High Performance Graphics (2016)
Ulf Assarsson and Warren Hunt (Editors)

SVGPU: Real Time 3D Rendering to Vector Graphics Formats

Apollo I. Ellis[†] and Warren Hunt[‡] and John C. Hart[†]

Abstract

We focus on the real-time realistic rendering of a 3-D scene to a 2-D vector image. There are several application domains which could benefit substantially from the compact and resolution independent intermediate format that vector graphics provides. In particular, cloud streaming services, which transmit large amounts of video data and notoriously suffer from low resolution and/or high latency. In addition, display resolutions are growing rapidly, exacerbating the issue. Raster images for large displays prove a significant bottleneck when being transported over communication networks. However the alternative of sending a full 3D scene worth of geometry is even more prohibitive. We implement a real time rendering pipeline that utilizes analytic visibility algorithms on the GPU to output a vector graphics representation of a 3D scene. Our system SVGPU (Scalable Vector on the GPU) is fast and efficient on modern hardware, and simple in design. As such we are making a much needed step towards enabling the benefits of vector graphics representations to be reaped by the real time community.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Hidden line/surface removal

1. Introduction

In the earliest days of computer graphics, the VRAM needed for a raster framebuffer was prohibitively expensive, and graphics was output in a vector format. Hidden line algorithms were needed to convert 3-D scene geometry into a planar map of view projected regions with depth a complexity of one, so their outlines could be displayed on the vector display devices available then. While today's platforms have ample VRAM, we nevertheless find many modern reasons to explore vector rendering of 3-D meshes into a planar map consisting only of the visible portions of its triangles.

We propose SVGPU, a GPU-optimized real-time vector image rendering system that renders a 3-D scene into a resolution independent vector image, a planar map consisting only of visible polygons. This intermediate-stage output with its proper visibility determination has several advantages over the typical final-stage raster image of visible pixels that relies on the z-buffer. Vector images provide a resolution independent representation that can be efficiently rasterized at any resolution onto an arbitrarily sized display, ranging from watches to videowalls to head-mounted displays, and as shown in Fig. 2 can include per-pixel texturing and shading. The rasterization of a planar map consists of only point-in-polygon tests (which modern GPU's can efficiently compute) and avoids the need to sort depth, and so does not suffer the pathological issues of depth buffering, c.f. [LJ99].

† University of Illinois at Urbana-Champaign
‡ Oculus Research

© 2016 The Author(s)
Eurographics Proceedings © 2016 The Eurographics Association.

Figure 1: *A toon shaded Stanford bunny of 70K triangles rendered as a resolution-independent planar map of visible triangle portions (depth complexity is no greater than one) in about 15 ms (67 Hz) by the SVGPU vector renderer, representing more than a four-fold improvement over previous GPU vector renderers.*

There are several specific modern computer graphics applications that would benefit from a modern real-time GPU version of a vector image renderer that generates a planar map of unit depth complexity triangles.

Some modern GPU's, in particular the PowerVR GPU's found in mobile devices, utilize tile-based deferred rendering (TBDR) [Lau10, Sol15]. The TBDR pipeline decomposes primitives after the per-vertex transform-and-lighting stages of the graphics pipeline into screen tile elements. On a per-tile bases, an early visibility test becomes feasible either through primitive sorting or ray casting, to eliminate unnecessary texture fetches and fragment shading calls for occluded fragments. Our SVGPU approach employs Robert's algorithm [Rob63] and silhouette clipping efficiently on a per-tile basis to offer an alternative approach for the early visibility test in TBDR to reduce rasterization pipeline work.

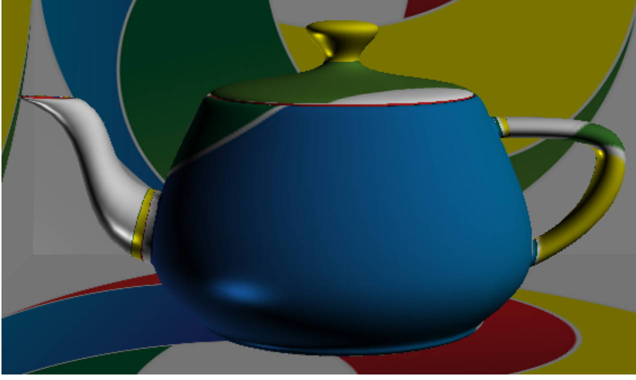


Figure 2: *A vector image of an environment-mapped Utah teapot. The SVGPU renderer outputs a resolution-independent vector image as a planar map consisting only of visible screen triangles. The vertices of these screen triangles include properly interpolated attribute data including texture coordinates, such that a rasterizer can texture and shade its primitives at whatever raster resolution is desired, even variable resolutions for a foveated display.*

In an era where network bandwidth is a critically valuable commodity, vector images are compact, reducing both network consumption and latency. Cloud gaming is an emerging trend of the video gaming industry, where the display image of a video game is rendered by a server and transmitted over the internet to the client. Current gaming-as-a-service (GaaS) systems render raster images that are transmitted as MPEG streams, but these streams consist of full resolution I-frames because the computation of block motion on these raw images needed for more efficient MPEG transmission creates too much latency and would require prediction since the streams are not static. In fact a server rendering 3-D game scenes directly to a planar map could also yield correspondences that would better support motion for more efficient game video transmission.

With the advent of lower power mobile VR, such as Samsung Gear VR and Google Cardboard, VR is becoming a more available, mainstream technology. However, these low power devices lack the capability to render complex scenes with lots of geometry, and benefit from the same advantages of SVGPU as do other cloud gaming clients. More advanced VR head-mounted displays can utilize eye tracking [Mas15] to support foveated rendering [GFD*12] which renders the portion of the screen an observer is looking at, at a significantly higher resolution than the remainder of the screen. The vector image output by the SVGPU renderer is resolution independent such that a variable-resolution rasterizer could then scan convert its foveated primitives (or tiles) at a higher resolution than its peripheral primitives/tiles.

In this paper we present a real-time triangle-based vector renderer that converts a 3-D meshed scene into a planar map of 2-D triangles. This result can be directly converted into a vector graphics representation (e.g. SVG). Or it can shipped across a network interface and quickly rasterized on a client system without need for a depth buffer and with display-resolution-dependence.

Our pipeline consists of five stages described in Section 3. The first stage performs transform, clipping, and binning operations, which is borrowed directly from the rasterization pipeline, and we make no noteworthy additions to these operations. The next three stages form the main contribution of the analytic visibility pipeline. The second stage, described in Section 3.1 performs silhouette extraction using GPU spatial hashing to quickly find neighboring frontfacing-backfacing triangle pairs with a linear sweep through the mesh. The third stage described in Section 3.2 clips triangles to the extracted silhouette edges, limited to the geometry in each bin, using GPU dynamic parallelism to better balance load across bins representing different amounts of geometry. This silhouette clipping of triangles allows the fourth stage described in Section 3.3 to simply cull triangles if any occlusion is detected, which is a quadratic-time comparison between all-pairs of triangles in a bin. The fifth stage of our system outputs the result, either as a vector representation or the planar map of triangular regions.

The design of our vector renderer is based on the idea that we use the same spatial coherence and streamed processing tricks as those developed for fast rasterization graphics pipelines, replacing the rasterization phase with anaylitic visibility. We evaluate performance in Section 4.

2. Previous Work

The hidden line problem was well studied decades ago [SSS74]. Most modern approaches have been based on Appel's algorithm [App67] which extracts continuous silhouette components to display, computing the quantitative invisibility as these components cross each other. The main benefit of Appel's algorithm is that once the silhouette is extracted (after a linear pass through the scene polygons), the polygons no longer need to be processed, and comparisons only need to occur along the silhouette edges, significantly reducing computation. The main drawback is that the mesh silhouette is plagued with special cases, including cusps, switchbacks and non-transverse intersections that can affect robustness. Our approach clips triangles instead of the silhouette to the silhouette edges, and does not require the silhouette edges to be connected into a continuous curve for fragile incremental visibility computation.

Robert's algorithm is an even older approach that simply compares all pairs of scene polygons, clipping and culling occluded portions of polygons [Rob63]. These all-pairs comparisons were slightly reduced using bounding boxes, but still resulted in a quadratic time complexity, but also a significantly more robust output than Appel's algorithm. Our SVGPU approach leverages this robustness, and further reduces the impact of quadratic all-pairs triangle occlusion comparisons through binning and clipping only against silhouette edges. It also maps better to the brute force streaming parallelism offered by modern GPU's than does Appel's algorithm.

A large number of non-photorealistic rendering systems have included renderers that convert a 3-D scene into 2-D planar map [WS94, HZ00, Geu03, SEH08, EWHS08, EPD09, KH11], but these have largely been offline CPU programs that focused on the quality of the output. Some have looked at the real-time non-photorealistic rendering (NPR), e.g. by fast (sublinear) statistical global searches

© 2016 The Author(s)
Eurographics Proceedings © 2016 The Eurographics Association.

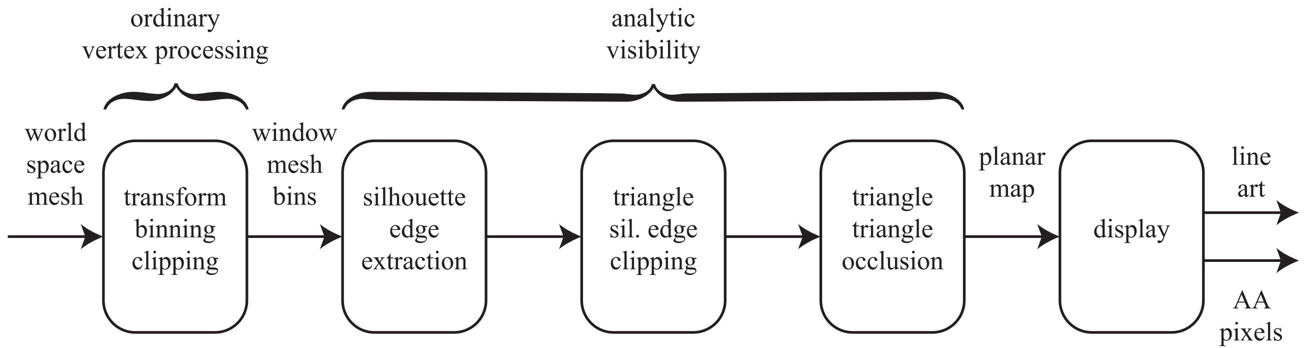


Figure 3: *The stages of our binned vector graphics rendering system.*

for seed segments of the silhouette [MKG*97], instead of our linear-time spatial hash approach to silhouette extraction. A variety of other techniques have also been employed to accelerate NPR rendering based on actual silhouette edges [BS00, KB05] or their approximation [RC99]. Some have also developed hardware solutions for real-time NPR contour extraction [Ras01, HK04, WSC*05].

The GPU has been used to compute the high-quality visibility of stylized lines by using a texture atlas as an intermediate frame buffer for compositing [CF09], but the actual visibility is based here on the depth buffer. The GPU was also used for analytic visibility of polygons, using an edge-based approach [AWJ13], whereas our approach is triangle based, using the silhouette edges only for clipping to yield better performance results.

3. The SVGPU Pipeline

The input to our pipeline is a 3-D scene of triangles. While our implementation uses an indexed face set representation, we do not require any particular organization, and our approach will work well with triangle soup. We do not require the meshes to be manifold, but any non-manifold or boundary edges will be classified as silhouette edges which are more expensive than manifold edges. In order to streamline our clipping and occlusion processes, we do require non-penetrating geometry, such that the only intersections between two triangles can occur along a shared edge, so scenes such as the Utah teapot would require re-tessellation.

The first stage of our pipeline performs the ordinary vertex processing pipeline found in common rasterization systems, such as OpenGL. This stage transforms world-space triangles into a perspective viewed "window" coordinate system. The triangles are then organized and rectangle-clipped into a 2-D grid of silhouette clipping bins organized across the window. As detailed in the following subsections, we extract the silhouette edges from the mesh, and clip the mesh triangles to these silhouette edges. Then an occlusion test can determine on a per-triangle basis, which triangles are visible, yielding a planar map containing only completely visible triangles. All stages are implemented as Cuda kernels using compute capability 5.2.

3.1. Silhouette Extraction

SVGPU detects "silhouette" edges with a spatial hash table, which can be efficiently constructed and accessed on the GPU [LH06]. (We use the term silhouette in this paper loosely, to refer to the visual contour of edges shared by both frontfacing and backfacing triangles.) We compute the hash index of each edge as a bit interleaved mixing of the sorted 3-D coordinates of the edge's two vertices. The triangle is inserted into the hash table at three places corresponding to the hash keys of its three edges.

Once all triangles have been entered into the hash table, the silhouette edges are extracted as entries whose corresponding face normals (in projected viewing coordinates) have oppositely signed z values. We also include as silhouettes any edges where the hash table lists any number of triangles besides two. Since this approach is based on vertex geometry and not mesh topology, it robustly handles triangle soup inputs so long as the shared vertices between neighboring triangles are sufficiently close enough to hash to the same table entry. Other hash collisions also occur, so during extraction each key bucket is traversed to produce only appropriate silhouette edge pairs. When each silhouette edge is identified, the edge is then binned at the same resolution as the clipping bins used for geometry, but in a separate set of bins. This kernel uses one thread per bucket.

3.2. Silhouette Clipping

The silhouette clipping stage clips every triangle to every silhouette edge in the current silhouette clipping bin. Since we expect the number of triangles that overlap each silhouette edge to be small, we segment this stage into two steps to retain GPU instruction coherence: a culling step (based on a trivial reject test) and a clipping step that performs the actual clipping operation. In the following subsections we will refer to variables and functions in our pseudocode, and we will italicize their names.

3.2.1. Silhouette Clipping: Culling and Setup

The culling phase is used to remove most of the non-overlapping triangle-edge pairings from consideration for clipping.

© 2016 The Author(s)
Eurographics Proceedings © 2016 The Eurographics Association.

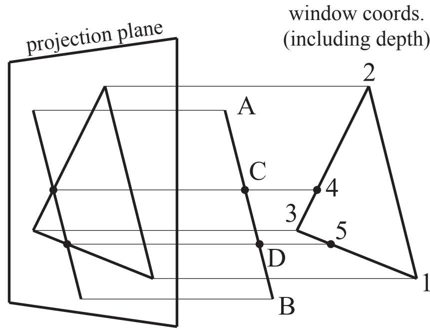


Figure 4: *Geometric configuration for clipping* $\Delta 123$ *to silhouette edge* $\overline{AB}$.

Let $(x_1,y_1,z_1),(x_2,y_2,z_2)$ and (x_3,y_3,z_3) be the window coordinates of a triangle $\Delta 123$, and let (x_A,y_A,z_A) and (x_B,y_B,z_B) be the window coordinates of the silhouette edge $\overline{AB}$, as shown in Figure 4. Then this culling step consist of three tests.

1. If $(x_1,y_1),(x_2,y_2)$ and (x_3,y_3) lie on one side of the line passing through (x_A,y_A) and (x_B,y_B), then cull.
2. If (x_A,y_A) and (x_B,y_B) lie on the outside of a line passing through any combination of $(x_1,y_1),(x_2,y_2)$ and (x_3,y_3), then cull.
3. Let C and D be the points on the line passing through A and B, and let 4 and 5 be the points on the edges of $\Delta 123$, such that $(x_C,y_C) = (x_4,y_4)$ and $(x_D,y_D) = (x_5,y_5)$ indicate where the window projection of the silhouette edge crosses the window projection of the triangle. If z_C and z_D are behind (less than) z_4 and z_5, then cull.

We also cull if (4) the silhouette edge is an edge of the triangle, (5) if the triangle is back facing, and (6) if the triangle is obviously in front of the silhouette edge: $\min z_1,z_2,z_3 > \max z_A,z_B$.

The culling segment is implemented as an $m \times n$ kernel that considers the coverage of m triangles by n silhouette edges, in each bin. We use dynamic parallelism to retain GPU utilization, see Section 3.2.3. We launch a single kernel from the CPU, with *BinCount* threads, where each thread retrieves m and n (*TriCount* and *EdgeCount*) for its bin, and launches the culling kernel as a child kernel. Each top level thread also launches the follow up kernels to create the data structures need for clipping. In fact these top level threads are additionally responsible for launching the clipping kernels described later.

In the culling kernel, $TrivialReject$, each thread considers whether one specific triangle is covered by one specific silhouette edge. If the thread survives all culling tests, it outputs a candidate pair consisting of the triangle id and the silhouette edge id. We use atomics to increment a per triangle edge counter for each pair ($EdgeCounts$), and to build a list of triangle ids that are going to be clipped, $TriangleList$.

The clipper requires an adjacency list ($EdgeList$) that, for each triangle id, holds a list of silhouette edge ids. These edge ids index into the silhouette bins created during silhouette extraction. To allocate and populate $EdgeList$, two kernels are launched directly following the culling kernel. The allocation kernel, $AllocateAdjacency$, runs first and reserves row space for each triangle. With $CandidateTriangles$ # threads, the kernel indexes into $TriangleList$ by thread id to retrieve the triangle id which each thread uses to read the triangle's edge count from $EdgeCounts$. Each thread then adds the edge count to an atomic counter, and stores the old count to a buffer. This old count will be used later as the row offset to a triangle's edges in $EdgeList$. The second kernel populates the $EdgeList$. Using $CandidatePairs$ # threads, we read the candidate pair buffer and use each pair's triangle id to index into the buffer of row offsets we just created in the previous kernel. We then store the pair's edge id in $EdgeList$ at the row offset plus a count which is incremented atomically with each edge.

Clipper also requires a data structure that holds position data for all polygons being clipped at any time. Luckily we can initialize this *PolygonData* structure in the culling kernel. We must only write the triangle's positions once, not for every triangle-edge pair. We check if the per triangle edge counter in $EdgeCounts$ was zero before it's increment, and if so, we use this thread to write the triangle into *PolygonData*. We index into $EdgeCounts$ using the triangle id i.e. the index of the triangle in its bin. Naturally a single per bin atomic is used to keep track of the write offset for a triangle into the polygon buffer.

3.2.2. Silhouette Clipping: Clipping

Clipping proceeds in rounds operating off of the $EdgeList$, the *PolygonData* buffer, and some indexing structures described below. Each round, each thread will clip one triangle by one of its corresponding silhouette edge candidates, so the thread count in each kernel launch is the count of participating triangles, $ActivePolygons$. After a single triangle is clipped, a polygon may be produced. As such we will refer to polygons as opposed to triangles from this point forward. Our clipping algorithm follows Bernstein's work with fast exact booleans [BF09].

The reason we use this clipper is to maintain a single level of clipping error throughout the pipeline. We store a list of edges for each polygon and always regenerate clip intersections from these original input edges. We do store the intersection points temporarily for evaluating point-on-edge-side predicates. Otherwise it would be necessary to re-derive the exact same point every time a polygon is considered, so this saves some computation.

A clip occurs as follows. We iterate through each neighboring triple of vertices on the polygon starting with the last vertex, first vertex, and second vertex in the polygon's vertex list. Bernstein showed this is necessary and sufficient to sort out all ambiguous clipping cases. The points are categorized as "on," "in," or "out" of the clipping edge using a simple point-edge-side predicate. The algorithm uses a lookup table to decide at each step whether to output the current edge, or to generate a new vertex via edge-edge intersection, and output the associated clip edge with it. Our lookup table differs slightly from Bernstein's, in that we use the convention of storing a vertex with the edge leaving the vertex. The LUT derivation however is quite simple following the boolean work, and we omit it here.

© 2016 The Author(s)
Eurographics Proceedings © 2016 The Eurographics Association.

```
function ClippingKernel(ClipStructures)
    //ClipStructures here is meant here to contain
    //all structures used in the clipper code :
    //PolygonData, PolygonInfo, EdgeInfo,
    //EdgeList, TriangleList, and various counters
    //By Bins we refer to the tri bins and edge bins
    T=TriCount * EdgeCount
    TrivialReject <<< T >>> (Bins,ClipStructures)
    //same value as ClipStructures.ActivePolygons
    T=ClipStructures.CandidateTriangles
    AllocateAdjacency <<< T >>> (ClipStructures)
    T=ClipStructures.CandidatePairs
    BuildAdjacency <<< T >>> (ClipStructures)
    Round=0
    while ClipStructures.ActivePolygons do
        T=ClipStructures.ActivePolygons
        Clip <<< T >>> (Bins,ClipStructures,Round)
        ClipStructures.swapBuffers
        Round++
    T=ClipStructures.TessCount
    Tessellate <<< T >>> (Bins,PolygonData)
ClippingKernel <<< BinCount >>> (Bins...)
function TrivialReject(Bins,ClipStructures)
    for all Edges and Triangles do
        Tri = ThreadId/SilhouetteCount
        Edge = ThreadId%SilhouetteCount
        if !CullingTests(Tri,Edge) then
            CLoc = INC(CandidatePairs)
            Candidates[CLoc]=(Edge.Id,Tri.Id)
            EdgeCount = INC(EdgeCounts[Tri.Id])
            if EdgeCount == 0 then
                PLoc = INC(PolygonCount)
                PolygonData[PLoc]=(Vertices,Edges)
function Clip(Bins,ClipStructures,Round)
    for all Polygons do
        PInfo=PolygonInfo[ThreadId]
        EInfo=EdgeInfo[PInfo.TriId]
        ClipEdge=EdgeList[EInfo.row_offset + Round]
        PData=PolygonData[PInfo.offset]
        NextPosition=ADD(PositionCounter,PInfo.Sides)
        RecheckTrivialReject(PData,CEdge)
        for I in PInfo.Sides do
            (Last,Curr,Next)=PData.GetVertices(I)
            (p0,p1,p2)=Predicate(Last,Curr,Next,Edge)
            COND=LUT(p0,p1,p2))
            if COND... then
                VertexOut=Curr.Vertex
            if COND... then
                VertexOut=Isect(Curr.Edge,ClipEdge)
            if COND... then
                EdgeOut=Curr.Edge
            if COND... then
                EdgeOut=ClipEdge
            OutputVertex(EdgeOut,VertexOut)
        OutputInfo(NextPosition,...)
```

Aside from the polygon position buffer *PolygonData*, there are two more structures used to keep track of information during clipping. For each active polygon being clipped, the *PolygonInfo* buffer holds the triangle id, side count, and an offset into *PolygonData* where its vertices live. For every original polygon, the *EdgeInfo* buffer holds the row offset into *EdgeList*, and an edge count. Since we launch one thread per active polygon during clipping rounds, we index into *PolygonInfo* by thread id. We then use its triangle id member to index into *EdgeInfo*. The row offset member of *EdgeInfo* is used to retrieve this actual clipping edge by indexing into *EdgeList* at the row offset with the current round number added to it. Finally we read *PolygonData* at the offset stored in *PolygonInfo* and proceed to clip. The *PolygonData* structure includes edges, vertices, and barycentric coordinates. Note that barycentric coordinates are clipped along with the polygon's edges in order to provide interpolation for the original triangle's vertex attributes.

Once all clipping rounds have occurred for all triangles and their respective child polygons, the clipping phase retires and we tessellate the polygons back into triangles with a final kernel. Tessellation is straightforward since all the polygons are convex. We launch one tessellation thread per polygon.

It is important to note a few things about the clipper. First, a single clip must retain both sides of the clipped polygon, because the portion of an edge clipped away by one silhouette edge may be reintroduced by a subsequent silhouette edge. Hence we run the clipping step twice, reversing the predicates the second time to obtain the polygonal region clipped by the first clipping run. The reason we use two separate runs is to improve thread coherence, since not all threads will clip, they do not need to participate in the second pass. Second, one should recheck trivial rejection of candidate silhouette edges against clipped (child) polygons. This mitigates the case in which there is a large polygon, behind a complex object, and it is repeatedly clipped by all edges in the complex object, when in fact those edges do not overlap the polygon. This must be done because the clipper itself is otherwise oblivious to whether or not an edge actually overlaps a polygon, since the edge equation extends to infinity. The trivial reject test can be implemented in the prologue of the clipping kernel or in a separate kernel. We experimented with both approaches and ended up with the prologue approach to reduce complexity.

3.2.3. Dynamic Parallelism

Dynamic Parallelism, available in Cuda on compute capability 3.5+ GPUs, is a natural fit to this kind of problem. It allows a kernel to launch another kernel. With clipping we have work items that can produce more work items in non uniform distributions, i.e. we have dynamically changing amounts of parallelism. So we need to be able to redistribute work between compute resources to keep the GPU busy. Otherwise load balance is lost.

There is more than one way to achieve this, but a simple solution is to restart the kernel every time a generation of work items has finished computing the next generation of work items. This however, causes a CPU synchronization bottleneck, and was the main motivation for turning to dynamic parallelism. With kernels that

© 2016 The Author(s)
Eurographics Proceedings © 2016 The Eurographics Association.

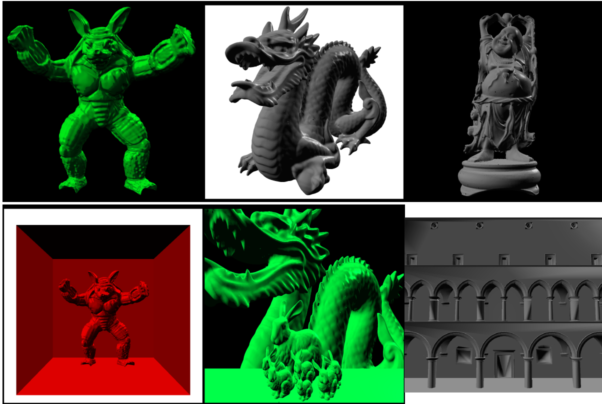


Figure 5: *Benchmark scenes (l-r, t-b: Armadillo, Dragon, Buddha, Armadillo Box, Dragon Bunnies, Sponza) with various shaders applied using interpolated view space positions and normals, in addition to the Bunny and Teapot.*

launch kernels we can avoid the CPU bottleneck and adjust to the changing workload as needed.

Dynamic parallelism is also a natural fit because it encapsulates the binned structure of our algorithm. Running a single kernel to compute different numbers of work items for different bins can be complicated to manage, likely requiring prefix sums to compute bin delimitations. We avoid this altogether with dynamic parallelism.

3.3. Triangle Occlusion

After clipping we are left with possibly overlapping but non-crossing triangles. No triangle is partially occluded, so if any part of a triangle is occluded, then it is completely occluded. The triangle occlusion step removes any occluded triangles, leaving a planar map of depth complexity one consisting only of the visible triangles of the scene.

We first re-bin the triangles output from silhouette clipping at a finer bin resolution to reduce occlusion's all-to-all time complexity, as detailed in Section 4.1. This step is straightforward and fast. We then run the triangle-to-triangle occlusion kernel, one thread per pair, in each of these occlusion bins.

Since each triangle is either fully occluded or fully visible, a simple centroid test suffices to determine visibility. We calculate the centroid of the potentially occluded triangle, and derive the barycentric coordinates of that point on the potentially occluding triangle. When we have the barycentric coordinates, we can interpolate the z value at the occluder's vertices and test it against the centroid z value. We discard any triangle whose centroid is overlapped by any other triangle. We only test the original triangles as occluders. This reduces the occlusion test to fewer triangles and is still valid since the original triangles are a superset of the many triangles that have been refined by silhouette clipping.

Stage	Bun.	Arm.	Drag.	Bud.	Box	D+B	Sza.	Tea.
Sil. Hash	1.2	3.8	24	35	3	66	.4	.19
Sil. Clip	12	30	42	64	175	249	177	22
Occlusion	2.3	18	38	78	39	179	8	2
Total	15.5	51.8	105	205	217	527	185.4	24.19

Table 1: *Profile of SVGPU run time performance (ms) per stage, using 1,024 silhouette clipping bins.*

4. Results

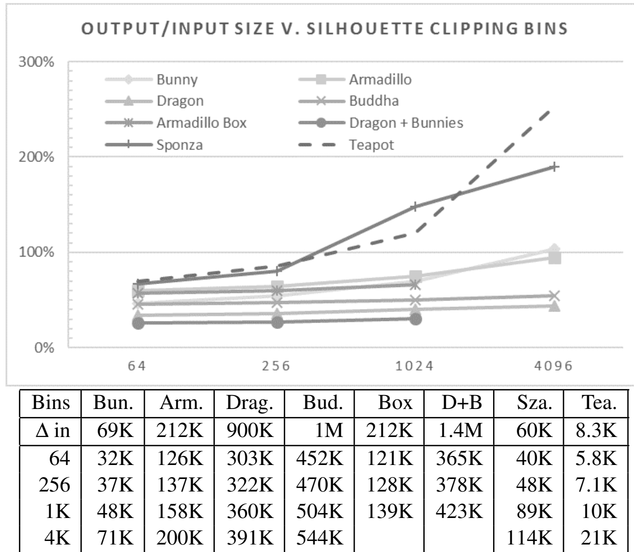
Our prototype implementation is demonstrated on a variety of well known models, specifically the bunny, teapot, armadillo, dragon and buddha, as well as on some larger scenes constructed to exhibit pathological cases: armadillo in the Cornell box, a dragon behind several bunnies, and the Sponza, as shown in Figure 5. The armadillo in the Cornell box exhibits a situation in which many tiny silhouette edges are candidates for clipping a few large polygons in the background. The dragon with bunnies on the other hand showcases a scenario in which a very large number of silhouettes are clipping against many small triangles. Each of these models was represented as an indexed face set, and for these examples our silhouette extraction used a hash on the vertex indices instead of the vertex coordinates.

Table 1 reports our profile performance measurements for the various stages of the SVGPU rendering process. Our measurements of silhouette hashing show it ranging from less than 1% of the total run time to almost 20%. While this run time is tied to the silhouette count, our experiments show it is also largely influenced by the number of hash collisions. Collisions affect the run time of a single thread's bucket traversal, so some threads will run long and diverge from the other threads in their warp, causing load imbalance. As expected, our profile performance measurements of the silhouette clipping process was greatly influenced by the number of silhouette clipping bins, and clipping was the major contributor to the bin variance shown later in the performance charts. Our profile of occlusion showed it to vary similarly to the silhouette hashing performance, suggesting that the same features that create silhouettes are also creating additional occlusions. Overall, clipping used about two-thirds of the time, occlusion about one-third, and silhouette hashing was either negligible or at most about one-fifth of the run time.

Fig. 6 shows the number of output triangles that SVGPU generates in the output planar map is quite low for individual models but grows for scenes. As the silhouette clipping bin resolution increases the output triangle count grows. This is attributed to clipping to bin edges, but utilizing more, smaller bins to help increase load balance and reduce the number of all-pairs silhouette-triangle clipping cases. We clip to bin edges to maintain correctness during the clipping stage, and more bins generate more clipping on bin borders, which produces more triangles. The armadillo-in-the-box scene produces fewer polygons than what might be expected from the excessive clipping in that scene. The largest growths comes from the high depth complexity of Sponza and the low initial polygon count of the Utah teapot.

Fig. 7 shows that the number of silhouette clipping bins affects

© 2016 The Author(s)
Eurographics Proceedings © 2016 The Eurographics Association.



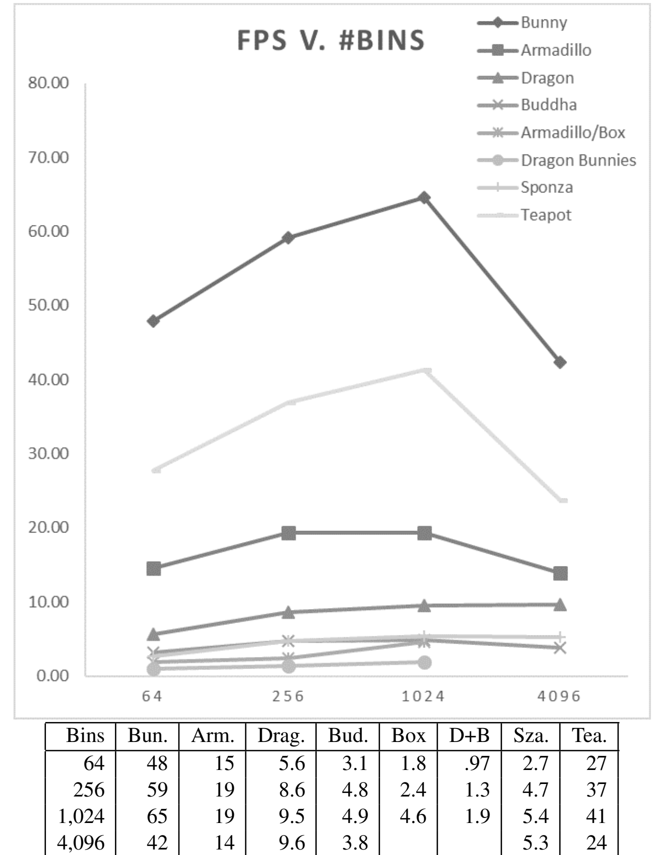
Bins	Bun.	Arm.	Drag.	Bud.	Box	D+B	Sza.	Tea.
Δ in	69K	212K	900K	1M	212K	1.4M	60K	8.3K
64	32K	126K	303K	452K	121K	365K	40K	5.8K
256	37K	137K	322K	470K	128K	378K	48K	7.1K
1K	48K	158K	360K	504K	139K	423K	89K	10K
4K	71K	200K	391K	544K			114K	21K

Figure 6: *Output size growth measured as the number of triangles output v. input, for different choices of number of silhouette clipping bins. (Dragon and Buddha examples reported for 4,096 bins actually only used 2,500 bins.)*

the performance. Larger numbers of smaller bins helps the clipping phase to better balance its load, subdividing the scene more aggressively to avoid teapot-in-a-stadium situations. However, at some point, in most cases separating the screen into 4,096 bins, the increase in the number of bins begins to have negative affects. The increases triangle clipping to the smaller rectangular boundaries of the more plentiful bins begins to affect both clipping time and occlusion time by generating more work for the clipper and more triangles for occlusion. The overall optimum appears to be at $1,024 = 32 \times 32$ silhouette clipping bins.

Fig. 8 reveals the SVGPU triangle rate, ranging from a peak of 8.65M triangles per second for the 900K element mesh of the dragon down to about 200K triangles per second for the teapot, whose meager 8.3K element mesh does not generate enough parallelism in SVGPU's thread configuration. Most of the models (bunny, armadillo, buddha) achieve a typical 4M Δ/s triangle rate. It is interesting to note that the bunny and Sponza are both similarly sized in the 60-70K range, but Sponza's triangle rate is significantly lower, likely from its depth complexity and the resulting increased impact in the per-bin all-pairs steps of clipping and occlusion.

Table 2 reveals the size of the various data structures and buffers used throughout the pipeline. It shows the max number of elements of a particular type that were in flight during run time and their averages over all non empty bins. Bin populations are also listed along with total output triangles. These numbers reflect the items not bytes. The scaling of storage requirements with bin size can be observed moving across each row. The trends behave as we would expect with all values shrinking with the increase in bin resolution. The main problem exhibited here is scaling. Bin resolutions are growing by powers of two, and what we would want to see is that the item counts also move down in powers of two (or more).



Bins	Bun.	Arm.	Drag.	Bud.	Box	D+B	Sza.	Tea.
64	48	15	5.6	3.1	1.8	.97	2.7	27
256	59	19	8.6	4.8	2.4	1.3	4.7	37
1,024	65	19	9.5	4.9	4.6	1.9	5.4	41
4,096	42	14	9.6	3.8			5.3	24

Figure 7: *Performance measured in frames per second (Hz.), for different choices of number of silhouette clipping bins. (Dragon and Buddha examples reported for 4,096 bins actually only used 2,500 bins.)*

This would keep the system stable/linear in terms of memory consumption. This is not the case however, the item counts are moving down just about linearly, so the increase in bin resolution is costly in terms of storage requirements.

4.1. Occlusion Binning

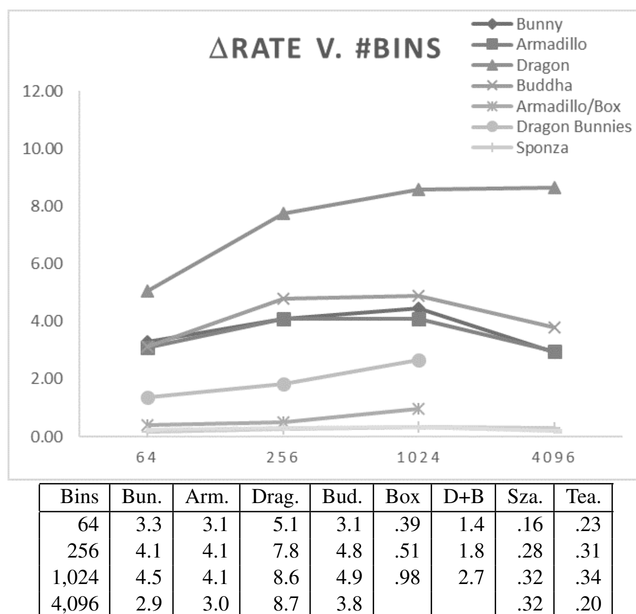
For the occlusion stage, as discussed earlier, we used more, smaller bins than we do in the silhouette clipping stage. Smaller silhouette clipping bin sizes reduce the number of triangles for that stage's all-pairs quadratic comparison of triangles to silhouette edges, but setting them too fine (as was the case of 4,096 silhouette clipping bins) requires too much bin rectangle clipping and outputs too much geometry to the occlusion stage. Thus we use a separate finer bin sizing for the occlusion stage.

In our examples, we used 4,096 bins for all of the models except the dragon and the Buddha, regardless of the number of silhouette clipping bins (64, 265, 1,024 or 4,096). Due to the heavy feature-driven occlusion of the Buddha and dragon models, we used 16,384 bins for their occlusion computation. These models eventually overflowed our available memory, when using 4096 sil-

© 2016 The Author(s)
Eurographics Proceedings © 2016 The Eurographics Association.

Bins	64		256		1,204		4,096		64		256		1,204		4,096	
	Ave.	Max	Ave.	Max	Ave.	Max	Ave.	Max	Ave.	Max	Ave.	Max	Ave.	Max	Ave.	Max
Model (tris.)	Bunny (69K)								Armadillo (212K)							
Clipping Bin Occupancy	1K	8K	497	6K	191	3K	75	2K	6K	26K	2K	12K	729	7K	243	2K
Occlusion Bin Occupancy	31	3K	43	3K	58	3K	75	2K	110	3K	154	3K	204	3K	243	2K
Candidate Pairs	333	2K	153	2K	76	1K	39	1K	2K	9K	705	4K	284	3K	113	2K
Clip Polygon Count	214	1K	108	1K	54	854	26	713	685	3K	322	2K	149	1K	66	585
Clip Vertex Buffer Size	981	7K	470	7K	228	4K	109	3K	4K	17K	3K	9K	1K	7K	723	4K
Model (tris.)	Dragon (900K)								Buddha (1M)							
Clipping Bin Occupancy	7K	31K	4K	13K	1K	8K	798	5K	23K	49K	8K	19K	3K	10K	1K	6K
Occlusion Bin Occupancy	215	3K	284	3K	356	3K	490	2K	118	2K	155	2K	190	2K	338	3K
Candidate Pairs	3K	10K	1K	5K	392	4K	253	2K	5K	13K	2K	8K	696	7K	378	4K
Clip Polygon Count	1K	4K	552	2K	243	2K	163	798	3K	5K	1K	3K	413	2K	240	1K
Clip Vertex Buffer Size	7K	18K	3K	9K	1K	7K	723	4K	13K	28K	5K	15K	2K	9K	1K	7K

Table 2: *The size of the various structure buffers used during clipping along with bin sizes and output triangle counts. The memory consumption of each data structure both at it's max across bins and on average can be approximated from these counts.*



Bins	Bun.	Arm.	Drag.	Bud.	Box	D+B	Sza.	Tea.
64	3.3	3.1	5.1	3.1	.39	1.4	.16	.23
256	4.1	4.1	7.8	4.8	.51	1.8	.28	.31
1,024	4.5	4.1	8.6	4.9	.98	2.7	.32	.34
4,096	2.9	3.0	8.7	3.8			.32	.20

Figure 8: *The triangle rate, measured in million triangles per second, for different choices of number of silhouette clipping bins. (Dragon and Buddha examples reported for 4,096 bins actually only used 2,500 bins.)*

houette clipping bins and 16384 occlusion bins, and so we were only able to generate them with a maximum of 2,500 silhouette clipping bins and 10,000 occlusion bins.

4.2. Comparison with Previous Work

Previous results from analytical visibility on the GPU [AWJ13] render the 70K-triangle bunny at a variety of raster resolutions in a time ranging from 99 to 128 ms (visibility only), on an NVidia GeForce GTX 680, which has 1,536 cores running at 1GHz. Our results were computed on an NVidia GeForce GTX 980, which has 2,048 cores running at 1.1GHz. Comparing results from different GPUs is a complex process, but we can estimate that since GTX 980 represents 33% more cores running 10% faster, we should see about a 47% improvement in speed over the 680 used for analytical visibility's results (ignoring many other differences, including e.g. memory bandwidth).

This is a trivial comparison but nonetheless one of the only apples to apples comparisons available. SVGPU renders the bunny into a planar map in about 15ms (visibility only), whereas the analytic system running 47% faster would compute visibility for the bunny in about 70 ms, leading us to believe SVGPU is about 4.5 times faster. However SVGPU scales well as is demonstrated by the fact that it finishes the visibility computations of the armadillo roughly two times faster than the analytic system can compute the bunny, and further the close to one million polygon dragon is only 30 percent slower than their bunny.

4.3. Failure Cases

There are several shortcomings in our prototype implementation that should be addressed in future work. The primary issues are that memory usage is high and GPU utilization is fairly low.

At higher bin resolutions the system requires a significant amount of memory and this inhibits the rendering of our more complex scenes, e.g. armadillo in the Cornell box and the dragon behind the bunnies. We could not load the required buffers onto the GPU at a 64×64 bin resolution. Further, we are not using our memory budget effectively in teapot in a stadium scenarios. We cannot render scenes like Fairy Forest, because some bins generate a huge amount of clipping and our current strategy of allocating memory uniformly doesn't handle this case well. A solution to this memory budget issue could be to use adaptive binning structures such as quad trees, as well as more adaptive memory allocation strategies.

Another drawback associated with memory usage is the forced tweaking of the size parameters for various memory buffers. It is a cumbersome and manual process. We would like to both automatically size regions based on bin resolution, and further cut down on the flatness of our buffer layouts. Again, essentially we need more adaptive memory allocation. In the best case we could pack

© 2016 The Author(s)
Eurographics Proceedings © 2016 The Eurographics Association.

everything, as opposed to using pre-defined offsets between bin allowances as we currently do.

We do however feel there is no reason this algorithm and implementation cannot be heavily optimized in future work to support a much wider variety of scene structure.

The kernels in our system have many execution dependencies on memory. This causes GPU threads to idle often waiting on requests. While GPU occupancy is reasonable around 50% to 75%, instruction issue efficiency is lower, around 20% to 25%. The kernels for clipping and occlusion spend a lot of time in setup reading indices and offsets into various buffers. The current implementation essentially suffers from excessive indirection, which would be addressed via more strategic GPU streaming techniques.

5. Conclusion

We have demonstrated that the GPU can implement a vector rendering system which, with some additional work, could be suitable for small scale client server 3D content streaming applications and some VR setups. By binning geometry into small screen tiles, about $1/32^2$ of the screen size, we achieve an optimal domain decomposition that distributes a parallel clipping workload evenly while limiting the impact of an all-pairs quadratic triangle occlusion test. The result yields about a $4.5\times$ improvement over the state of the art.

We have analysed several performance factors which may be useful in future implementations. In particular we have sought to understand the impact of bin resolution on performance, worst case memory budget requirements, and optimization such a rebinning for increased performance.

While the system is far from complete and has it's share of drawbacks, we believe most of these to be implementation specific, not algorithmic. The core algorithm will likely prove useful in future analytic visibility efforts, whether they be in hardware or in software. In fact we expect a hardware implementation to be feasible, and potentially, required for real world rendering workloads.

Acknowledgments

Thanks to Sean Keely at the University of Texas Austin for invaluable advice on efficient GPU programming methods and fast math operations for visibility. Further thanks to Eric Huber and Ashwin Kumar Vijay at the University of Illinois Urbana-Champaign for their ideas about vector rendering, and for their coding contributions towards a vectoring rendering tool suite.

References

[App67] APPEL A.: The notion of quantitative invisibility and the machine rendering of solids. *Proc. 22nd ACM Natl. Conf.* (1967), 387–393. 2

[AWJ13] AUZINGER T., WIMMER M., JESCHKE S.: Analytic visibility on the gpu. *Computer Graphics Forum (Proc. Eurographics) 32*, 2 (May 2013), 409–418. 3, 8

[BF09] BERNSTEIN G., FUSSELL D.: Fast, exact, linear booleans. In *Proceedings of the Symposium on Geometry Processing* (Aire-la-Ville, Switzerland, Switzerland, 2009), SGP '09, Eurographics Association, pp. 1269–1278. 4

[BS00] BUCHANAN J. W., SOUSA M. C.: The edge buffer: A data structure for easy silhouette rendering. *Proc. NPAR* (2000), 39–42. 3

[CF09] COLE F., FINKELSTEIN A.: Fast high-quality line visibility. *Proc. I3D* (2009), 115–120. 3

[EPD09] EISEMANN E., PARIS S., DURAND F.: A visibility algorithm for converting 3d meshes into editable 2d vector graphics. *ACM TOG 28*, 3 (2009), 83:1–83:8. 2

[EWHS08] EISEMANN E., WINNEMÖLLER H., HART J. C., SALESIN D.: Stylized vector art from 3d models with region support. *Proc. EGSR* (2008), 1199–1207. 2

[Geu03] GEUZAINE C.: GL2PS: an OpenGL to PostScript printing library. www.geuz.org/gl2ps, 2003. 2

[GFD*12] GUENTER B., FINCH M., DRUCKER S., TAN D., SNYDER J.: Foveated 3D graphics. *Proc. SIGGRAPH Asia, ACM TOG 31*, 6 (2012). 2

[HK04] HO S. N., KOMIYA R.: Real time loose and sketchy rendering in hardware. *Proc. Spring Conference on Computer Graphics* (2004), 83–88. 3

[HZ00] HERTZMANN A., ZORIN D.: Illustrating smooth surfaces. *Proc. SIGGRAPH* (2000), 517–526. 2

[KB05] KIM K.-J., BAEK N.: Fast extraction of polyhedral model silhouettes from moving viewpoint on curved trajectory. *Comput. Graph. 29*, 3 (2005), 393–402. 3

[KH11] KARSCH K., HART J. C.: Snaxels on a plane. *Proc. NPAR* (2011), 35–42. 2

[Lau10] LAURITZEN A.: Deferred rendering for current and future rendering pipelines. *SIGGRAPH Course Notes: Beyond Programmable Shading* (2010). 1

[LH06] LEFEBVRE S., HOPPE H.: Perfect spatial hashing. *Proc. SIGGRAPH, ACM TOG 25*, 3 (2006), 579–588. 3

[LJ99] LAPIDOUS E., JIAO G.: Optimal depth buffer for low-cost graphics hardware. *Proc. SIGGRAPH/EUROGRAPHICS Hardware Workshop* (1999), 67–73. 1

[Mas15] MASON W.: Oculus is working on eye tracking technology for the next generation of VR. http://uploadvr.com/oculus-is-working-on-eye-tracking-technology-for-next-generation-of-vr, 2015. 2

[MKG*97] MARKOSIAN L., KOWALSKI M. A., GOLDSTEIN D., TRYCHIN S. J., HUGHES J. F., BOURDEV L. D.: Real-time nonphotorealistic rendering. *Proc. SIGGRAPH* (1997), 415–420. 3

[Ras01] RASKAR R.: Hardware support for non-photorealistic rendering. *Proc. SIGGRAPH/Eurographics Hardware Workshop* (2001), 41–47. 3

[RC99] RASKAR R., COHEN M.: Image precision silhouette edges. *Proc. I3D* (1999), 135–140. 3

[Rob63] ROBERTS L.: *Machine perception of three-dimensional solids*. Tech. Rep. TR 315, Lincoln Laboratory, MIT, 1963. 1, 2

[SEH08] STROILA M., EISEMANN E., HART J.: Clip art rendering of smooth isosurfaces. *IEEE TVCG 14*, 1 (2008), 135–145. 2

[Sol15] SOLLEFELDT R.: A look at the PowerVR graphics architecture: Tile-based rendering. http://blog.imgtec.com/powervr/a-look-at-the-powervr-graphics-architecture-tile-based-rendering, 2015. 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. 2

[WS94] WINKENBACH G., SALESIN D. H.: Computer-generated pen-and-ink illustration. *Proc. SIGGRAPH* (1994), 91–100. 2

[WSC*05] WANG J., SUN J., CHE M., ZHAI Q., NIE W.: Image space silhouette extraction using graphics hardware. *Proc. ICCSA* (2005), 284–291. 3

© 2016 The Author(s)
Eurographics Proceedings © 2016 The Eurographics Association.