shared memory footprint.

The procedure REPORTSEGMENT uses the same scan method to obtain the correct offset in order to write the line segments into the output buffer. As before, the line segments are output in a non-deterministic fashion and we again employ a radix sort to assign the segments to their respective edges. This list of line segments constitutes the output of our analytic visibility method and provides the necessary information to employ an analytic anti-aliasing method.

4.4. Analytic Integration

We implemented the analytic sampling of Auzinger et al. [AGJ12] to render the final output image using CUDA C. It should be noted that the input to this stage is an unordered list of line segments per tile. A reconstruction of the visible regions is not needed explicitly, since the integration is evaluated over their boundary segments. As shown in algorithm 4, we again employ a two-level parallelization. The output image is subdivided into tiles that are assigned to the warps of the GPU. Each warp alternately executes two distinct stages; in the first stage each thread loads an input line segment whereas in the second stage, each thread computes the contribution of all loaded segments to a single pixel of the tile. This reduces the shared memory needs as all input segments reside in the threads' registers, and beginning with the Kepler architecture of NVidida GPUs, register data can be shared by the threads of a warp without shared memory transfers. Once the contributions of all input line segments to the pixels of a tile are computed, the tile is written to the output texture in global memory. Only the assignment of tiles to the warps has to be synchronized, since access to a tile is exclusive for a single warp. As before, we use global memory atomics for this purpose.

Algorithm 4 Analytic filter convolution.

Input: L is the set of all visible line segments (with a reference to their respective triangle). L_τ denotes the subset of L which is relevant for tile τ of the output texture. F is the supplied convolution filter.

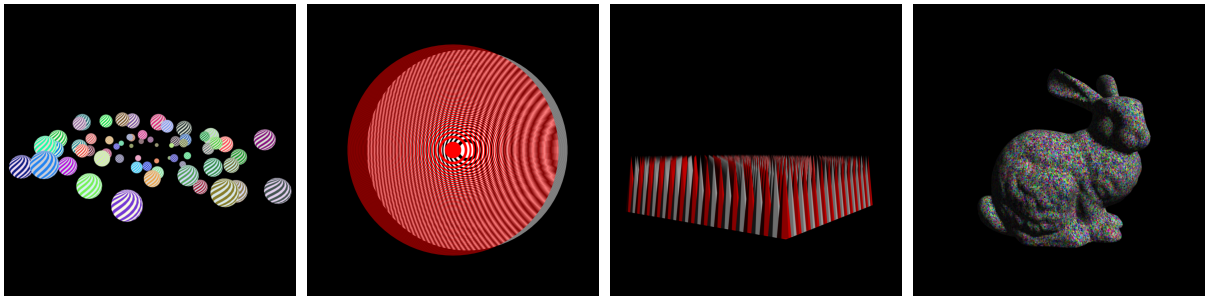
Output: WRITETILE writes a tile of the output texture.

```
1:  procedure INTEGRATION(L)
2:      for all tile τ do in parallel
3:          for all batch b in L_τ do
4:              for all pixel p ∈ τ do in parallel
5:                  for all segment l ∈ b do
6:                      τ(p) ← INTEGRATE(l, p, F)
7:                  end for
8:              end for
9:          end for
10:         WRITETILE(τ)
11:     end for
12: end procedure
```

While being exact, analytic integration methods suffer the drawback of requiring a mathematical description that can be evaluated symbolically, i.e. the associated integrals need to have a closed-form solution. Wavelet rasterization [MS11] integrates constant functions, while the method we apply also accommodates linear functions in screen space. Gouraud shading is linear in object space but is represented as a ratio of polynomials in screen space, due to perspective distortions. This complicates its analytic evaluation and it will not evaluate to polynomials. We leave this extension for future work.

5. Results

We evaluated the performance of our analytic visibility implementation on a GeForce GTX 680 GPU with 4GB RAM and a Core i7 CPU clocked at 2.67 Ghz and with 12GB RAM. The operating system was Windows 7 with the CUDA framework 4.2. Four scenes with different characteristics were used (see figure 5). ZONEPLATES exhibits very fine scale geometry and serves as a test for the numerical robustness of our method. As can be seen in figure 5b, even



(a) PLANETS (b) ZONEPLATES (c) SPIKES (d) BUNNY

Figure 5: *Our test scenes with low (b) & (d) to high (c) depth complexity and low (d) to high (b) & (c) geometrical detail. The images were generated with our method using a Gaussian filter kernel with a radius of 2.3 pixel [AGJ12] and a resolution of* 1024^2. *The* ZONEPLATES *scene (b) consists of two superposed zone plates while* SPIKES *is a regular grid of square pyramids.*

geometric intersections of subpixel scale of the two superposed zone plates are correctly resolved. SPIKES serves as stress test for a large edge intersection count, due to its high depth complexity. As standard scenes we use a stylized system of PLANETS and the Stanford BUNNY. All scenes were rendered with a Gaussian filter kernel with a radius of 2.3 pixel [AGJ12].

As a first step, we investigated the algorithm's behavior with different bin sizes (see section 4.3 for information on the bins). As the integration stage benefits from localized line segments, it executes fastest for the smallest bin size (in our case 8^2 pixel). However, the visibility stage shows the best performance at certain bin sizes relative to the image size, i.e. for a certain ratio between resolution and bin size (see table 1). Smaller bin sizes quadratically increase the number of bins that a given triangle is assigned to. This could cause multiple computation of the same intersection between an edge and a triangle in different bins, thus incurring a performance penalty. For too large bins, the number of triangles per bin increases and causes a quadratic increase in the number of intersection computations. The preferred ratio of the visibility phase for a given resolution is consistent across scenes and can be taken as a performance guideline. Due to the significant increase in computation time of the whole pipeline for bin sizes greater than 8^2, caused by the integration stage, we use bin size 8^2 for the following measurements.

Table 2 gives a detailed overview of the statistics and timings of our method when rendering different levels of detail (LoDs) of the BUNNY test scene. The first and fourth column show the triangle count of the model and the number of bins that had at least one triangle assigned to. All other columns give the execution time of the respective kernels in milliseconds. The values in brackets is the number of output elements of the column's computation. They are, from left to right, frontfacing triangles, bin-triangle-assignments, edge intersections, visible edge segments, and the line segments to complete the boundaries of the visible regions. The overhead column gives the total of all intermediate radix sorts, scans, and initializations. Column **Total** gives the total time of all GPU operations. The rendering of all LoDs at the given resolutions runs at interactive framerates and it can be seen that most of the runtime of the visibility stage is spent computing the edge intersections.

Scene	Resolution	Visibility (R/B)			Int (B)	
		128^2	64^2	32^2	8^2	16^2
BUNNY (70k)	512^2	✗	155	141	56	236
	1024^2	122	95	96	51	155
	2048^2	83	73	81	71	174
PLANETS (37k)	512^2	✗	121	111	59	229
	1024^2	94	74	77	31	141
	2048^2	64	56	63	40	100

Table 1: *Determination of the optimal bin size for our implementation. Columns* ***Vis (R/B)*** *give the timing of the analytical visibility for a bin size* ***B****, such that* ***B*** *times the* ***Resolution*** *equals the header value. The timings of the integration stage are given in columns* ***Int (B)****, with the header value equaling the bin size. Note that the visibility computation benefits from a certain ration of bin size and resolution, while the integration prefers the smallest bin size.*

ZONEPLATES and SPIKES illustrate the robustness of our geometric computations, and in table 3 we give an overview of their render timings. As expected, we see that the runtime of the visibility stage depends mainly on the number of generated edge intersections, while the integration procedure depends on the size of its line segment input.

An informal comparison with traditional sampling-based rasterization is given in figure 6 using multi pass hardware rasterization with DirectX. Although a substantial amount of samples is needed for scenes with high anti-aliasing re-

Tri. count	Setup	Resolution	Filled Bins	Edge intersections	Hidden line elimination	Boundary completion	Integration	Overhead	Total
7k	0.04 (3k)	512^2	918 (16k)	6.3 (0.5M)	0.4 (75k)	1.0 (25k)	7.4	6.4	25
		1024^2	3.4k (29k)	9.9 (0.6M)	0.8 (0.1M)	1.6 (33k)	9.1	7.8	34
		2048^2	13k (68k)	21 (1.2M)	1.6 (0.3M)	2.7 (55k)	16	8.8	56
26k	0.16 (13k)	512^2	910 (44k)	31 (1.5M)	1.1 (0.2M)	3.5 (87k)	23	9.4	75
		1024^2	3.4k (63k)	26 (1.9M)	1.6 (0.3M)	4.2 (0.1M)	21	10	71
		2048^2	13k (116k)	37 (2.7M)	2.8 (0.5M)	6.0 (0.1M)	33	13	100
70k	0.42 (30k)	512^2	906 (102k)	116 (3.9M)	2.6 (0.5M)	10 (0.3M)	56	16	212
		1024^2	3.3k (134k)	77 (4.5M)	3.4 (0.7M)	11 (0.3M)	51	18	172
		2048^2	12k (212k)	79 (5.9M)	5.3 (1.0M)	14 (0.4M)	72	22	208

Table 2: *Statistics from rendering the* BUNNY *model at different levels of detail with our method. See the text for details.*

Scene	Size	Intersections	Segments	Visibility	Integration
PLANETS	37k	3.7M	576k	94	31
SPIKES	5.0k	25M	973k	448	43
ZPLATE	14k	4.7M	1.6M	165	104
BUNNY	70k	4.5M	941k	122	51

Table 3: *Statistics from rendering the test scenes (see figure 5) with our method using a bin size of* 8^2 *and a resolution of* 1024^2. ***Size*** *gives the number of scene triangles,* ***intersections*** *the number of edge intersections, and* ***segments*** *the number of visible line segments. The timings of the complete* ***visibility*** *stage and the analytic* ***integration*** *are given in milliseconds. See the text for details.*

quirements, traditional sampling still has a runtime advantage over our method.

6. Conclusions and Future Work

We have presented an analytic visibility method to perform exact hidden surface removal on the GPU. We showed that with an adequate geometric computation scheme and adaptation to the two-level parallelism of SIMD architectures, it is possible to robustly perform analytic anti-aliased rendering of 3D scenes at interactive frame rates on GPUs. A possible future extension of our pipeline can be the use of dynamic load balancing with adaptive scene subdivisions in contrast to our static tiling. Future development in GPU architecture (e.g. *Dynamic Parallelism* of NVidia) seem to support such approaches.

As already mentioned in section 4.4, Gouraud shading or more complex shading variants were not treated so far due to their complicated (or possibly non-existent) closed form solutions. This could constitutes a worthwhile research avenue to generalize analytic anti-aliased rendering.

While sampling based rasterization proves hard to beat in terms of speed, we see our work as a first step to bring back analytic methods to rendering. The applications of our method are plenty and largely unexplored. Just employing the hidden-line elimination stage allows analytic line rendering. The output of our visibility stage gives a full description of the scene visibility from a viewpoint and can be used to generate an analytic visibility map, analytic shadow maps or direct rendering to vector graphics. Furthermore, the method does not depend on the final image resolution and will have advantages for large scale images. In the integration phase, the polynomial filter function can be altered on the fly, which allows the use of different filters in the same output image. This can be applicable for motion blur or depth-of-field effects. A change in the visibility algorithm could allow analytic depth peeling and support analytic anti-aliased transparency effects.

7. Acknowledgments

We want to thank the reviewers for their insightful and helpful remarks and Gernot Ziegler for providing help with CUDA. Funding was provided by the FWF grant P20768-N13.

References

[AGJ12] AUZINGER T., GUTHE M., JESCHKE S.: Analytic Anti-Aliasing of Linear Functions on Polytopes. *Computer Graphics Forum 31*, 2 (2012), 335–344. 1, 2, 7, 8

[Ake93] AKELEY K.: Reality engine graphics. SIGGRAPH '93, pp. 109–116. 1

[App67] APPEL A.: The notion of quantitative invisibility and the machine rendering of solids. ACM '67, pp. 387–393. 1, 3, 4

[ARS79] APPEL A., ROHLF F. J., STEIN A. J.: The haloed line effect for hidden line elimination. SIGGRAPH '79, pp. 151–157. 2

[BH11] BELL N., HOBEROCK J.: Thrust: A productivity-oriented library for cuda. In *GPU Computing Gems* (2011). 7

[Cat78] CATMULL E.: A hidden-surface algorithm with anti-aliasing. SIGGRAPH '78, pp. 6–11. 2

[Cat84] CATMULL E.: An analytic visible surface algorithm for independent pixel processing. SIGGRAPH '84, pp. 109–115. 1, 2

[CJ81] CHANG P., JAIN R.: A multi-processor system for hidden-surface-removal. *SIGGRAPH Comput. Graph. 15*, 4 (1981), 405–436. 2

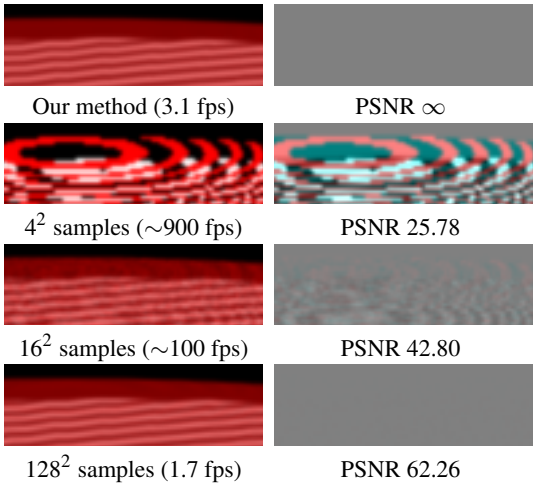


Figure 6: *Comparison of our method with massive supersampling at a resolution of* 1024^2*. The left column shows a detailed view of a* ZONEPLATES *rendering with our method (top) and with a supersampling approach with three different sampling densities. The sample count is per-pixel and the timings are for the full render cycle. The right column gives the corresponding difference images and the peak signal-to-noise ratio (PSNR) when compared with our rendering. The supersampling method uses stratified sampling and sample sharing by collecting the samples over multiple rendering passes. Note that for highly detailed models,* 16^2 *samples are not sufficient to faithfully approximate the Gaussian filter kernel that is evaluated analytically with our method. A break-even in terms of fps with our method is reached for approximately* 100^2 *samples and the reference solution with* 128^2 *samples gives near-identical results.*

[Cro77] CROW F. C.: The aliasing problem in computer-generated shaded images. *Commun. ACM 20*, 11 (1977). 2

[Dév11] DÉVAI F.: An optimal hidden-surface algorithm and its parallelization. ICCSA'11, pp. 17–29. 2, 4

[Dur00] DURAND F.: A multidisciplinary survey of visibility. In *ACM SIGGRAPH Courses* (2000). 2

[DW85] DIPPÉ M. A. Z., WOLD E. H.: Antialiasing through stochastic sampling. SIGGRAPH '85, pp. 69–78. 2

[EC90] ELBER G., COHEN E.: Hidden curve removal for free form surfaces. SIGGRAPH '90, pp. 95–104. 2

[FNS04] FRANKEL A., NUSSBAUM D., SACK J.-R.: Floating-point filter for the line intersection algorithm. In *Geographic Information Science*, vol. 3234. 2004, pp. 94–105. 4

[Fra80] FRANKLIN W. R.: A linear time exact hidden surface algorithm. *SIGGRAPH Comput. Graph. 14*, 3 (July 1980), 117–123. 2

[Gal69] GALIMBERTI R.: An algorithm for hidden line elimination. *Commun. ACM 12*, 4 (1969), 206–211. 2

[GBAM11] GRIBEL C. J., BARRINGER R., AKENINE-MÖLLER T.: High-quality spatio-temporal rendering using semi-analytical visibility. *ACM Trans. Graph. 30*, 4 (2011), 54:1–54:12. 2

[GDAM10] GRIBEL C. J., DOGGETT M., AKENINE-MÖLLER T.: Analytical motion blur rasterization with compression. HPG '10, pp. 163–172. 2

[GS98] GUPTA N., SEN S.: An improved output-size sensitive parallel algorithm for hidden-surface removal for terrains. In *IPPS/SPDP 1998* (1998), pp. 215 –219. 2

[GT96] GUENTER B., TUMBLIN J.: Quadrature prefiltering for high quality antialiasing. *ACM Trans. Graph. 15*, 4 (1996), 332–353. 2

[Hor82] HORNUNG C.: An approach to a calculation-minimized hidden line algorithm. *Computers & Graphics 6*, 3 (1982), 121 – 126. 2

[KPRG12] KELLER A., PREMOZE S., RAAB M., GRUENSCHLOSS L.: Advanced (quasi) monte carlo methods for image synthesis. In *ACM SIGGRAPH Courses* (2012). 2

[McC95] MCCOOL M. D.: Analytic antialiasing with prism splines. SIGGRAPH '95, pp. 429–436. 2

[McK87] MCKENNA M.: Worst-case optimal hidden-surface removal. *ACM Trans. Graph. 6*, 1 (1987), 19–28. 2

[MG11] MERRILL D., GRIMSHAW A.: High performance and scalable radix sorting. *Parallel Processing Letters 21*, 02 (2011), 245–272. 6

[MN88] MITCHELL D. P., NETRAVALI A. N.: Reconstruction filters in computer-graphics. SIGGRAPH '88, pp. 221–228. 2

[MS11] MANSON J., SCHAEFER S.: Wavelet rasterization. *Computer Graphics Forum 30*, 2 (2011), 395–404. 1, 2, 7

[Mul89] MULMULEY K.: An efficient algorithm for hidden surface removal. *SIGGRAPH Comput. Graph. 23*, 3 (1989), 379–388. 2

[NVI] NVIDIA: Cuda technology. `http://www.nvidia.com/cuda`. 6

[Pan11] PANTALEONI J.: Voxelpipe: A programmable pipeline for 3d voxelization. In *Proc. HPG 2011* (2011). 6

[RCDF08] RUSINKIEWICZ S., COLE F., DECARLO D., FINKELSTEIN A.: Line drawings from 3d models. In *ACM SIGGRAPH 2008 classes* (2008), pp. 39:1–39:356. 2

[RFK90] RANDOLPH FRANKLIN W., KANKANHALLI M. S.: Parallel object-space hidden surface removal. *SIGGRAPH Comput. Graph. 24*, 4 (1990), 87–94. 2

[RKLC*11] RAGAN-KELLEY J., LEHTINEN J., CHEN J., DOGGETT M., DURAND F.: Decoupled sampling for graphics pipelines. *ACM Trans. Graph. 30*, 3 (May 2011), 17:1–17:17. 1

[Rob63] ROBERTS L. G.: *Machine Perception of Three-Dimensional Solids*. 1963. 1

[RS88] REIF J. H., SEN S.: An efficient output-sensitive hidden surface removal algorithm and its parallelization. SCG '88, pp. 193–200. 2

[SO92] SHARIR M., OVERMARS M. H.: A simple output-sensitive algorithm for hidden surface removal. *ACM Trans. Graph. 11*, 1 (1992), 1–11. 2

[SSS73] SUTHERLAND I. E., SPROULL R. F., SCHUMACKER R. A.: Sorting and the hidden-surface problem. AFIPS '73, pp. 685–693. 1

[SSS74] SUTHERLAND I. E., SPROULL R. F., SCHUMACKER R. A.: A characterization of ten hidden-surface algorithms. *ACM Comput. Surv. 6*, 1 (1974), 1–55. 1

[Tur90] TURKOWSKI K.: Filters for common resampling tasks. In *Graphics gems*. 1990, pp. 147–165. 2

[WA77] WEILER K., ATHERTON P.: Hidden surface removal using polygon area sorting. *SIGGRAPH Comput. Graph. 11*, 2 (July 1977), 214–222. 2