# A HIGH PERFORMANCE VECTOR RENDERING PIPELINE

BY

APOLLO ISAAC ORION ELLIS

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

## ABSTRACT

Vector images are images which encode visible surfaces of a 3D scene, in a resolution independent format. Prior to this work generation of such an image was not real time. As such the benefits of using them in the graphics pipeline were not fully expressed.

In this thesis we propose methods for addressing the following questions. How can we introduce vector images into the graphics pipeline, namingly, how can we produce them in real time. How can we take advantage of resolution independence, and how can we render vector images to a pixel display as efficiently as possible and with the highest quality.

There are three main contributions of this work. We have designed a real time vector rendering system. That is, we present a GPU accelerated pipeline which takes as an input a scene with 3D geometry, and outputs a vector image. We call this system SVGPU: Scalable Vector Graphics on the GPU.

As mentioned vector images are resolution independent. We have designed a cloud pipeline for streaming vector images. That is, we present system design and optimizations for streaming vector images across interconnection networks, which reduces the bandwidth required for transporting real time 3D content from server to client.

Lastly, in this thesis we introduce another added benefit of vector images. We have created a method for rendering them with the highest possible quality. That is, we have designed a new set of operations on vector images, which allows us to anti-alias them during rendering to a canonical 2D image.

Our contributions provide the system design, optimizations, and algorithms required to bring vector image utilization and benefits much closer to the real time graphics pipeline. Together they form an end to end pipeline to this purpose, i.e. ”A High Performance Vector Rendering Pipeline.”

*To Mom, Dad, and Tenaya*

## ACKNOWLEDGMENTS

I wish to thank my family: my mother and stepfather Linda and Tony Tiritilli, my father and stepmother Gilbert and Janice Ellis, my sister Tenaya, my brothers Noah and Tymonn, and all my wonderful nieces and nephews. I love you all.

Sylvia Du, thank you for being part of my journey this last year, and for your love and support including, getting engaged to me, and moving with me to a small town in the midwest to watch me suffer through the last stretch of my doctorate degree.

My friends have been integral to my success. They have set bright examples, showed me acceptance and patience, always been encouraging, and have been the best of companions. I was blessed to find them. In order of appearance I wish to thank Evan Battaglia, David Miller, Justin Bledin, Stephen Butler and his wife Kayla Wesley, Vladimir Zyuzin, Sean Keely, Warren Hunt, and last but in no conceivable way least, my best friend, Cameron Alston and his wife Karen Chu.

To Rastislav Bodik, thank you for supporting my decision to attend graduate school, and writing what was apparently a glowing letter of recommendation, which I don't know about, obviously. And to James F. O'brien, thank you for continued support and encouragement over the course of my educational career.

For support, positive influences, and friendship both in and out of the lab, I wish to acknowledge Donald Fussell, Sean Keely, Sarah Abraham, Peter Djeu, and everyone in the high performance graphics and parallel systems lab at UT Austin, not to forget Peter Shirley. I could not have done this without you guys.

My experiences at Sony, Intel, AMD, Apple, Facebook, and Nvidia were guided by great leaders and teammates. Thank you all for the patience, challenges, and encouragement.

I would not be here now if not for my incredible committee. To John and Warren, I am forever grateful for you guiding me through this journey from start to finish, and to Steve and Sanjay thank you for coming along just in time, to provide mentorship, and more than fair quarter.

To Ashwin Vijay, Eric Huber, and Chris Widdowson, thank you for being inspiring, honest, so very helpful all this time, you are my graphics lab.

To the many different members of the graphics community, whom I have interacted with, and been mentored by over the years, thank you for setting directions and examples, and for paving the road ahead.

To the congregation at University Baptist Church, Pastor Brett and Tammi Smith, Marc

Taylor, Matt Robertson, Jessica Boakye, Michael and Amy Agapito and everyone else, thank you for support and fellowship, and may God bless you all.

To everyone else, I have in fact been writing a memoirs lately, and I remember you all. Thank you.

## TABLE OF CONTENTS

# CHAPTER 1: INTRODUCTION

Rendering is a field concerned with making images. Images in computer graphics are typically stored and displayed as grids of pixels, or some other discretized element of color. Vector images are an alternative format for storing images. They are continuous signals of color as opposed to discrete ones. Avoiding discretization avoids loss of useful information, and also keeps images in a more compact form.

In order to achieve the task of displaying 3D geometry on most any device currently in use, we must settle on pixels. However, vector images can play a valuable role in rendering systems. They provide a compact resolution independent format for an image of a 3D scene, which can be used to reduce bandwidth usage for many applications, and can be rendered with optimal anti-aliasing. We explore these ideas in the coming sections.

## 1.1 WORKING WITH VECTOR IMAGES

Vector image formats such as planar maps provide several interesting benefits to graphics systems and in particular to real time renderers and computer games. The value stems from their resolution independence.

In the common case, a vector image representing a 3D scene, i.e. its planar map representation, is much smaller than it's raster image. This is beneficial in rendering applications, including deferred shading, and cloud gaming. In deferred shading, visibility information is stored in memory for later processing so it's important to maintain it in a compact form [1] [2]. In cloud gaming images are transferred across the internet, and transferring compressed vector images is cheaper in terms of bandwidth and compression loss, than video streaming [3].

In a real time graphics pipeline, resolution independence i.e. being able to access the image of the scene continuously, yields the ability, among other things, to render and analytically anti-alias an image at an arbitrary resolution. In the current landscape of growing display resolutions this is extremely valuable.

### 1.1.1 Generating Vector Images with SVGPU

The above gives the motivation and driving forces behind putting vector images into the real time graphics pipeline. In the applications above, when exploiting the benefits of vector images, they needed to be generated in real time in both cases, cloud gaming, and dynamic content with anti-aliasing. Indeed except in the case of vector textures, vector images need to be generated in real time in order to be consumed by the real time graphics pipeline. Currently there is no way to do this efficiently. There is no real time system. Thus the design of a real time vector renderer is important.

Our implementation of a vector renderer is GPU based. SVGPU, or Scalable Vector Graphics on the GPU, is a novel real time system for rendering 3D scenes to vector images e.g. planar maps. Our system provides the first real time results for vector rendering [4].

The core of SVGPU is a three stage visibility pipeline which runs entirely on the GPU. After tiling and binning, the first stage in our visibility system implements 2D spatial hashing for silhouette edge extraction. The second stage of the pipeline is a general purpose parallel triangle-edge clipper, based on a 2D version of Bernstein and Fussell [5] [6]. The third stage of the pipeline is an occlusion determination phase, which has been vastly simplified by relying on the notion that visibility ambiguities have been resolved by clipping.

SVGPU relies on many sources of efficiency and tuning. First, clipping only needs to occur between silhouette edges and other geometry, this is why we extract silhouettes using spatial hashing. Second we perform a trivial rejection step, i.e. we stream through the triangle and edge data, checking if pairs actually need to be processed. Third, clipping is tuned for parallel efficiency, using double buffering, and dynamic work distribution. We also change our tile/bin resolution mid pipeline to tune the performance for occlusion calculations, which benefit more from having more bins than clipping does. This is due to the brute force $n^2$ nature of those calculations. Lastly, a trivial point in polygon test from the centroid of a triangle against all triangles in a bin suffices to resolve all occlusions correctly.

We validate SVGPU over several common scenes including some pathological situations designed to tank performance. Our results show roughly a 5x improvement over state of the art. We range from being real time for low complexity scenes, on the order of 50k to 200k polygons, to running on the order of 500ms for million plus triangle scenes with huge arrays of silhouettes.

The key contributions necessary for such a real time vector rendering system are elimination of excess clipping work by hashing silhouettes and clipping/rejecting valid triangle silhouette edge candidate pairs, bin tuning, high performance clipping parallelization, and efficient occlusion tests.

#### 1.1.2 Planar Map Streaming

Cloud gaming systems offer numerous benefits to both users and developers of game content, and they also suffer from numerous road blocks.

Developer benefits include increased security for game assets, lower distribution costs, and decoupling of game content from console hardware. Users also benefit from device agnostic gaming, leaving them untethered from OS or hardware. There is no need for game asset or software updates and patches, downloads of the games themselves, or installations. Furthermore users retain instant access to games at home or on the go.

Cloud gaming systems still have some serious bottlenecks. Network delay is most prominent, but systems must also consider resource allocation, response time management, graphics and video streaming, adapting to network conditions, and managing perceived quality by user. Some of these may seem related, but can effectively be treated separately since clever ideas have arisen to optimize different components by sacrificing performance, or quality in others. For an extensive treatment of cloud gaming services refer to the survey on cloud gaming by Cai et al. [7].

What we have done is proposed and evaluated a 3D content streaming system that relies on the rendering, compression, and distribution of planar maps.. In the simplest case, visibility calculations are done on the server producing a planar map for shading on the client. The planar maps compactly encode what is being rendered for transport. If the system is allowed to be a bit less transparent we can store a data base on the client of geometry per level or scene. In this way we can send primitive ids and compressed barycentric coordinates only. Once a compressed planar map (ids and barycentrics) reaches the client, we decompress and walk through the id list pulling the primitives out of the data base, and reconstruct each primitive that is visible in the planar map, using barycentric interpolation.

We experiment with a pipeline of compression techniques to further encode the planar maps efficiently. We have experimented with various flavors of each stage but leave that to the interested reader [3]. We perform uniform quantization, we show that delta prediction is not effective, and we settle on LZMA entropy encoding. We built an end to end test platform for measuring quality and comparing against state of the art video compression. We yield exciting results in perceptual video quality for fixed bandwidth and less bandwidth for fixed video quality including findings that x265 codes fall behind us in SSIM by up to 0.14, and of course being a fixed size moving to 2K to 4k resolution widens the performance gap. Future experiments will include work on client render time, and testing how thin we can allow user hardware to be.

### 1.1.3 Rendering and Sampling Vector Images

Vector graphics images are resolution independent, and can produce crisp raster images if sampled and anti-aliased cleverly. Seeking to anti-alias them has recieved a lot of attention.

Anti-aliasing of text and vector primitives embedded in 3D environments in the form of vector based textures or text fonts is hard and has been pursued avidly.

The problem with the approximations to date, is new platforms. In VR and especially AR, in-environment GUI clarity and text readability is harder to manage. A user in either VR or AR can orient GUI surfaces and text objects at arbitrary degrees of isotropy. Coupled with low relative pixel density, anti-aliasing quality in these environments becomes much more important. However, more accurate techniques such as super sampling are prohibitively expensive.

We propose a technique that adopts analytic anti-aliasing, using direct pixel-primitive-integrals. This is a hard problem, we are clipping an arbitrary primitive against an arbitrary pixel footprint. It is typically slow to compute, which is why this approach is not usually taken. The novelty in our work lies in our approach. We present clever math operations that drastically simplify integral calculation for any primitive, making the computation extremely rapid.

We also produce optimized kernels that compute coverage for shapes often found in font encodings, and vector images as well. These utilize the simplifying assumptions produced by the math we introduce, to compute pixel-primitive integrals extremely rapidly.

## 1.2 SUMMARY

This dissertation is divided into four chapters, fast vector rendering, applications of vector images in client-server rendering, and fast high quality rendering of vector primitives.

In chapter 2 , we ground our work in the literature by looking at previous techniques that produce vector images as outputs. We then introduce our pipeline for rendering 3D scenes to vector images in real time SVGPU, Scalable Vector on the GPU. We discuss it's implementation on GPUs, performance, and quality results.

In chapter 3 we ground our application of vector rendering to cloud gaming in it's field of competing applications. We then delve into the design evaluation and decisions, and results for our new system using SVGPU rendered vector images for transmission in a cloud gaming infrastructure which reduces bandwidth and improves scores on various image quality and network performance metrics.

Finally, in chapter 4 we survey literature on fast high quality rendering of vector graphics, grounding our work on fast rendering of high quality anti-aliased vector images. We present our method for integration of pixel differentials over vector font glyphs and provide real time results for analytic anti aliasing using our new math for graphics and optimized covergae computation kernels.

We conclude this thesis with a summary of vector rendering, streaming vector images for cloud gaming, and high quality rendering of vector graphics.

# CHAPTER 2: SVGPU: REAL TIME 3D RENDERING TO VECTOR GRAPHICS FORMATS

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.

## 2.1 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. 2.2 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. [8].

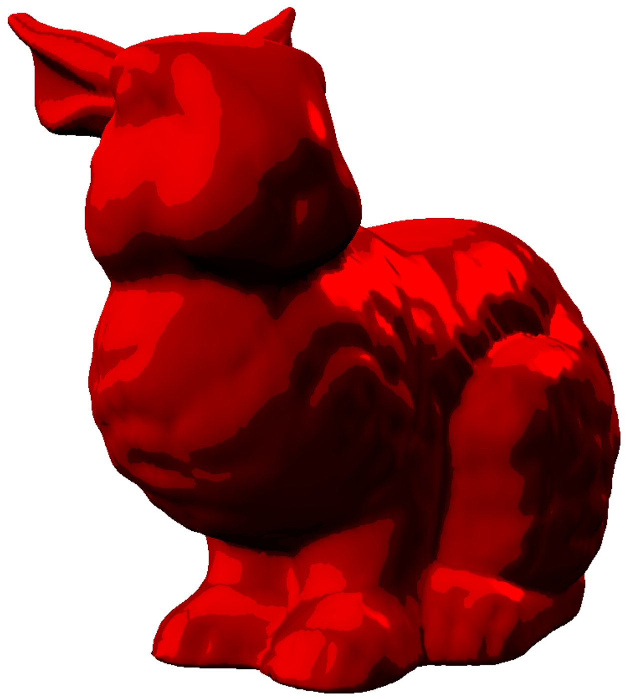


Figure 2.1: A toon shaded Stanford bunny of 70K triangles rendered as a resolution-independent planar map of visible triangle portions (depth complexity is no greater than one) in about 15 ms (67 Hz) by the SVGPU vector renderer, representing more than a four-fold improvement over previous GPU vector 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) [9, 10]. 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 [11] 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.

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

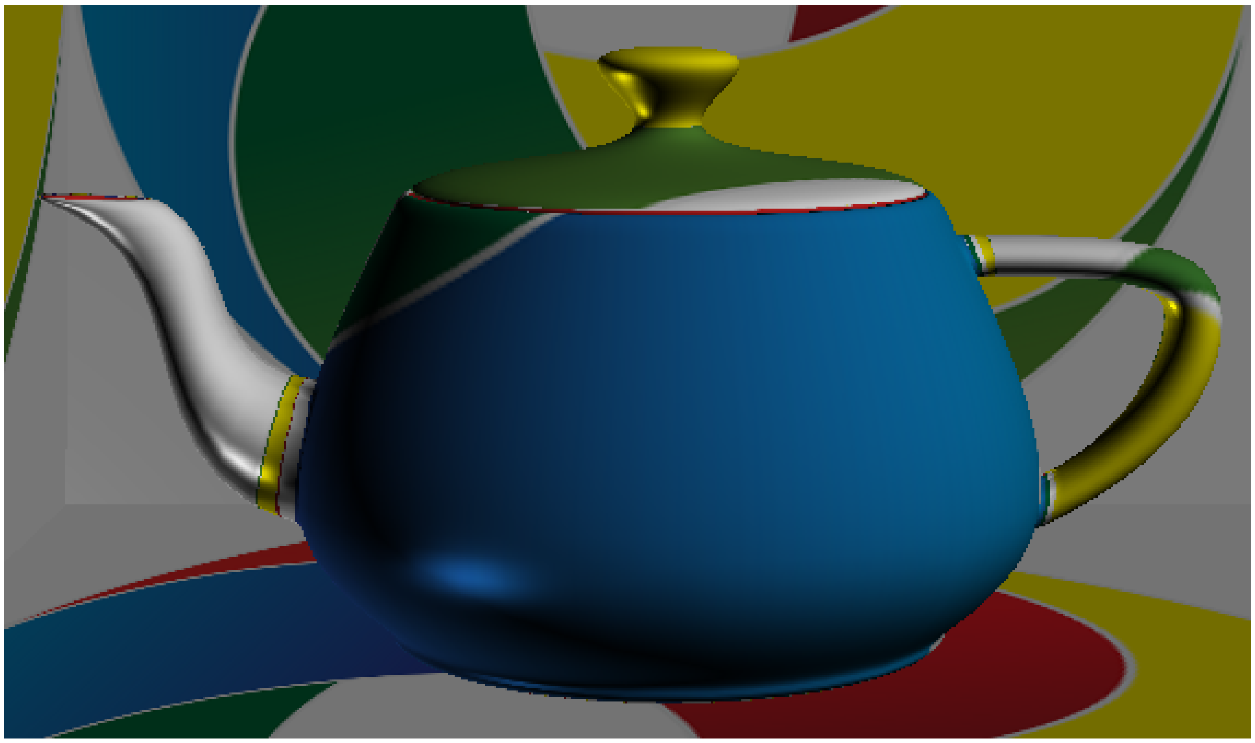
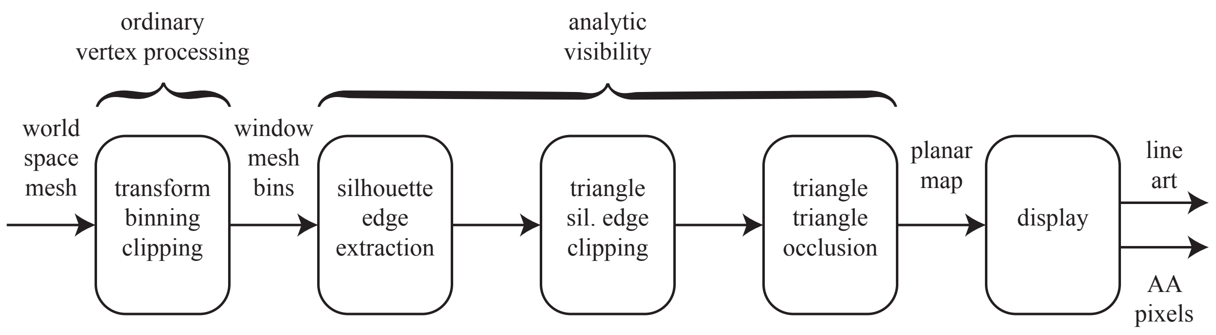


Figure 2.2: A vector image of an environment-mapped Utah teapot. The SVGPU renderer outputs a resolution-independent vector image as a planar map consisting only of visible screen triangles. The vertices of these screen triangles include properly interpolated attribute data including texture coordinates, such that a rasterizer can texture and shade its primitives at whatever raster resolution is desired, even variable resolutions for a foveated display.

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 [12] to support foveated rendering [13] 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.

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.



ordinary vertex processing
analytic visibility
world space mesh
transform binning clipping
window mesh bins
silhouette edge extraction
triangle sil. edge clipping
triangle triangle occlusion
planar map
display
line art
AA pixels

Figure 2.3: The stages of our binned vector graphics rendering system.

Our pipeline consists of five stages described in Section 2.3. 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 2.3.1 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 2.3.2 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 2.3.3 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 2.4.

## 2.2 PREVIOUS WORK

The hidden line problem was well studied decades ago [14]. Most modern approaches have been based on Appel's algorithm [15] 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 [11]. 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 [16, 17, 18, 19, 20, 21, 22], 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 for seed segments of the silhouette [23], 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 [24, 25] or their approximation [26]. Some have also developed hardware solutions for real-time NPR contour extraction [27, 28, 29].

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 [30], 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 [31], whereas our approach is triangle based, using the silhouette edges only for clipping to yield better performance results.

## 2.3 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.

### 2.3.1 Silhouette Extraction

SVGPU detects "silhouette" edges with a spatial hash table, which can be efficiently constructed and accessed on the GPU [32]. (We use the term silhouette 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.

### 2.3.2 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

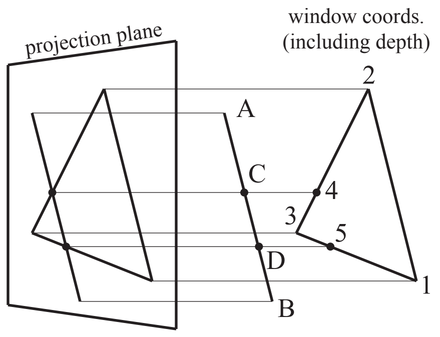


Figure 2.4: Geometric configuration for clipping $\Delta 123$ to silhouette edge $\overline{AB}$.

our pseudocode, and we will italicize their names.

Silhouette Clipping: Culling and Setup

The culling phase is used to remove most of the non-overlapping triangle-edge pairings from consideration for clipping. 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 2.4. 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 2.3.2. 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.

## 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

```
function ClippingKernel(ClipStructures)
    $T = TriCount * EdgeCount$
    $TrivialReject{<}{<}{<}T{>}{>}{>}(Bins, ClipStructures)$
    $T = ClipStructures.CandidateTriangles$
    $AllocateAdjacency{<}{<}{<}T{>}{>}{>}(ClipStructures)$
    $T = ClipStructures.CandidatePairs$
    $BuildAdjacency{<}{<}{<}T{>}{>}{>}(ClipStructures)$
    while ClipStructures.ActivePolygons do
        $T = ClipStructures.ActivePolygons$
        $Clip{<}{<}{<}T{>}{>}{>}(Bins, ClipStructures)$
        $ClipStructures.swapBuffers$
function TrivialReject($Bins, ClipStructures$)
    $Tri = ThreadId/SilhouetteCount$
    $Edge = ThreadId\%SilhouetteCount$
    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($Bins, ClipStructures, Round$)
    $PInfo = PolygonInfo[ThreadId]$
    $EInfo = EdgeInfo[PInfo.TriId]$
    $ClipEdge = EdgeList[EInfo.row_offset + Round]$
    $PData = PolygonData[PInfo.offset]$
    $NextPosition = ADD(PositionCounter, PInfo.Sides)$
    $RecheckTrivialReject(PData, CEdge)$
    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
            $EdgeOut = ClipEdge$
        $OutputVertex(EdgeOut, VertexOut)$
    $OutputInfo(NextPosition, ...)$
```

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 [33].

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.

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.

#### 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 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.

### 2.3.3 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

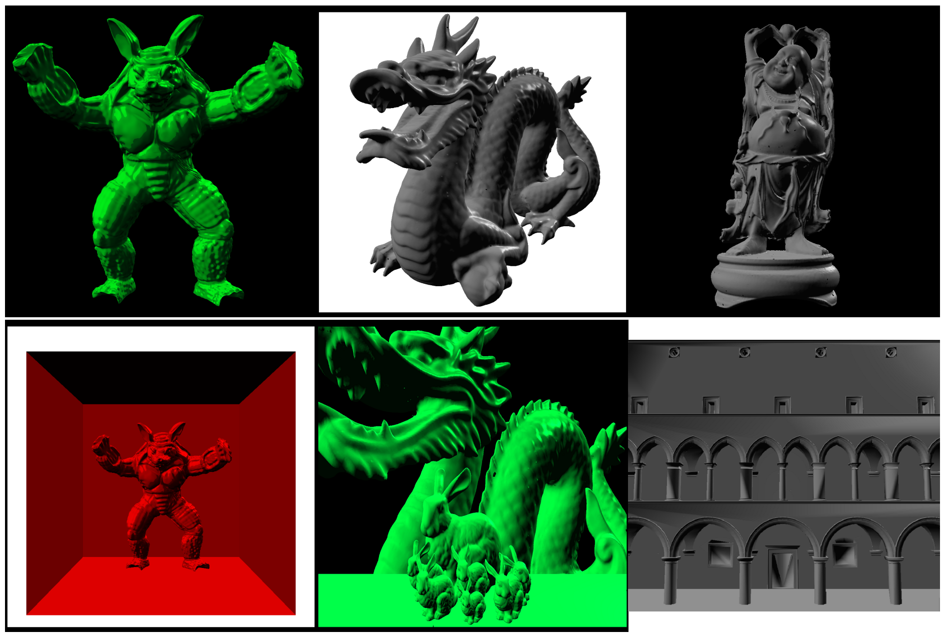


Figure 2.5: Benchmark scenes (l-r, t-b: Armadillo, Dragon, Buddha, Armadillo Box, Dragon Bunnies, Sponza) with various shaders applied using interpolated view space positions and normals, in addition to the Bunny and Teapot.

to reduce occlusion's all-to-all time complexity, as detailed in Section 2.4.1. 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.

## 2.4 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 2.5. The armadillo in the Cornell box

| 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 2.1: Profile of SVGPU run time performance (ms) per stage, using 1,024 silhouette clipping bins.

| Bins | Bun. | Arm. | Drag. | Bud. | Box | D+B | Sza. | Tea. |
|---|---|---|---|---|---|---|---|---|
| $\Delta$ 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 2.6: 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.)

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 2.1 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. 2.6 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

| 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 2.7: 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.)

| 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 2.8: 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.)

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. 2.7 shows that the number of silhouette clipping bins affects 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. 2.8 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

| 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) | | | | | | | |
| Clip Bin Usage | 1K | 8K | 497 | 6K | 191 | 3K | 75 | 2K | 6K | 26K | 2K | 12K | 729 | 7K | 243 | 2K |
| Occlude Bin Usage | 31 | 3K | 43 | 3K | 58 | 3K | 75 | 2K | 110 | 3K | 154 | 3K | 204 | 3K | 243 | 2K |
| Candidates | 333 | 2K | 153 | 2K | 76 | 1K | 39 | 1K | 2K | 9K | 705 | 4K | 284 | 3K | 113 | 2K |
| Clip Polygons | 214 | 1K | 108 | 1K | 54 | 854 | 26 | 713 | 685 | 3K | 322 | 2K | 149 | 1K | 66 | 585 |
| Vertex Buffer Out | 981 | 7K | 470 | 7K | 228 | 4K | 109 | 3K | 4K | 17K | 3K | 9K | 1K | 7K | 723 | 4K |
| Model (tris.) | Dragon (900K) | | | | | | | | Buddha (1M) | | | | | | | |
| Clip Bin Usage | 7K | 31K | 4K | 13K | 1K | 8K | 798 | 5K | 23K | 49K | 8K | 19K | 3K | 10K | 1K | 6K |
| Occlude Bin Usage | 215 | 3K | 284 | 3K | 356 | 3K | 490 | 2K | 118 | 2K | 155 | 2K | 190 | 2K | 338 | 3K |
| Candidates | 3K | 10K | 1K | 5K | 392 | 4K | 253 | 2K | 5K | 13K | 2K | 8K | 696 | 7K | 378 | 4K |
| Clip Polygons | 1K | 4K | 552 | 2K | 243 | 2K | 163 | 798 | 3K | 5K | 1K | 3K | 413 | 2K | 240 | 1K |
| Vertex Buffer Out | 7K | 18K | 3K | 9K | 1K | 7K | 723 | 4K | 13K | 28K | 5K | 15K | 2K | 9K | 1K | 7K |

Table 2.2: 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.

typical 4M $\Delta$/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 2.2 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). 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.

### 2.4.1 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 silhouette 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.

### 2.4.2 Comparison with Previous Work

Previous results from analytical visibility on the GPU [31] 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.

### 2.4.3 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 $64 \times 64$ 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 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.

# CHAPTER 3: PLANAR MAP STREAMING

We propose a new cloud gaming platform to address the limitations of the existing ones. We study the rendering pipeline of 2D planar maps, and convert it into the server and client pipelines. While doing so naturally gives us a distributed rendering platform, compressing 2D planar maps for transmission has never been studied in the literature. We propose a compression component for 2D planar maps with several parametrized modules, where the optimal parameters are identified through real experiments. The resulting cloud gaming platform is evaluated through extensive experiments with diverse game scenes. The evaluation results are promising, compared to the state-of-the-art x265 codec, our platform: (i) achieves better perceptual video quality, by up to 0.14 in SSIM, (ii) runs fast, where the client pipeline takes $\leq$ 0.83 ms to render each frame, and (iii) scales well for ultra-high-resolution displays, as we observe no bitrate increase when moving from 720p to 1080p, 2K, and 4K displays. The study can be extended in several directions, e.g., we plan to leverage the temporal redundancy of the 2D planar maps, for even better performance.

## 3.1 INTRODUCTION

Cloud gaming refers to: (i) running complex computer games on powerful servers in data centers, (ii) capturing, compressing, and streaming game scenes over the Internet, and (iii) interacting with gamers using thin clients on inexpensive computing devices [34]. In the past few years, we have witnessed strong interests in cloud gaming from both the industrial [35] and academic [36] sides. Existing commercial cloud gaming platforms are *video streaming* based, where the cloud servers perform all the rendering tasks, and the thin clients are merely video decoders. Such a design decision treats computer games as *black boxes* and allows cloud gaming service providers to trade *gaming experience* for *time-to-market*. Video streaming based cloud gaming, however, comes with several limitations, including *high bandwidth consumption, limited scalability, and little room for optimization*, and thus calls for *next-generation* cloud gaming platforms [37].

We make some observations on these three limitations:

- **High bandwidth consumption.** Although video streaming based cloud gaming imposes low computation requirements on thin clients, it incurs high networking bandwidth requirements. This is partially caused by *under-utilizing* the computing power of thin clients. Nowadays, even low-cost smartphones, come with GPUs (Graphics Processing Units), which are certainly more capable than video decoders. *Moving some* *rendering tasks from cloud servers to thin clients may reduce the network bandwidth consumption.*

- **Limited scalability.** Since all the rendering tasks are done on cloud servers, supporting more gamers not only leads to higher bandwidth cost, but also results in higher computational cost on cloud gaming service providers. This in turn makes the cloud gaming less profitable and not scalable to many gamers. *Distributing rendering tasks between cloud servers and thin clients may improve the scalability.*
- **Little room for optimization.** Treating computer games as black boxes prevents cloud gaming platforms from leveraging in-game context for performance optimization. *Extracting simplified forms of 3D scenes from computer games may open up a large room for optimization.*

We believe the crucial step of building next-generation cloud gaming platforms is to study the *rendering pipelines* of games for deeper integration between games and platforms.

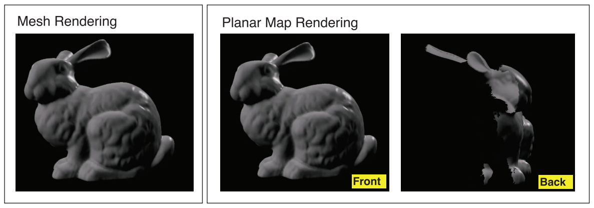


Figure 3.1: Traditional 3D mesh rendering vs. planar map rendering.

We study the rendering pipeline of *planar maps* [38, 39, 40], to understand its potential for addressing the three limitations. The planar map is a vector image consisting of points and edges in the plane, in our case, representing the visible triangles of a 3D scene. Planar maps were first proposed by Baudelaire and Gangnet [39] for graphic design. They define planar maps as 2D graphical objects of arbitrary complexity levels. Planar map tools were then implemented and optimized by Asente et al. [40] for interactive illustration systems. There was however no real-time solution to visibility computation for generation of planar maps until Ellis et al. [38] designed and implemented a planar map pipeline, which is around five times faster than previous work. Fig. 3.1 illustrates that (i) the 3D mesh rendering pipeline (on the left) renders everything, removing hidden surfaces on the fly via z-buffering, and (ii) the 2D planar map rendering pipeline (on the right) only renders visible triangles, where the depth complexity is reduced to one. Therefore, planar maps are concise and suitable for efficient rendering and streaming. Moreover, because planar maps are vector images, they scale for especially ultra-high-definition displays. Hence, planar map is an enabler for next-generation cloud gaming platforms because it may: (i) reduce the bandwidth consumption,

(ii) increase the scalability, and (iii) better optimize gaming experience.

We apply planar maps in cloud gaming platforms. This is achieved in two major steps. As the first step, we propose a distributed pipeline for *planar map rendering*. The crux of the new pipeline is the *compressor*, as the compression of planar maps has never been investigated in the literature. In the second step, we design a compressor for planar maps, consisting of several parameterized modules. Using real game scenes, we systematically derive the optimal parameters for a compressor that is specifically designed for *planar map streaming*. Our experiment results are quite promising. Compared to video streaming based platforms, our planar map based platform: (i) achieves higher perceptual video quality at the same bitrate, (ii) supports complex scenes when considering perceptual video quality, (iii) runs fast, especially at the client side, and (iv) scales well to ultra-high-resolution displays.

## 3.2 RELATED WORK

Cloud gaming platforms [36, 41, 42] can be roughly divided into three groups [43]: (i) video, (ii) graphics, and (iii) hybrid streaming. Video streaming refers to gaming platforms in which each frame is rendered completely on the server, and the frames are compressed into video streams for transmission to the client. Graphics streaming describes transmission of game scene data and/or rendering commands of each frame from server to client. Hybrid streaming as implied by the name refers to some clever combination of the above technologies. We explore a new graphics streaming approach using planar maps for cloud gaming.

There are several compression algorithms proposed for graphics streaming. For example, Meilander et al. [44] propose: (i) a caching mechanism for rendering commands, (ii) a compression algorithm for rendering instructions, and (iii) multi-layer representations of 3D objects. Nan et al. [45] introduce a hybrid delivery approach, where the server progressively streams the encoded frames and the graphics information. Similarly, Chuah et al. [46] aim to fully leverage the computational power on the client by rendering the low-quality base layer locally, while the server transmits a high-quality enhancement layer. Compared to the existing cloud gaming platforms, our platform: (i) naturally achieves distributed rendering and brings cloud gaming to inexpensive computing devices with weak GPUs, (ii) produces and compresses concise data representations for clients with limited network bandwidth, and (iii) scales to high-resolution game scenes for large displays.

## 3.3 RENDERING PIPELINE OF PLANAR MAP

Planar map rendering pipelines, such as the one proposed in Ellis et al. [38], have not been customized for distributed rendering in the literature. The input of the whole rendering pipeline is the gaming 3D scene and gamer's viewing information. The planar maps are generated in the following components: (i) silhouette detection, (ii) silhouette clipping, (iii) triangle-triangle occlusion, and (iv) vector rendering.

**Silhouette detection.** The term silhouette is used loosely here to refer to the visual edge or convex contour edge shared by both a frontfacing and backfacing triangle. We leverage the vertex geometry properties instead of mesh topologies to efficiently detect silhouette. In particular, we use a hash table to record all the edges in the 3D scene as entries. Once the hash table is constructed, we detect the silhouette edges by checking whether the corresponding face normals have opposite signs in the $z$ component. Lastly, we mark an edge shared by more than two triangles as a silhouette.

**Silhouette clipping.** We clip each triangle according to those detected silhouettes. First, we remove the non-overlapping triangle-silhouette pairs using their geometric information. Secondly, we clip each triangle against its list of overlapping edges. We apply a version of Bernstein and Fussells' clipping algorithm [47] to handle clipping with reduced error. Lastly, we tessellate all output polygons from this phase back into triangles.

**Triangle-triangle occlusion.** The important property we use to remove occluded triangles is that no triangle is partially occluded. That is, we adopt a lightweight centroid test to determine visibility. We discard the triangles whose centroid is overlapped by any other triangle.

**Vector rendering.** The vector renderer works as follows. First, the vector renderer generates the vertex attribute information including 2D position coordinates, texture coordinates, and normal vectors by barycentric interpolation. Barycentric coordinates are the input and output of the compressor/decompressor discussed later, and they are the main primitives of transmission. We thus interpolate vertex attributes from their original positions in the quasi-static database to the clipped positions. We pass the vertex data, as well as the scene information, such as viewing matrix, to the GPU to render the scenes for gamers.

## 3.4 USING A PLANAR MAP IN CLOUD GAMING

To leverage planar maps in cloud games, we divide the ordinary planar map rendering pipeline into the server and client sides. The first three components are put at the server side, and the vector renderer is put on the client side. The design rationale is to have the

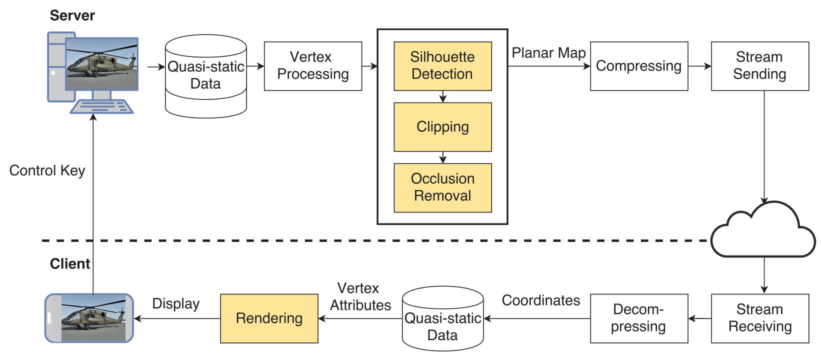
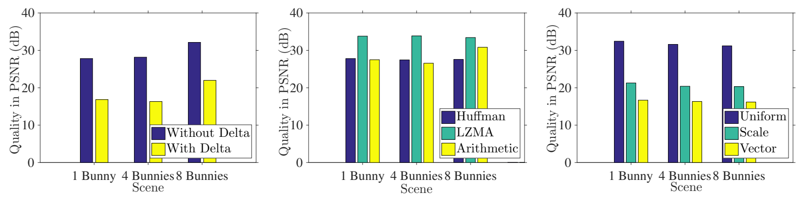


Figure 3.2: The revised planar map rendering pipeline for our cloud gaming platform. The shaded boxes are from the ordinary planar map rendering pipeline.

light weight vector renderer on the client, so that the thin client can render frames *merely* with per-pixel texturing and shading. To connect the server and client pipelines, three additional components are added to our platform: (i) *compressing*, (ii) *decompressing*, and (iii) *streaming sending/receiving*. We also include two quasi-static databases of 3D models (including vertices, texture, and shaders) at the server and the client. The database at the server contains 3D models for all game scenes, while the one at the client stores a subset of 3D models in the current and nearby scenes.

Fig. 3.2 shows the distributed planar map rendering pipeline, which works as follows. Starting from the server side, game scenes can be viewed as a set of 3D models from the quasi-static database in the model space. More specifically, scenes are represented in 3D models in files, such as obj files. With the typical 3D rendering transformations, we can take 3D models from model space, to world space, to view space, and finally into screen space using vertex processing. Then through the silhouette detection, clipping, and occlusion processes (see Sec. 3.3), we obtain 2D planar maps. The 2D planar maps are then compressed to further reduce the required network bandwidth and streamed to client side through the networks. At the client side, we decode the received data stream into coordinates in the decompression component. The quasi-static database at the client side is pre-populated offline, like most computer games. Note that all the quasi-static data are independent of the viewpoints and control inputs from gamers. Instead, they only depend on the game scenes determined by game states. With barycentric coordinate interpolation and the corresponding quasi-static data, we can render the game frames and display them to the gamer by GPU rendering.

The presented rendering pipelines have the heavier components at the resourceful cloud servers, and the lighter component at the thin clients with weak GPUs. Our key optimization problem is the *compression of planar maps* for mitigating high bandwidth consumption, which, to our best knowledge, has not been rigorously studied in the literature.



(a) Delta (b) Entropy (c) Quantized

Figure 3.3: Average video quality with different compression approaches of: (a) delta prediction, (b) entropy coding, and (c) quantization.

## 3.5 COMPRESSING PLANAR MAPS

### 3.5.1 Coordinate Systems and Data Format

While the planar maps in the ordinary rendering pipeline [38] are in barycentric coordinate system, the streamed planar maps can be represented using *Cartesian* or *barycentric* coordinate systems. Cartesian coordinates are relative to a single origin, and thus preserve the spatial property across vertices and among video frames. However, Cartesian coordinates do not leverage the (triangles of the) 3D models in the quasi-static database, which may result in unexploited redundancy. In contrast, barycentric coordinates describe each vertex information, including position, normal, and texture, within a triangle (in the quasi-static database) using three floating numbers in $[0, 1]$, say $u$, $v$, and $1 - u - v$. The merits of barycentric coordinates include: (i) shorter indexes for triangles, and (ii) common triangle patterns on unclipped triangles. However, barycentric coordinates are related to individual triangles, making the correlation among vertices harder to be leveraged. Since the Cartesian and barycentric coordinates both have pros and cons, we consider both coordinate systems for now.

Next we explain the format of the planar map data sent from the server to the client. Figures of detailed formats are omitted due to the space limitations. For each video frame, there are three headers: (i) model view matrix, (ii) draw call number, and (iii) draw call size. In particular, the model view matrix *transforms* the world space vertices into model space vertices; the draw call number and size facilitate multi-texture mapping. Moreover, each frame contains a set of *triangles*. With the Cartesian coordinates, each triangle is represented using vertex positions, vertex normals, world space coordinates, and texture coordinates. The vertex positions refer to the 2D positions in the screen space; the vertex numbers and world space coordinates are used for shading; and the texture coordinates are required by adding textures. With the barycentric coordinates, each triangle is represented by a triangle ID (in the quasi-static database), followed by 3 pairs of $u$, $v$ of individual vertices.

### 3.5.2 Compression Modules

Although the compression of planar maps has not been investigated, compression of 3D meshes is traditionally done in three steps: quantization, delta prediction, and entropy coding [48]. These common compression modules are described below.

**Quantization.** Quantization could be classified into uniform and non-uniform quantization. Non-uniform quantization, including *scalar* and *vector* quantization, is the foundation of floating-point number compression. For scalar quantization, we apply Lloyd's algorithm [49] on individual dimensions sequentially. For vector quantization, *all dimensions* are jointly quantized using K-means algorithm [50], in which all inputs are clustered in several groups, and the centroid of each group is determined.

**Delta prediction.** To leverage the properties that close-by vertices share similar information, a prediction algorithm may use previous coordinates to predict current coordinates [48], thus we adopt the delta prediction [51] approach. Only the first input is encoded into a symbol, as a 32 bit float. Others are represented in the deltas compared to the previous input.

**Entropy coding.** Entropy coding further exploits the different symbol frequency to reduce transmission bandwidth. We choose two widely used entropy codecs: *Huffman* and *arithmetic* coding [52]. The Huffman coding builds a Huffman tree with the more frequent elements at lower levels of the tree. We then assign shorter codes to the symbols closer to the root. For arithmetic coding, it converts the whole symbol sequence into a floating point between 1 and 0. The procedure loops through the symbols and shrinks the interval based on the symbol probability. In addition, we also consider the Lempel Ziv Markov Chain Algorithm (LZMA) and use 7-zip as its implementation. LZMA is a lossless dictionary compressor, which encodes a stream with an adaptive binary range coder. We notice that the coordinates may not be byte-aligned, which may be difficult for the entropy coders to handle. We therefore expand the symbols to the next byte boundary before entropy coding, e.g., 7-bit symbols are padded with one extra highest zero bit. Some sanity checks show that the padding strategy reduces the bitrate with no penalty on video quality.

### 3.5.3 Module Parameter Selection

We record three game scenes in 720p resolution, in which we vary the number of the popular Bunny model, among 1, 2, and 8. In every scene, bunnies with different textures and standing positions are placed on a plane, and the viewing position and orientation changes according to the gamer's speed. Each scene lasts for 10 secs at 30 Hz frame rate. We consider the following performance metrics to quantitatively compare module parameters.

- **Video quality.** The rendered quality in PSNR, SSIM, and Perceptual Evaluation of Video Quality (PEVQ). PEVQ is a video quality metric described in ITU-T J.247 Annex B [53], and implemented in Skarseth et al. [54].
- **Compression ratio.** The compressed stream size over the uncompressed stream size.

We send the game scenes through a compressor with different quantization, delta prediction, and entropy coding algorithms. We report four most important findings on the experiment results below.

**Quantization with Cartesian coordinates causes huge distortion.** Quantized Cartesian coordinates severely damage the video quality. A closer look indicates that this is because Cartesian coordinates have a very wide range: all real numbers are possible, making them more challenging to compress. On the other hand, all barycentric coordinates are between 0 and 1. Furthermore, Cartesian coordinates contain 30-dimension vectors, which dictate more bits to quantize.

**Delta prediction negatively impacts compression ratio as well as video quality.** We measure the PSNR and compressed stream size with and without delta prediction. Fig. 3.3a plots the achieved average video quality of different scenes with and without the delta prediction at a bitrate of 2.3 Mbps. We find that the delta prediction makes more variance on the resulting symbols, which then leads to lower compression ratio. Moreover, because barycentric coordinates are all positive numbers, applying delta prediction requires one extra sign bit, which worsens the performance.

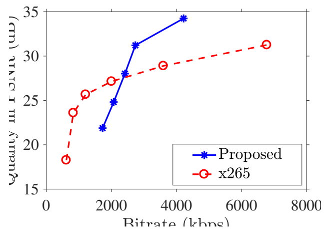
**LZMA outperforms other entropy coding algorithms.** We compress each scene multiple times with different entropy coders and plot the average PSNR at the same compression ratio using these entropy coders in Fig. 3.3b. It is clear that LZMA outperforms the other two entropy coders.

**Uniform quantization outperforms other quantization approaches.** On the 2D plane, the vector quantization divides the space into quadrilaterals, and the scale quantization divides the space into variable-size rectangles. In contrast, uniform quantization crops the space into equal size blocks. We plot the average PSNR from the three quantization approaches at the same compression ratio in Fig. 3.3c. This figure reveals that the uniform quantization outperforms others by far. We believe this is because the barycentric coordinates are between 0 and 1, and have weak clustering property.

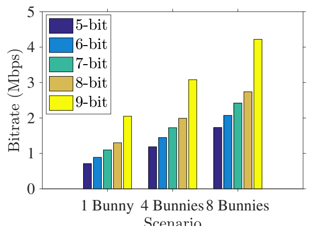
Based on the above findings, we adopt *barycentric coordinates, uniform quantization, and*

*LZMA coder* to compress the 2D planar maps. We note that while we report sample results in PSNR, results in SSIM and PEVQ (not shown due to the space limitations), also support the same findings.

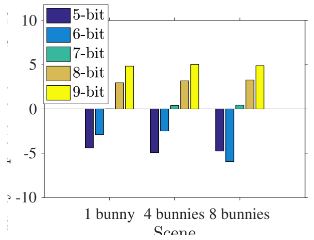
## 3.6 EVALUATIONS



(a) R-D curve



(b) Bitrate

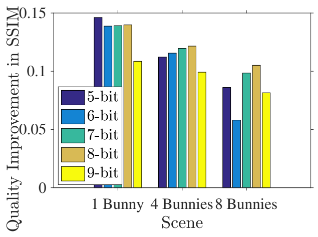


(c) Quality

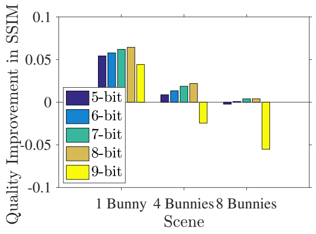
Figure 3.4: Performance of our proposed solution in PSNR with basic bunnies: (a) sample R-D curves from slow scene with 8 bunnies, (b) bitrate of our proposed solution at a bit-depth of 8-bits, and (c) quality improvement of our proposed solution.

### 3.6.1 Implementation and Setup

We have implemented the proposed server and client pipelines using a combination of C++ and Matlab. We compare our solution against the current cloud gaming platforms, in which all the rendering tasks are done at cloud servers. The rendered videos are compressed by the state-of-the-art video codec, such as x265 [55], and streamed to the clients. We do not compare against pure graphic streaming platforms, where all the rendering tasks are done



(a) SSIM



(b) SSIM Fine

Figure 3.5: Performance improvement of our proposed solution in SSIM, with: (a) basic bunnies and (b) fine-grained bunnies.

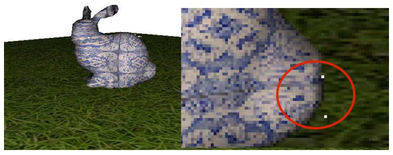


Figure 3.6: Rendered video frames with holes.

at the clients. This is because for the (not-so-thin) clients that have enough horsepower to render game scenes, the benefits of cloud gaming are limited. We expand the game scenes used in the last section. In addition to the number of bunnies, we also consider diverse : (i) model complexity, basic or fine-grained bunnies, and (ii) moving speed, slow or fast. In particular, we adopt 12 game scenes in our experiments. We consider three performance metrics: video quality in PSNR, SSIM, and PEVQ, bitrate in kbps, and component-wise running time in ms. We run the experiments on an i7 3.4 GHz PC with an NVidia Quadro M4000 card.

### 3.6.2 Results

**Potential of our proposed solution.** We first present the results from the basic bunnies at the low speed. We configure x265 with ultra-fast preset and compress each scene with 6 different Quantization Parameter (QPs) to get its Rate-Distortion (R-D) curve. For our proposed solution, we exercise the tradeoff between bitrate and quality using the bit-depth of the uniform quantizer. By varying the bit-depth between 5- and 9-bit, the resulting streams have different bitrates and video quality, which lead to the R-D curves. We first plot sample R-D curves from the scene with 8 bunnies in Fig. 3.4a. This figure shows that when bitrate is higher, the proposed solution outperforms x265 in terms of video quality in PSNR. The same observation holds in other scenes, although we cannot present all R-D curves due to the space limitations.

**Video quality improvement of our proposed solution.** We next compare the video quality in PSNR of our solution against that of x265. Fig. 3.4b presents the bitrate of our solution under different bit-depths. Upon having the bitrate of each game scene, we derive the expected video quality of x265 by performing linear interpolation on its R-D curve. Fig. 3.4c plots the quality improvement of our proposed solution over x265. This figure shows that up to 5 dB improvement is possible, and as long as the bit-depth is $\geq 7$ bits, our proposed solution results in higher PSNR.

**Implications of complex game scenes.** Our proposed solution may be more vulnerable to complex game scenes, in comparison to existing cloud gaming platforms that stream videos at a fixed resolution. Our game scenes contain either basic bunnies, each with 37,677 vertices on average, or fine-grained bunnies, each with 200,700 vertices. We find that our proposed solution leads to worse PSNR and SSIM when the scenes contain fine-grained bunnies. This is not a huge concern, because in cloud gaming graphics design, we can use simple 3D models with pre-rendered textures to achieve the same visual result as fine grained models.

**Perceptual video quality metrics.** Figs. 3.5a and 3.5b plot the video quality improvement of our proposed solution in SSIM with basic bunnies and fine-grained bunnies. The gap is as high as 0.14 in SSIM. However, when using PEVQ as the quality metric, x265 scores up to 4 and our solution scores up to 1.5. The reason of the inferior PEVQ scores of our solution may be due to some holes in the resulting videos, which is shown in Fig. 3.6. These holes may negatively affect the PEVQ scores; mitigating these holes is among our future tasks.

**Per-component running time.** We report the average running time per video frame in Table 3.1. We only consider the computationally intensive components, and more complex game scenes (8 bunnies). This table shows that the running time of the client side component is much smaller compared to those of the server side components. Although the rendering time is measured on a workstation in our experiments, the negligible values of rendering component ($\leq 0.83$ ms on average) reveal that porting it to resource-constrained clients is possible.

| | **Server** | | | **Client** |
|---|---|---|---|---|
| | **Detection** | **Clipping** | **Occlusion** | **Rendering** |
| **Basic** | 0.27/0.28 | 25.56/33.41 | 1.94/2.68 | 0.22/0.48 |
| **F.G.** | 3.66/4.73 | 60.55/88.14 | 17.43/24.02 | 0.83/3.13 |

Table 3.1: Running Time (ms), Average/Maximum, 8 Bunnies

**Implications of diverse speed and resolution.** We observe virtually no difference in terms of bitrate of slow and fast game scenes. At a bit-depth of 8, the rate increase due to the speed is merely 0.04% on average. This can be attributed to the fact that we haven't leveraged the temporal redundancy, which is among our future tasks. We also compare the bitrate of our proposed solution and x265 at higher resolutions of 1080p, 2K, and 4K. We encode the game scenes using our proposed solution at 720p and 8-bit bit-depth, and get an average video quality of 31.66 dB in PSNR. We then increase the resolution, and compute the rate increases of achieving 31.66 dB. We find that, for our proposed algorithm, higher resolutions lead to lower bitrates (at the same video quality). More specifically, reductions of 15% (1080p), 13% (2K), and 12% (4K) are observed. A closer look indicates that most artifacts due to quantization happen at the corners of triangles. Such negative impacts are diluted under higher resolutions. If we perform the same analysis on x265, higher resolutions lead to higher bitrate; increases of 18% (1080p), 22% (2K), and 30% (4K) are observed. This shows that our proposed solution has more potential in the future, where ultra-high-resolution displays become more popular.

# CHAPTER 4: SHAPES ON A PLANE

When embedded in 3D environments, UI clarity and text legibility are important to get right. The user can orient surfaces at various isotropic degrees, and in VR and AR, the pixel densities are low relative to user's eye distance. In fact text legibility is currently one of the biggest problems in VR and AR. It is known to influence and sometimes drives display resolution choices. It is hard to get these features right because of aliasing.

We propose a purely analytic anti-aliasing technique, which computes the coverage of vector primitives over pixel footprints directly. This is a hard problem, and typically slow to compute. However, using new methods we are able to solve it efficiently. We present here techniques that make the computation extremely rapid.

## 4.1 INTRODUCTION

Anti-aliasing of text and vector primitives embedded in 3D environments has been pursued avidly. The problem remains, how does a system provide an efficient coverage value for a pixel foot print and a vector primitive? It has been approached with a variety of methods. These include mainly approximating primitives with signed distance fields, embedding a fixed number of primitive features into textures, or employing various flavors and combinations of these approaches. These methods all rely on some form of approximation due to the nature of the problem, it is difficult to efficiently compute the integral of a vector primitive over a pixel footprint directly. More accurate techniques such as super sampling are generally too expensive, and require many samples to approach high quality images.

We propose an analytic approach with extremely high quality and performance. Our contribution lies in providing new usage of linear algebra operations before shape over pixel integral calculations that drastically simplify the integration operation for any primitive. The operations transform the problem from shape integration with an arbitrary footprint, into shape integration with a unit square. The second contribution of this work is a set of highly optimized shape integration kernels which take full advantage of our problem transformation.

In this work we consider only text glyphs. We pre-process true-type font glyphs into trapezoidal and curved shapes which can be used to exactly represent any true-type font and a number of other vector primitives and formats. With the benefits of our math operations we can render text glyphs embedded in 3D for an incredibly low cost. Our current results place us on the performance spectrum between 1x and 4x super sampling with effectively optimal anti-aliasing.

We present work to date on vector graphics and text anti-aliasing in 3D environments in Section 4.2. Many of these works amount to data structures and sampling algorithms for vector textures, or utilize some form of signed distance field. We present our methodology for changing the integration space, and transforming primitives into this space for rapid integration in Section 4.3. Section 4.4 details the integration kernels. We present trapezoid and curve kernels, with algorithm walkthroughs to provide intuition into our methods. Section 4.5 briefly discusses our system implementation, and section 4.6 presents quantitative results and quantitative results. Section 4.7 provides additional pseudo code for our optimized integration kernels.

## 4.2 PREVIOUS WORK

### 4.2.1 SDFs and ADFs

Signed distance fields (SDFs), have been used for anti-aliasing by several authors. One of the most notable uses of SDF is for anti-aliasing text and vector graphics embedded in 3D environments. Green conducted anti-aliasing by storing distance values in the alpha channel of a texture, using smoothstep to transition towards the underlying surface. [56] Qin, et al. also anti-alias text by evaluating the SDFs on the GPU .

Most SDFs discretize surfaces by storing a representation of the distance to the closest primitive of an underlying surface in a spatial data structure. The most common method for propagating distances through a grid is Danielsson's 8SED method, or more accurately, the signed variant SSED8. The grid structure can then be used to quickly access information about the underlying surface, albeit with some error due to discretization. Aside from pixel error, when sharp or thin features with multiple edges are present in the SDF the evaluation of distance breaks down. Approximations then yield over-rounding at corners and artifacts on complex and thin features. These artifacts are exacerbated at higher resolutions. [57] Using multiple channels for averting these issues was studied by Chlumsky, but they were not completely eliminated. [58]

### 4.2.2 Vector Graphics

Nehab and Hoppe implement a Vector Graphics rendering scheme for the GPU pipeline. The approach works in a pixel shader in which they both anti-alias and super sample. They use a lattice data structure storing a list of vector primitives at each cell. They pre-filter each cell, with a ray cast from left side of pixel to determine closest segment distance. This is an approximation, as the ray may miss, since the segment may be sharp. They resort to mipmapping when minification causes the pixel foot print to extend outside of a lattice cell, and handle cubics by converting to quadratics pieces, and linear segments. In comparison to our work this technique is heavy handed and still approximate.

Qin et al. presented a fast method for computing distance to a curve for SVG strokes [59]. It is based on binary search; they store the curve in an acceleration structure using 3 textures with indirection. The curve is split in half at the parametric center, and a plane is constructed about the normal. They test hit points against this plane and continue on the point containing side until an error tolerance is reached or for a fixed number of iterations.

### 4.2.3 Features

One of the earliest feature based techniques was developed by Bala et al. who used features for reducing sample interpolation error across frames during rendering. Shading interpolation was previously employed in GI schemes such as irradiance caching, and render caches [60], whereas samples are interpolated within and across frames because shading is expensive. Bala et al. improved on these techniques by testing interpolations against a discontinuity invariant referred to as reachability. This ensures a shaded sample is not interpolated across a discontinuity which causes unnatural blurring and error. [61].

Feature based textures and feature curves follow Bala et al. in incorporating features into raster image lookup. [62] [63] They incorporate similar techniques, but with the goal of implementing feature aware bi-linear interpolation for textures. FBTs support Bezier primitives up to the cubic Bezier. They segment a texel into regions with one feature per region. Bi-linear interpolation then only samples other textures which are aligned in terms of region. They do not sample in directions that are not reachable from the current sample. [62] Feature curves augment the feature texture with SDF information and provide an implementation better adapted to the GPU. [63] While both of these techniques improve on bi-linear filtering during texturing, coverage was not considered.

Shadow silhouette maps augmented shadow maps with silhouette information to remove the jagged artifacts produced by magnifying the shadow map when viewing a sparsely sampled region up close. [64] Silhouette maps for textures extends the work to the texture magnification problem, and shows how to extend the shadow work to implement bi-linear filtering that respects silhouette boundaries, in textures. [65]

Vector texture maps, Pinchmaps, Bixels and Infinite resolution textures, each store curves or boundaries in quickly accessible structures for texture mapping with IRT being state of the art. However vector texture maps and pinch maps don't handle multi-facet features and intersections elegantly, and bixels are limited. [66] While IRT averts most these issues, the application to analytic anti-aliasing of arbitrary primitives, i.e. computing coverage, was not studied. [67] [68] [69] [66].

### 4.2.4 Ray casting curves

Recently authors have approached anti-aliasing for vector primitives by storing the primitives directly in texture format. Shape outlines are then sampled using fast winding number calculation and 2D sub pixel ray casting [70] [71]. However these implementations suffer from rounding error and produce artifacts. Most recently, these techniques have been improved upon by the Slug Library implementation by Eric Lengyel [72]. The approach eliminates precision issues, and is state of the art w.r.t. speed and quality. The library is not openly available, however we compare directly against a scene from a demo release.

## 4.3 PIXEL MATH

Our system, given a hit point on a surface generated with any visibility algorithm, uses the pixel differential on the surface to compute an inverse pixel transform. This transform changes the space of the integration calculation used for computing analytic coverage of a primitive into a unit square. In this section we provide the derivations and algorithms used to compute and apply this transform. We assume here, that the surface has been hit, or a sample on a surface has been taken, and that the pixel differentials at that hit point are available in UV space.

### 4.3.1 Pre-process

We discuss text rendering, though these approaches can be leveraged for vector images as well. We use true-type fonts for all experiments. Our system is built around integrating the vector primitives found in, or derived from true-type glyphs.

We break the glyphs down into two types of shapes by pre-processing them with the thorax true-type library. [73] Thorax computes a breakdown of a glyph into trapezoid for large non curved convex areas, and into bi-quadratic curves for convex curved areas. Concave curved areas e.g. the inside of an "O" are also represented by bi-quadratics. These curve representations are inverted which yields areas which are opposite in sign of convex areas. This causes canceling in the integration kernels which we discuss in the next section.

To enable the system to process shape over pixel integrals as rapid as possible, we maintain an invariant for shapes in true-type font glyphs. We clip curves such that the apex is never aligned with the horizontal axis, and we ensure all trapezoids also have x axis alignment in their parallel edges. The other edges of the trapezoid can be at arbitrary orientations, and the curves can otherwise, be arbitrary bi-quadratics.

### 4.3.2 Computing the transform

We compute a transform from pixel differential space to unit square space. Pixel differentials represent the transformation of a pixels image-space shape into the shape of it's projection onto a surface in 3D. A differential in UV space, is represent by two vectors, $dUVdx$ and $dUVdy$, which represent the difference in UV space along the horizontal axis and vertical axis of the original image space pixel respectively. We place these differentials into a $2x2$ matrix of the following form.

$$\begin{bmatrix} dUVdx_x & dUVdx_y \\ dUVdy_x & dUVdy_y \end{bmatrix}$$

For example a normalized pixel differential on the image plane would form the following matrix.

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

In general this matrix can represent a 2D affine transform, which may be decomposed into a non-uniform scale, a sheer, and a rotation. The matrix transforms a unit square, centered at the origin, (the pixel), into the shape of it's projection onto the UV plane.

A pixel for all practical purposes can be represented by a Gaussian. A Gaussian is radially symmetric. This means if possible, we can safely remove the rotation from the differential matrix. As it turns out this is quite possible. Using QR factorization, we can decompose the differential matrix into none other than a rotation and a shear. Upon inspection the bounding volume of a square transformed by the factored out shear will tightly bound the Gaussian representing the pixel area. See Figure 4.1.

Taking advantage of the rotational invariant property of the Gaussian, we can safely discard the rotation. Our choice of factorization operations is Givens rotations. Givens rotations compute the QR decomposition of a matrix, by zeroing out elements one at a time. The resulting matrix Q is an ortho-normal matrix, i.e. a rotation, and the matrix R is upper triangular i.e. an x-shear. In 2D, we only need to zero out one element, the y-shear

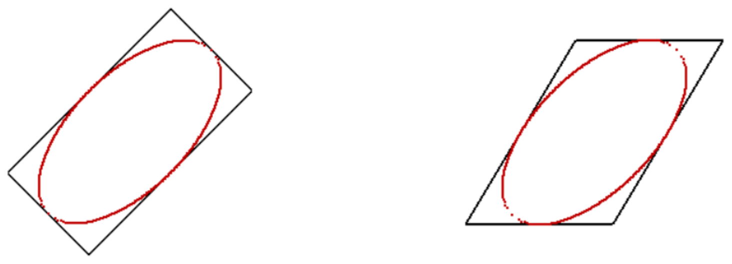


Figure 4.1: A rotation and scale,= (left), and the shear factored out of that matrix (right), both tightly bound the pixel Gaussian.

component. The code for our implementation of QR is shown in Figure 4.2.

```
matrix2x2 QRDecomp(matrix2x2 A)
    float a00 = A[0][0];
    float a01 = A[0][1];
    float a10 = A[1][0];
    float a11 = A[1][1];
    float csn = a00 / sqrt(a00*a00 + a11*a11);
    float sn = a11 / sqrt(a00*a00 + a11*a11);
    matrix3x3 Q = matrix3x3::identity();
    Q[0][0] = csn;
    Q[0][1] = sn;
    Q[1][0] = -sn;
    Q[1][1] = csn;
    #QR = A
    #R = Q^t*A
    return transpose(Q)*A;
```

Figure 4.2: Code for computing the QR decomposition of the 2D differential matrix.

We use only the shear going forward. It should be clear that, inverting the shear transform yields the transform which takes the pixel foot print in UV space into the a unit square. The inversion contains only an x shear, and thus maintains x axis alignment with the square. As mentioned we pre-process glyphs into trapezoids and bi-quadratic curves which we force to have x axis alignment. However, other vector primitives such as triangles, quads, and curves, can be decomposed such that they have x axis alignment. We ensure this in our pre-process. Thus we have achieved an important goal. Integrating an x axis aligned shape with a unit square is a much simpler process in comparison to the original problem.

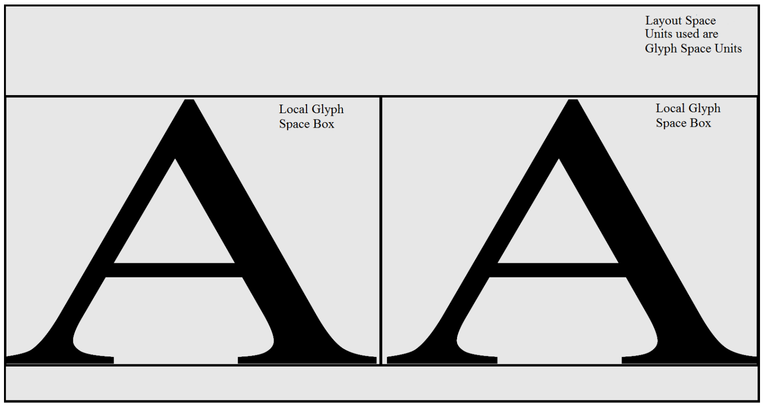


Figure 4.3: Two glyphs laid out in a text box.

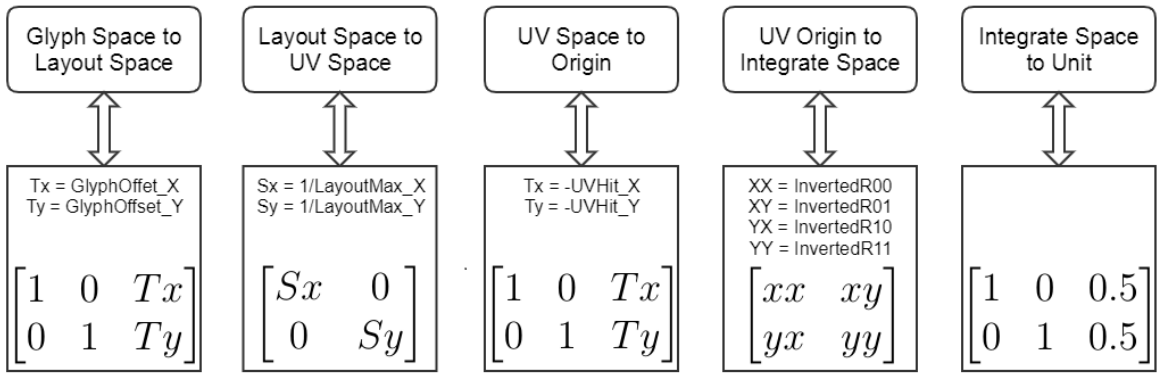

Glyph Space to Layout Space
Tx = GlyphOffet_X
Ty = GlyphOffset_Y
$\begin{bmatrix}1 & 0 & Tx\\0 & 1 & Ty\end{bmatrix}$
Layout Space to UV Space
Sx = 1/LayoutMax_X
Sy = 1/LayoutMax_Y
$\begin{bmatrix}Sx & 0\\0 & Sy\end{bmatrix}$
UV Space to Origin
Tx = -UVHit_X
Ty = -UVHit_Y
$\begin{bmatrix}1 & 0 & Tx\\0 & 1 & Ty\end{bmatrix}$
UV Origin to Integrate Space
XX = InvertedR00
XY = InvertedR01
YX = InvertedR10
YY = InvertedR11
$\begin{bmatrix}xx & xy\\yx & yy\end{bmatrix}$
Integrate Space to Unit
$\begin{bmatrix}1 & 0 & 0.5\\0 & 1 & 0.5\end{bmatrix}$

Figure 4.4: Transform Pipeline from glyph space to integration space at the unit square.

Taking full advantage of this math, integration kernels can now rely solely on trivial primitive operations and use factored down versions of most equations. The factoring is possible in the special case where primitive elements are being clipped against the unit square from $(0, 0)$ to $(1, 1)$. We discuss this in the next section.

### 4.3.3 Layout to Integrate

Now that the QR decomposition has been computed and we have the effective inverse of the differential transform, we can move shapes into the integration space at the square. This requires 5 transforms, although this is for clarity, some transforms can be concatenated.

Refer to figure 4.4 for the following discussion. Consider a glyph in glyph space bounded by the coordinates $(0, 0)$ and $(1024, 1024)$. Note, in true-type speak, this could be referred to as em space, or the em grid, see ”TrueType Fundamentals”. [74] Along with this we have defined a text box with two of these glyphs aligned horizontally, without spacing for simplicity. See Figure 4.3. The text box is also in em space in terms of units, but we refer to it as layout space. It encompasses the glyphs’ local em grids. It is a block of space in which glyphs are laid out with em size units. The example text box ranges from $(0, 0)$ to $(2048, 1024)$. If we are to move the right side glyph into layout space, we only need to translate from the origin of our layout space, $(0, 0)$, to the origin of the glyph in layout space, $(1024, 0)$. Next we divide the by the total number of units in layout space $(2048, 1024)$. So this first step requires only a translation and a squash, which can be computed before hand during glyph layout.

Once the glyph is in UV space, we move it to the origin, and transform it using the inverted shear. The first step involves, for each shape, subtracting the UV sample point from the UV coordinates of the shape. Lastly we transform the shape into integration space, and translate it to the unit square, where we will perform our integration calculations. As mentioned the shape remains axis aligned in x.

The full transformation pipeline is displayed in figure 4.4.

## 4.4 INTEGRATION KERNELS

Here we introduce two rapid integration kernels which integrate a shape against the unit square. The kernels compute areas as opposed to clipping. We walk through each kernel’s pseudo code, one instruction at a time to ensure clarity in reproduction. Luckily the kernels are short, and also branch free, which leads to a straight forward discussion. We refer to diagrams in each kernel discussion, to yield intuition into the general correctness of the operations.

### 4.4.1 Trapezoid Integration

Trapezoid integration pseudo code is displayed in figure 4.6. To begin, we’ll consider the class containing linear equation abstractions, also in figure 4.6. We use three linear equation structures for each trapezoid x0, x1, and y0. The inputs A and B are depicted for each equation in figure 4.5.

Lines 4 and 5 construct the parametric offset t of the linear equations at 0 and 1. The idea is to be able to clip the lines against sides, and top and bottom of the unit square. The code uses interval variables called ranges. Ranges automatically keep the upper and lower t values for a line’s upper and lower intercepts, sorted as (min,max) in, range.lower,

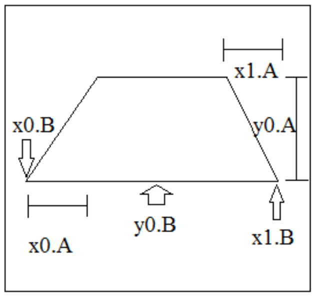


Figure 4.5: Depiction of the variables in the linear equation for trapezoid integration

range.upper, respectively. Note however that while the t values are sorted in the range, we also construct ranges for the actual coordinates at those t values without resorting those by min and max. So for a range storing x coordinates lower may not be the lower x coordinate, rather the x coordinate corresponding to the lower t value.

Moving into the trapezoid integration kernel, line 12 finds the vertical intercept t values as a range variable from the trapezoid equation y0, and clamps it to 0 and 1. This produces t of y0, $ty0$. Line 13 computes the y values themselves. This is done by adding the y coordinate y0.B to the parametric distance t, multiplied by the difference in y, y0.A. See Figure 4.5. This will generate the y coordinate range of y0, $yy0$. The result of the $yy0$ computation is depicted in figures 4.7 and 4.8 as $yy0.upper$ and $yy0.lower$.

Lines 15 and 16 are symmetric operations on the x0 line equation (left line) with the exception that the range $tx0$ is first clamped to $ty0$, this will clip the left line to the top and to the bottom of the square. $yx0$ is the y coordinate range of $tx0$, and $xx0$ is the x range at $tx0$. Line 17 is used in the area calculation to find the height of a point of mid way between the lower and upper y intercepts of the, potentially clipped, left line.

We'll look now to computing the areas, and the areas we compute have been overlaid in colored boxes on the reference figures. *madd* is a multiply add instruction, *nmadd* is multiply negate and add, *msub* is multiply subtract. Line 19 computes $ymid01$, which is the distance between the midpoint of $yx0$ upper and lower, and the bottom of the trapezoid $yy0.lower$. Line 20 computes the area of the rectangle of height $ymid01$ and width $xx0.lower$, the left most x coordinate. In figure 4.8 this area is zero. In figure 4.7 this is the light blue rectangular area.

Line 21 computes $ymid02$, which is the distance between the midpoint of $yx0$ upper and

```
 1 LinearEqn {
 2  LinearEqn(A, B) : a(A),b(B){
 3   i_a = -1.0f / a;
 4   i_b = i_a * b;
 5   tr = Range(i_b, i_b - i_a);
 6  }
 7  Apply(t) { return Range(madd(a, t.low, b), madd(a, t.up, b)); }
 8  Get(){ return tRange; }
 9  a,b,tRange;
10 };
11 IntegrateTrap(x0, x1, y0) {
12  ty0 = Clamp(y0.GetT());
13  yy0 = y0.ApplyT(ty0);
14 
15  tx0 = Clamp(x0.GetT(), ty0);
16  yx0 = y0.ApplyT(tx0);
17  yx0_mid = yx0.lower + yx0.upper;
18  xx0 = Clamp(x0.ApplyT(tx0));
19  ymid01 = msub(0.5f, yx0_mid, yy0.lower);
20  area = xx0.lower * ymid01;
21  ymid02 = nmadd(0.5f, yx0_mid, yy0.upper);
22  area = madd(xx0.upper, ymid02, area);
23 
24  tx1 = Clamp(x1.GetT(), ty0);
25  yx1 = y0.ApplyT(tx1);
26  yx1_mid = yx1.lower + yx1.upper;
27  xx1 = Clamp(x1.ApplyT(tx1));
28  ymid11 = msub(0.5f, yx1_mid, yy0.lower);
29  area = nmadd(xx1.lower, ymid11 , area);
30  ymid12 = nmadd(0.5f, yx1_mid, yy0.upper);
31  area = nmadd(xx1.upper, ymid12, area);
32  return area;
33 }
```

Figure 4.6: Trapezoid integration pseudo code

lower, and the top of the trapezoid $yy0.upper$. This is best seen in figure 4.8, because it differs from $ymid01$ in that image. In the simple image, figure 4.7, $ymid01$ and $ymid02$, and in fact all ymid calculations are identical, as such we have labeled that height as $ymidall$. However, as mentioned, in figure 4.8 the reason for difference in their computation is apparent. Line 22 computes the area of the rectangle of height $ymid02$ and width $xx0.upper$, this area is dark blue. The dark blue box in figure 4.8 shows the nature of this calculation when the

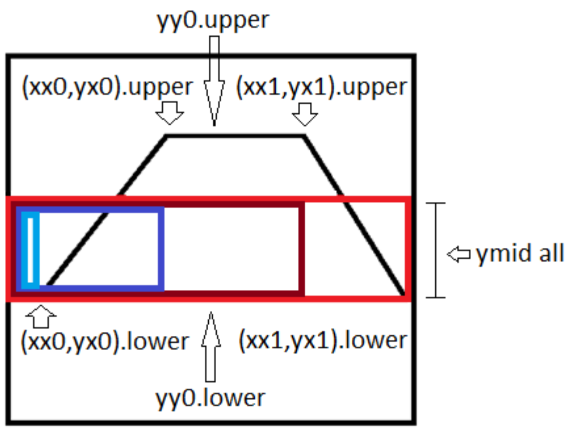


Figure 4.7: Trapezoid reference for pseudo code walk through, a simple case

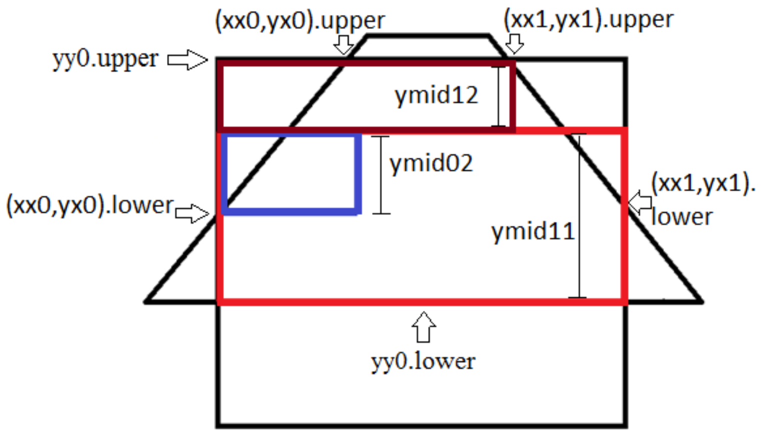


Figure 4.8: Trapezoid reference for pseudo code walk through, a more complex case

trapezoid is clipped. Note these are both positive areas.

Subtracting the light blue box from the dark blue box would yield an area equal to the left triangular region of the trapezoid, this can be seen by flipping the triangle formed by the region above the dark blue box down, forming the rectangular area that would be left if we subtracted light blue from dark blue. This subtraction happens in the next area calculation. As it stands we have double counted the light blue area.

Lines 24 through 27 are symmetric. We compute everything from the left side of the square at $x = 0$. Line 28 computes $ymid11$, which is the distance between the midpoint of $yx1$ upper and lower, and the bottom of the trapezoid $yy0.lower$. Line 29 computes the area of the rectangle with height $ymid11$ and width $xx1.lower$, this is shown in orange. Line 30 constructs $ymid12$, which is the distance between the midpoint of $yx1$ upper and lower, and the top of the trapezoid $yy0.upper$. Line 31 computes the area of the rectangle with height $ymid12$ and width $xx1.upper$, shown in dark red. Both of these areas are negative. The first areas, light and dark blue, are added in.

The double counted area under light blue cancels with both orange and red, leaving zero area. The dark blue area cancels with one of the overlapping regions leaving one negative accounting for the blue area. The large orange region accounts for the right triangular region, again seen by flipping the triangle down. It also accounts for the lower part of the body of the trapezoid. The dark red region then accounts for the upper body of the trapezoid.

The reader should note the regions are overlaid in the simple figure, where as in the more complex example, they are depicted in such a way as to invite intuition into their purpose in the calculations. Although these calculations seem unorthodox, they are symmetric, and were designed to optimize the area calculations to be simple rectangular area calculations in all cases. The design has lead to an incredibly rapid and efficient integrator.

### 4.4.2 Curve Integration

Curve integration pseudo code is displayed in figure 4.9. Note, while our quadratic equation evaluation and structure is optimized we have moved it to section 4.7. For following the code, it should suffice to note, a curve is defined by two linear equations x0 and y0, and two quadratics as they are bi-quadratic curves, x1 and y1.

We use figure 4.10 as a reference for this discussion. Lines 2 calculates the area of the trapezoidal region beneath the curve, a simplified version of the trapezoid clipper for a one sided region. This area cancels with the area calculations discussed shortly. Line 5 and 6 solve for t of the y apex of the curve and the delta in t to the y intercepts of the curve with the box i.e. at 0 and 1. $-b/2a$ of the quadratic equation $y = at^2 + bt + c$ can be thought of as the apex in t of the quadratic equation, this follows from that fact that the x coordinate of the *vertex* of a quadratic is defined as $-b/2a$, and plugging it into $y = at^2 + bt + c$ yields that y coordinate of the vertex. In this way $\sqrt{b^2 - 4ac}\ /\ 2a$ and $(\sqrt{b^2 - 4a(c+1)}\ /\ 2a)$ can be viewed as the offsets or deltas from the apex to the intercepts at 0 and 1, since they yield t values which when plugged into $y = at^2 + bt + c$ give the y values of those intercepts.

T of y1, i.e, t along the curve to the hit points, must be selected carefully based on the sign of the apex to ensure correctness. If the curve opens to the right t = 0 is at the bottom of the curve, it's the opposite for left opening. If the apex is negative, then either the bi-quadratic opens to the right and the apex is near the bottom, in which case we *add* the $ty1\_deltas$ to the apex to find the box hits, or it opens to the left and the apex is near the top in which case we also add the deltas. Otherwise we subtract them.

```
1  IntegrateCurve(x0, y0, x1, y1) {
2   area = CalculateTrapezoidalArea(x0,y0);
3
4   y1 = eqns.y1;
5   ty1_apex = y1.SolveApex();
6   ty1_delta = y1.SolveDelta();
7   ty1_apex_sign = ty1_apex & SignMask();
8   ty1 = ApexToTSelect(ty1_apex, ty1_delta, ty1_apex_sign);
9
10  x1 = eqns.x1;
11  tx1_apex = x1.SolveApex();
12  tx1_delta = x1.SolveDelta();
13  tx1_lower = tx1_apex−tx1_delta.upper, tx1_apex−tx1_delta.lower;
14  tx1_upper = tx1_apex+tx1_delta.lower, tx1_apex+tx1_delta.upper;
15
16  yx1_lower = y1.Apply(tx1_lower);
17  ymid_lower = yx1_lower.lower + yx1_lower.upper;
18  xx1_lower = Clamp(x1.Apply(tx1_lower));
19  ymid01 = msub(0.5f,ymid_lower,yy0.upper);
20  area = madd(xx1_lower.lower,ymid01,area);
21  ymid02 = nmadd(0.5f,ymid_lower,yx1_lower.upper);
22  area = madd(xx1_lower.upper,ymid02,area);
23
24  yx1_upper = y1.Apply(tx1_upper);
25  ymid_upper = yx1_upper.lower + yx1_upper.upper;
26  xx1_upper = Clamp(x1.Apply(tx1_upper));
27  ymid11 = msub(0.5f,ymid_upper,yx1_lower.upper);
28  area = madd(xx1_upper.lower,ymid11,area);
29  ymid12 = nmadd(0.5f,ymid_upper,yy0.lower);
30  area = madd(xx1_upper.upper,ymid12,area);
31  return area;
32 }
```

Figure 4.9: Curve integration pseudo code

Lines 10 through 14 solve for the x apex and deltas. Since our bi-quadratics may indeed face left or right we have four intercepts. Otherwise the logic is similar to the y curve. In figure 4.10 *tx1_lower.lower* and *tx1_lower.upper* are the t values at the intercepts at the top of the curve. This is because we subtracted them from *tx1_apex* moving them towards the top. *yx1_lower* and *xx1_lower* are the coordinates at these t values. *tx1_upper.lower* and *tx1_upper.upper* are the t intercepts at the bottom of the curve. Similarly *yx1_upper* and *xx1_upper* are the coordinates there. We have shown only the t values to reduce clutter in

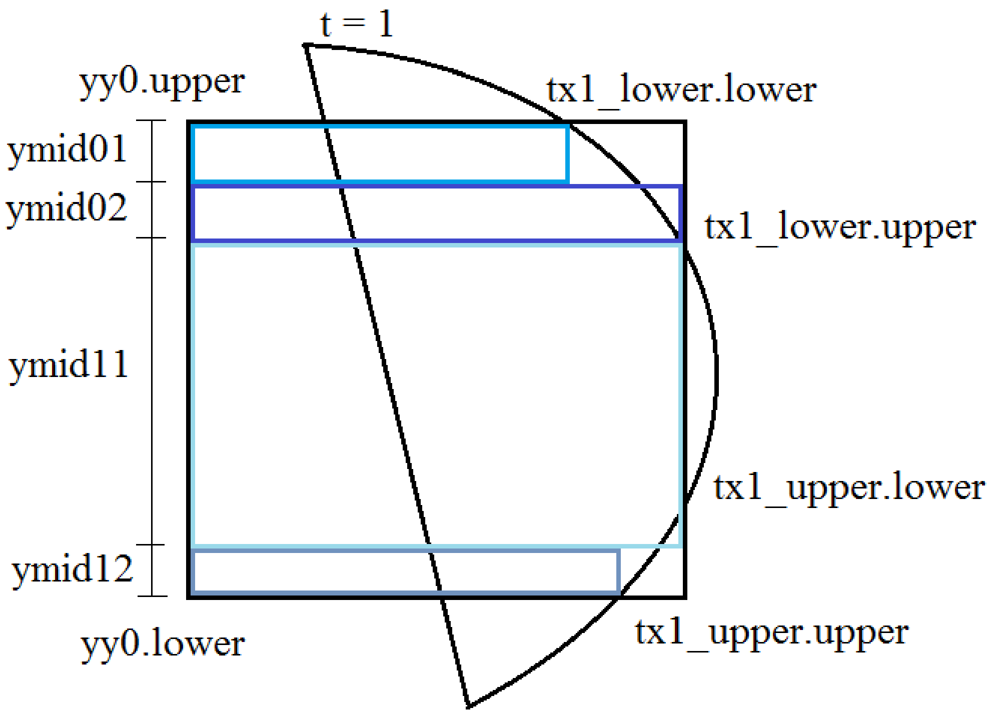


Figure 4.10: Curve reference for pseudo code walk through

figure 4.10.

The area calculations mirror those in the trapezoid clipper, this because we are approximating the curves with the trapezoid rule. While Simpsons rule could also be used for reducing error, we feel the resultant error reduction is negligible. Line 19 computes $ymid01$, the distance between the midpoint of the $yx1\_lower$ intercepts and $yy0.upper$. Line 20 computes the area of a rectangle of height $ymid01$ and width $xx1\_lower.lower$, shown in light blue. Line 21 computes $ymid02$, the distance between the midpoint of the $yx1\_lower$ intercepts and $yx1\_lower.upper$. Line 22 computes the area of a rectangle of height $ymid02$ and width $xx1\_lower.upper$, shown in dark blue. Line 27 computes $ymid11$ shown in the figure and 28 computes the area of the rectangle of height $ymid11$ and width $xx1\_upper.lower$, shown in turquoise. The purple area calculation is symmetric to the light blue. All of the areas are negative in this example.

Again, these calculations, may seem less intuitive, but are arguably the least amount of work that can be done with simple rectangular areas to compute the correct result.

Using simple rapid rectangular area calculations is our motivation for using an unorthodox sequence. They are also a bit tricky, since they must ensure correctness in all cases, and we have tested them extensively. We believe, while complex, this discussion will serve as a guide for reproduction of our highly optimized kernels.

## 4.5 IMPLEMENTATION

We briefly discuss our data structure, but call attention to the notion that our data structure was chosen to be simple to implement and reasonably efficient. For fair evaluation, we use the same structure between super sampling (SSAA) and our shapes on a plane system (SOP). More efficient data structures surely exist and are encouraged in future efforts. While these may increase performance, they should not affect the relative performance of our integration vs super sampling. This is because the data structures are used store and retrieve glyph shapes overlapping pixel foot prints, any foot print based renderer is expected to return the same shapes. The efficiency with which this is done is what varies between data structures.

As discussed we utilize the thorax true-type font rendering library by Warren Hunt for loading and pre-processing all fonts [73]. We have augmented thorax with our data structure builder. Thorax uses a custom 2D bounding volume hierarchy implementation. We have chosen to employ a two level uniform grid.

The lowest level grid is built over each glyph, and the resolution is calculated from the number of shapes in each dimension while taking into account average overlap. For example if a square glyph contains five rows of rectangles, which span an entire row each, the horizontal overlap is quite high and the horizontal resolution can thus reduced to 1. Where as the overlap in the vertical dimension is zero and thus the vertical dimension should simply be the number of shapes which in this case is five.

The upper level grid is built over the text box, and it's resolution is set based on the glyph count in each dimension, i.e. the longest sentence (horizontal) and the number of new lines vertical. Again, we do not expect to achieve optimal performance with this data structure, but since both methods of computing coverage use it, our comparisons are fair. If anything it hurts our performance comparison with Slug, which uses tuned production level data structures. However, we will show this result is also quite promising.

Our system is running on a Intel i7 7700K CPU running at 4.2GHz, and an Nvidia Titan X Maxwell GPU. We have integrated SOP and SSAA into the HVVR ray casting system from Facebook Reality Labs [75].

## 4.6 EVALUATION

We evaluate our system for quality and performance in four scenarios. Our scenes provide examples of multiple degrees of isotropic integration. We have over laid text on multiple objects within each scene, and we provide zoom images of particularly interesting sections. We hope this will provide an easier comparison of quality with respect to reference images.

All SSAA images are super sampled using point in shape tests. We utilize Loop-Blinn to sample curves, and a point in trapezoid test based on edge testing, and testing points for being above the bottom and below the top. We feel these are fair comparisons, but for clarity we have provided the pseudo code for our point in shape tests in section 4.7.

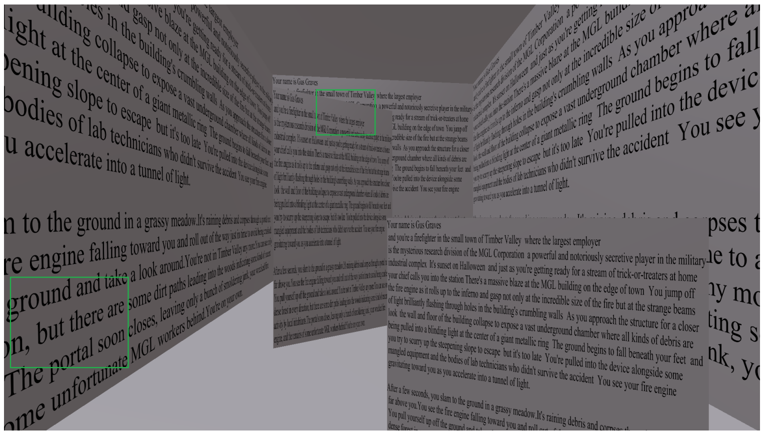


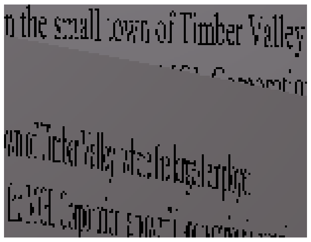
Figure 4.11: Cornell Box Scene

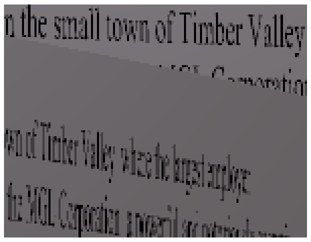
### 4.6.1 Quality and Performance

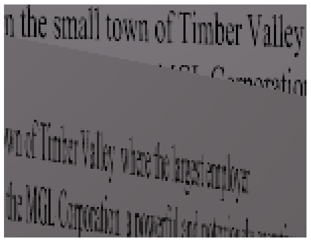
Figure 4.11 shows our first scene, We have pasted text on every vertical surface in the Cornell Box, displaying varying degrees of isotropic angles. We draw attention to the zoom images in figures 4.12 and 4.13.

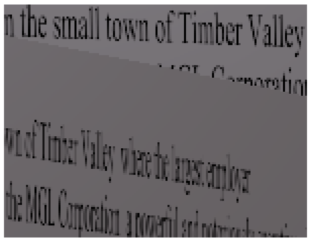
In figure 4.12 for 1x sampling (a), the sentence fragment, ”Timber Valley where the largest employer” is all but illegible. This is to be expected with no anti-aliasing. The quality improves up to (c) 16X super sampling and even to 1024 which we use a reference here, (d) and (e). We repeat the SSAAx1024 ref on the bottom row at (e) to make side by side comparison easier with SOP. In our SOP result, (f), we arguably improve readability of this fragment, even past 1024 sampling, consider the word, ”the”, in the fragment, and the phrase ”MGL Corporation”. These arguably improve from reference to SOP.

In figure 4.13 aliasing is still visible with 4x super sampling, and improves greatly with 16x. See the right side of the ”u” in the word ”ground” between 4x and 16x. One could


(a) SSAAx1


(b) SSAAx4


(c) SSAAx16


(d) SSAAx1024 REF

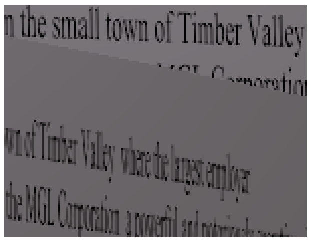
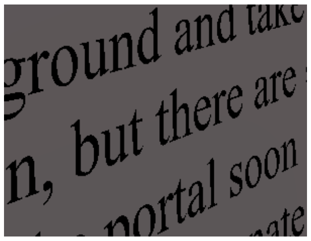
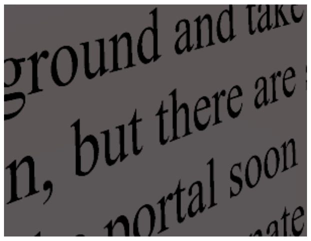
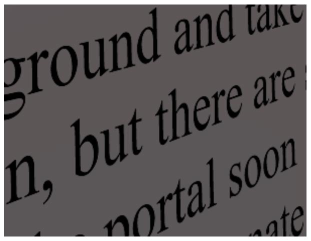
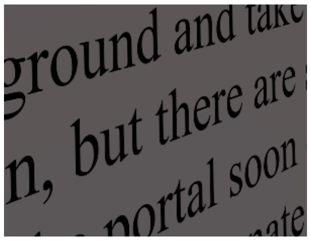

(f) SOP

Figure 4.12: Cornell box zoom. 1x, 4x, 16x, and Ref (1024x) super sampling, vs SOP integration kernels.


(a) SSAAx1


(b) SSAAx4


(c) SSAAx16


(d) SSAAx1024 REF

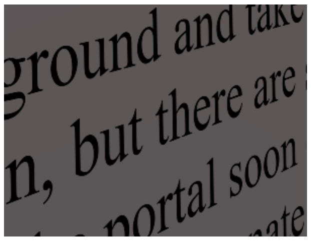

(f) SOP

Figure 4.13: Second cornell box zoom. 1x, 4x, 16x, and Ref (1024x) super sampling, vs SOP integration kernels.

argue text at this scale, is more important than the text in figure 4.12. So it would appear 16x is reasonable quality. In fact even in figure 4.12, text becomes legible with 4 samples in each dimension. In figure 4.13 however, there is a noticeable difference in quality of jaggies on rounded edges, see "b" in "but", between 16x and reference. SOP does much better.

Figure 4.14 displays our books scene. We again draw attention the zoom images in fig-

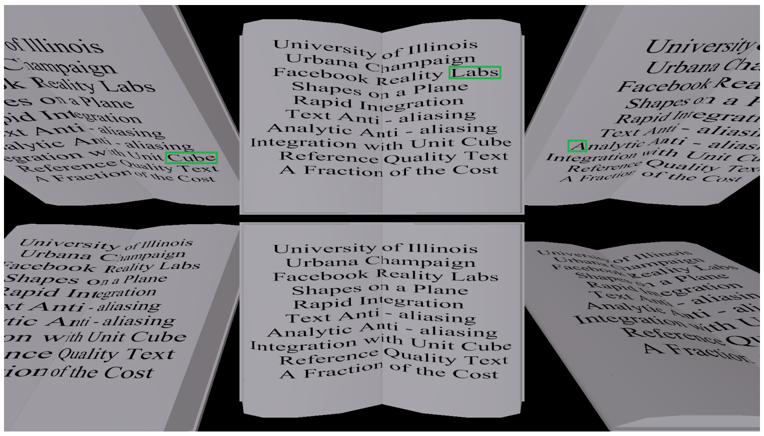
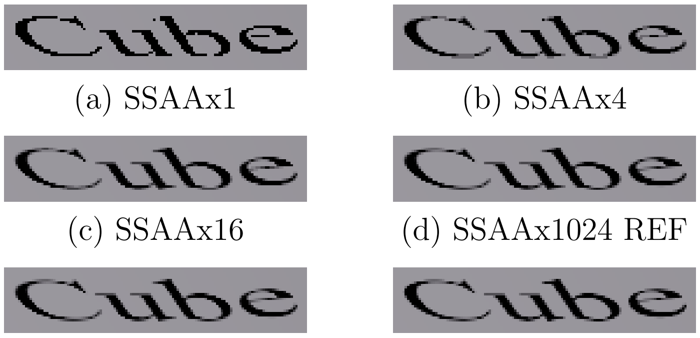


Figure 4.14: Books Scene

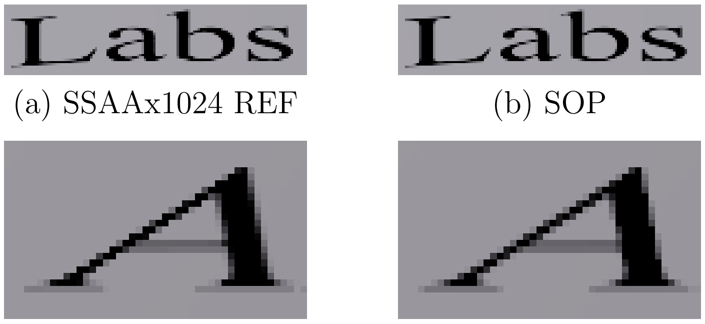


(a) SSAAx1 (b) SSAAx4

(c) SSAAx16 (d) SSAAx1024 REF

(e)SSAAx1024 REF Repeated (f) SOP

Figure 4.15: Books scene zoom.



(a) SSAAx1024 REF (b) SOP

(c) SSAAx1024 REF Repeated (d) SOP

Figure 4.16: Another books scene zoom, comparison between only SOP and Ref

ures 4.15 and 4.16. Figure 4.15 shows SSAA is actually effective at only 4x, but clearly shows aliasing on the "e" in cube. Again SOP comes closer to the ref than 16x, this can be seen by observing the "C", 16x is aliasing. Figure 4.16 shows only SOP and Ref, again they are very close.

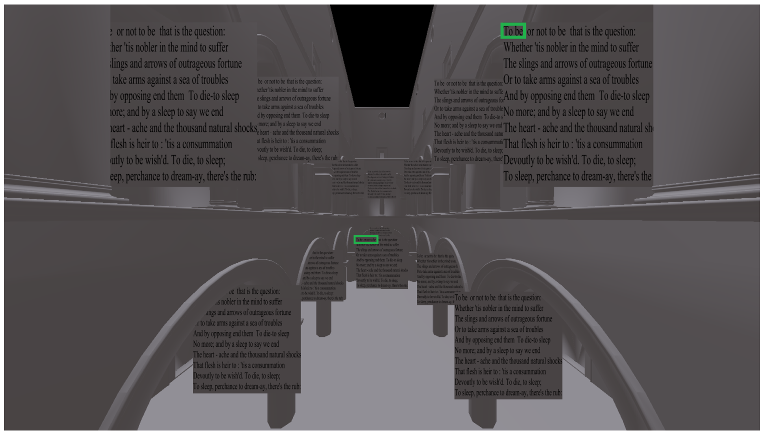
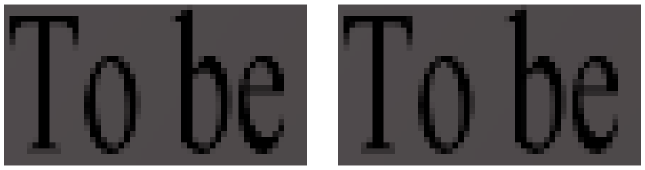
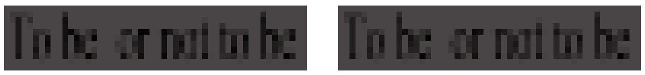

To be or not to be that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune
Or to take arms against a sea of troubles
And by opposing end them To die-to sleep
No more; and by a sleep to say we end
The heart - ache and the thousand natural shocks
That flesh is heir to : 'tis a consummation
Devoutly to be wish'd. To die, to sleep;
To sleep, perchance to dream-ay, there's the rub:

Figure 4.17: Sponza Scene



(a) SSAAx1024 REF (b) SOP



(c) SSAAx1024 REF (d) SOP

Figure 4.18: Sponza zoom, comparison between only SOP and Ref

Figure 4.17 displays our modified Sponza scene, which was mostly included for performance results on a realistic scene. It containing panels of text, with a few verses of Hamlet. The zoom sections in figure 4.18 again match the ref very closely, and are arguably pixel identical, or better looking in SOP.

### 4.6.2 Performance

Figure 4.19 shows the total frame time in milliseconds for five scenarios for each scene. We show the time taken by HVVR to render the scene without text, with text using SSAA 1x, using SSAA 4x, 16x, and our shapes on a plane implementation. Shapes on a plane is faster than or competitive with all SSAA levels, which was mostly expected, but we are sometimes even faster the 1x SSAA. This is surprising. Again refer to section 4.7 for fair comparison evaluation of our SSAA implementation. Figure 4.20 shows total GPU time which is the overhead caused by rendering text in the scene as opposed to rendering the scene for the Slug demo scene.

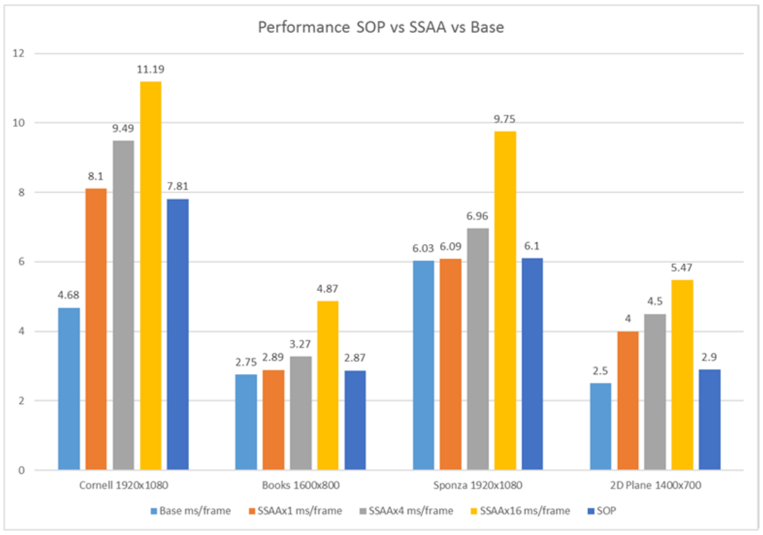


Figure 4.19: Performance of SSAA vs SOP

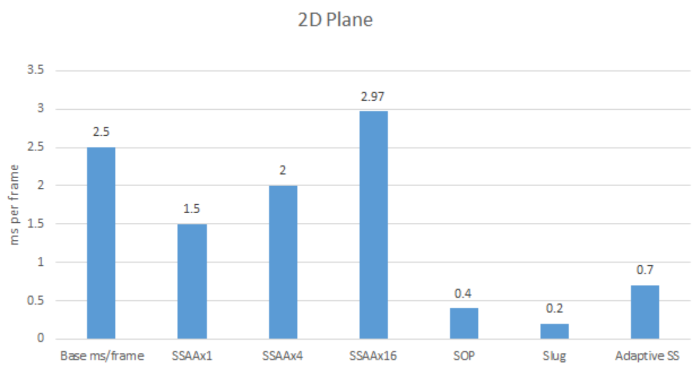


Figure 4.20: Performance of SOP vs SSAA and Slug Library

We are always ahead of SSAA, but are slightly behind the Slug library in it's default mode, as opposed to it's adaptive super sampling mode. The Slug demo contains a story's text roughly two paragraphs, and fills most of an HD window. We don't have a pure 2D mode, but we rendered a plane in 3D and matched the resolution and story text in the slug demo. For this evaluation, we simply draw it on a full screen quad. While the text is identical in size we have not included quality comparisons, since our renderer is 3D without kerning and other effects utilized by Slug to enhance text layout quality. So a qualitative comparison would be apples to oranges. Slug renders faster than us in default mode, we are within 2X, no doubt due to optimization and better layout of data structures. However, the adaptive sampling mode of Slug which is required to produce their highest quality text is behind our implementation by roughly 25%. See figure 4.20. One could assume based on our qualitative results vs SSAA at 1024 samples, that we are ahead of Slug, unless of course Slug default is better the 1024x SSAA quality. Which is not likely, based on their implementation [72]. This concludes our discussion.

```
struct QuadraticEqn{
 QuadraticEqn(A, B, C) {
   //same for an x curve
   //(y2 - y0) - 2 * (y1 - y0)
   a = float(A);
   //2 * (y1 - y0)
   b = float(B);
   //y0
   c = float(C);
   i_a = float(-1.0 / A);
   i_b = 0.5 * B * i_a;
   apex_sign = B == 0.0 || i_b < 0.0 ? -1.0 : 1.0;
   auto i_c = i_b * i_b + i_a * C;
   solve = Range(i_c, i_c - i_a);
  }
  ApplyT()(t) { return (a * t + b) * t + c; }
  Range ApplyToRange()( Range& t) {
   return Range(ApplyT()(t.lower), ApplyT()(t.upper));
  }
  SolveApex() { return i_b; }
  GetApexSign() { return apex_sign; }
  Range SolveDelta() const {
   return Range(sqrtf(max(0.0f, solve.lower)),
   sqrtf(max(0.0f, solve.upper)));
  }
  a, b, c;
  i_a, i_b;
  apex_sign;
  solve;
 };
}
```

Figure 4.21: Quadratic Equations

```
//Called one per shape
CurvePreSampling(x0, y0, v0X, v0Y, v2X, v2Y, &e0, &e1, &e2)
{
 //derive the two end points
 float x1 = x0 + v0X;
 float y1 = y0 + v0Y;
 float x2 = x0 + v2X;
 float y2 = y0 + v2Y;
 //edge 0 constants
 e0.x = (y2 - y1);
 e0.y = (x1 - x2);
 e0.z = (x2*y1 - x1*y2);
 //edge 1
 e1.x = (y0 - y2);
 e1.y = (x2 - x0);
 e1.z = (x0*y2 - x2*y0);
 //edge 2
 e2.x = (y1 - y0);
 e2.y = (x0 - x1);
 e2.z = (x1*y0 - x0*y1);
}
```

Figure 4.22: Curve sampling

```
//Called once per sample
HitCurveLoopBlinn(sampleX, sampleY, e0, e1, e2, insideCurve)
{
 //eval edge equations
 inLine0 = (w = e2.x*sampleX + e2.y*sampleY + e2.z) >= 0;
 inLine1 = (u = e0.x*sampleX + e0.y*sampleY + e0.z) >=0;
 inLine2 = (v = e1.x*sampleX + e1.y*sampleY + e1.z) >= 0;

 mag_i = 1 / (u + v + w);
 u *= mag_i;
 v *= mag_i;
 w *= mag_i;

 //interpolate texture coordinates
 s = v*0.5 + w;
 t = w;
 //calculate inverse st
 s2 = v*0.5 + u;
 t2 = u;

 insideEdges = inLine2 ? inLine2 : !inLine0 && inLine1;
 hit = insideCurve ? (s2*s2 - t2) < 0 : (s*s - t) < 0;
 none = (insideCurve && hit && insideEdges);
 one = (!insideCurve && hit && insideEdges);
 return none ? -1 : one ? 1 : 0;
}
```

Figure 4.23: Curve sampling

```
//Called once per shape
TrapPreSampling(leftBottom, leftTop, rightTop,
rightBottom, bottom, top, &l, &r)
{
 float left = max(leftBottom, leftTop);
 float right = min(rightBottom, rightTop);
 //edge constants
 l.x = (top - bottom);
 l.y = (leftBottom - leftTop);
 l.z = (leftTop*bottom - top*leftBottom);
 r.x = (bottom - top);
 r.y = (rightTop - rightBottom);
 r.z = (rightBottom*top - bottom*rightTop);
}
//Called once per sample
HitTrapOptimized(sampleX, sampleY, bottom, top, l, r)
{
 //top bottom test
 vin = sampleY >= bottom && sampleY <= top;
 //edge evals
 inr = (r.x*sampleX + r.y*sampleY + r.z > 0);
 inl = (l.x*sampleX + l.y*sampleY + l.z > 0);
 return (vin && inr && inl) ? 1.0f : 0.0f;
}
```

Figure 4.24: Trapezoid sampling

# CHAPTER 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 in VR systems. 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 analyzed 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.

We also studied the feasibility of leveraging the 2D planar map streaming and distributed rendering in cloud gaming. We first presented the server and client pipelines, based on the standalone 2D planar map rendering pipeline with additions of several components for compression and transmission. We then dived into the core challenge of the platform: the design of the compressor/decompresser of 2D planar maps, which has not been studied before. We designed a parameterized compression component, and derived the optimal parameters through real experiments. We then compared our rendering platform against the state-of-the-art x265. Our results are quite promising. Although our platform is outperformed by x265 in PSNR at low bitrate, we significantly outperform it at high bitrates. In addition, our platform outperforms x265 in terms of video quality, e.g., by up to 0.14 in SSIM. Other merits of the proposed platform include: (i) fast running time, especially at the client side and (ii) high scalability to ultra-high resolutions without bitrate penalty.

Lastly, we have proposed a technique that adopts analytic anti-aliasing, using direct pixel-primitive integrals. This is a hard problem. The novelty in our work lies in our approach. We provide new math operations that drastically simplify integration/clipping operations for any primitive. The first contribution of this work lies in a new transform for shapes on a plane, which maintains horizontal alignment of shapes while changing the integration space from arbitrary into a unit square. The second contribution is a set of shape integrators with very few instructions and no branches, thus very GPU friendly, which exemplify the simplicity achieved by using our novel transform operation.

We implemented our techniques in the HVVR GPU raycasting system for VR with a simple data structure, and still our results show competitive speed with state of the art, while achieving effectively optimal AA. We have compared with production level systems, and come out ahead in some cases, while falling slightly behind in others, no doubt due to our system structure being largely unoptimized with the exception of our fast integration kernels. This a very necessary future improvement for our system, a better data structure and data layout. Regardless, we have shown stark improvement over super sampling with respect to quality and speed. In fact our system meets closer to reference quality than 16x super sampling, for the price of taking a single point in shape sample. Shapes on a plane is fast and accurate, and we believe a large contribution to improving state of the art in primitive anti-aliasing.

In all, we have presented three contributions which form an end to end pipeline for enabling and exploiting vector images, or vector graphics, for use in real time graphics systems i.e. a "High Performance Vector Rendering Pipeline."

# REFERENCES

[1] C. A. Burns and W. A. Hunt, "The visibility buffer: A cache-friendly approach to deferred shading," *Journal of Computer Graphics Techniques (JCGT)*, vol. 2, no. 2, pp. 55–69, August 2013. [Online]. Available: http://jcgt.org/published/0002/02/04/

[2] W. Engel, "Triangle visibility buffer," 2018. [Online]. Available: http://diaryofagraphicsprogrammer.blogspot.com/2018/03/triangle-visibility-buffer.html

[3] P.-C. Wang, A. I. Ellis, J. C. Hart, and C.-H. Hsu, "Optimizing next-generation cloud gaming platforms with planar map streaming and distributed rendering," in *Proceedings of the 15th Annual Workshop on Network and Systems Support for Games*, ser. NetGames '17. Piscataway, NJ, USA: IEEE Press, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3167906.3167911 pp. 25–30.

[4] A. I. Ellis, W. Hunt, and J. C. Hart, "Svgpu: Real time 3d rendering to vector graphics formats," in *Proceedings of High Performance Graphics*, ser. HPG '16. Goslar Germany, Germany: Eurographics Association, 2016. [Online]. Available: http://dl.acm.org/citation.cfm?id=2977336.2977339 pp. 13–21.

[5] I. E. Sutherland and G. W. Hodgman, "Reentrant polygon clipping," *Commun. ACM*, vol. 17, no. 1, pp. 32–42, Jan. 1974. [Online]. Available: http://doi.acm.org/10.1145/360767.360802

[6] G. Bernstein and D. Fussell, "Fast, exact, linear booleans," in *Proceedings of the Symposium on Geometry Processing*, ser. SGP '09. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1735603.1735606 pp. 1269–1278.

[7] W. Cai, R. Shea, C. Huang, K. Chen, J. Liu, V. C. M. Leung, and C. Hsu, "A survey on cloud gaming: Future of computer games," *IEEE Access*, vol. 4, pp. 7605–7620, 2016.

[8] E. Lapidous and G. Jiao, "Optimal depth buffer for low-cost graphics hardware," *Proc. SIGGRAPH/EUROGRAPHICS Hardware Workshop*, pp. 67–73, 1999.

[9] A. Lauritzen, "Deferred rendering for current and future rendering pipelines," *SIGGRAPH Course Notes: Beyond Programmable Shading*, 2010.

[10] R. Sollefeldt, "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.

[11] L. Roberts, "Machine perception of three-dimensional solids," Lincoln Laboratory, MIT, Tech. Rep. TR 315, 1963.

[12] W. Mason, "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.

[13] B. Guenter, M. Finch, S. Drucker, D. Tan, and J. Snyder, “Foveated 3D graphics,” *Proc. SIGGRAPH Asia, ACM TOG*, vol. 31, no. 6, 2012.

[14] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, “A characterization of ten hidden-surface algorithms,” *ACM Comput. Surv.*, vol. 6, no. 1, pp. 1–55, 1974.

[15] A. Appel, “The notion of quantitative invisibility and the machine rendering of solids,” *Proc. 22nd ACM Natl. Conf.*, pp. 387–393, 1967.

[16] G. Winkenbach and D. H. Salesin, “Computer-generated pen-and-ink illustration,” *Proc. SIGGRAPH*, pp. 91–100, 1994.

[17] A. Hertzmann and D. Zorin, “Illustrating smooth surfaces,” *Proc. SIGGRAPH*, pp. 517–526, 2000.

[18] C. Geuzaine, “GL2PS: an OpenGL to PostScript printing library,” www.geuz.org/gl2ps, 2003.

[19] M. Stroila, E. Eisemann, and J. Hart, “Clip art rendering of smooth isosurfaces,” *IEEE TVCG*, vol. 14, no. 1, pp. 135–145, 2008.

[20] E. Eisemann, H. Winnemöller, J. C. Hart, and D. Salesin, “Stylized vector art from 3d models with region support,” *Proc. EGSR*, pp. 1199–1207, 2008.

[21] E. Eisemann, S. Paris, and F. Durand, “A visibility algorithm for converting 3d meshes into editable 2d vector graphics,” *ACM TOG*, vol. 28, no. 3, pp. 83:1–83:8, 2009.

[22] K. Karsch and J. C. Hart, “Snaxels on a plane,” *Proc. NPAR*, pp. 35–42, 2011.

[23] L. Markosian, M. A. Kowalski, D. Goldstein, S. J. Trychin, J. F. Hughes, and L. D. Bourdev, “Real-time nonphotorealistic rendering,” *Proc. SIGGRAPH*, pp. 415–420, 1997.

[24] J. W. Buchanan and M. C. Sousa, “The edge buffer: A data structure for easy silhouette rendering,” *Proc. NPAR*, pp. 39–42, 2000.

[25] K.-J. Kim and N. Baek, “Fast extraction of polyhedral model silhouettes from moving viewpoint on curved trajectory,” *Comput. Graph.*, vol. 29, no. 3, pp. 393–402, 2005.

[26] R. Raskar and M. Cohen, “Image precision silhouette edges,” *Proc. I3D*, pp. 135–140, 1999.

[27] R. Raskar, “Hardware support for non-photorealistic rendering,” *Proc. SIGGRAPH/Eurographics Hardware Workshop*, pp. 41–47, 2001.

[28] S. N. Ho and R. Komiya, “Real time loose and sketchy rendering in hardware,” *Proc. Spring Conference on Computer Graphics*, pp. 83–88, 2004.

[29] J. Wang, J. Sun, M. Che, Q. Zhai, and W. Nie, “Image space silhouette extraction using graphics hardware,” *Proc. ICCSA*, pp. 284–291, 2005.

[30] F. Cole and A. Finkelstein, "Fast high-quality line visibility," *Proc. I3D*, pp. 115–120, 2009.

[31] T. Auzinger, M. Wimmer, and S. Jeschke, "Analytic visibility on the gpu," *Computer Graphics Forum (Proc. Eurographics)*, vol. 32, no. 2, pp. 409–418, May 2013.

[32] S. Lefebvre and H. Hoppe, "Perfect spatial hashing," *Proc. SIGGRAPH, ACM TOG*, vol. 25, no. 3, pp. 579–588, 2006.

[33] G. Bernstein and D. Fussell, "Fast, exact, linear booleans," in *Proceedings of the Symposium on Geometry Processing*, ser. SGP '09. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2009, pp. 1269–1278.

[34] P. Ross, "Cloud computing's killer app: gaming," *IEEE Spectrum*, vol. 46, no. 3, p. 14, 2009.

[35] "PlayStation Now web page," January 2015, http://www.playstation.com/en-us/explore/playstationnow/.

[36] C. Huang, K. Chen, D. Chen, H. Hsu, and C. Hsu, "GamingAnywhere: the first open source cloud gaming system," *ACM Transactions on Multimedia Computing, Communications, and Applications*, vol. 10, no. 1s, pp. 10:1–10:25, 2014.

[37] W. Cai, R. Shea, C. Huang, K. Chen, J. Liu, V. Leung, and C. Hsu, "The future of cloud gaming," *Proceedings of the IEEE*, vol. 104, no. 4, pp. 687–691, 2016.

[38] A. Ellis, W. Hunt, and J. Hart, "Svgpu: real time 3D rendering to vector graphics formats," in *Proc. of High Performance Graphics (HPG'16)*, 2016, pp. 13–21.

[39] P. Baudelaire and M. Gangnet, "Planar maps: an interaction paradigm for graphic design," in *Proc. of the SIGCHI Conference on Human Factors in Computing Systems (CHI'89)*, 1989, pp. 313–318.

[40] P. Asente, M. Schuster, and T. Pettit, "Dynamic planar map illustration," *ACM Transactions on Graphics*, vol. 26, no. 3, p. 30, 2007.

[41] M. Hemmati, A. Javadtalab, A. Shirehjini, S. Shirmohammadi, and T. Arici, "Game as video: bit rate reduction through adaptive object encoding," in *Proc. of ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'13)*, 2013, pp. 7–12.

[42] A. Jurgelionis, P. Fechteler, P. Eisert, F. Bellotti, H. David, J. Laulajainen, R. Carmichael, V. Poulopoulos, A. L. P. Perälä, A. Glora, and C. Bouras, "Platform for distributed 3D gaming," *International Journal of Computer Games Technology*, vol. 2009, pp. 1–15, 2009.

[43] W. Cai, R. Shea, C. Huang, K. Chen, J. Liu, V. Leung, and C. Hsu, "A survey on cloud gaming: future of computer games," *IEEE Access*, vol. 4, pp. 7605–7620, 2016.

[44] D. Meiländer, F. Glinka, S. Gorlatch, L. Lin, W. Zhang, and X. Liao, "Bringing mobile online games to clouds," in *Proc. of IEEE Computer Communications Workshops (INFOCOM WKSHPS'14)*, 2014, pp. 340–345.

[45] X. Nan, X. Guo, Y. Lu, Y. He, L. Guan, S. Li, and B. Guo, "A novel cloud gaming framework using joint video and graphics streaming," in *Proc. of IEEE International Conference on Multimedia and Expo (ICME'14)*, 2014, pp. 1–6.

[46] S. Chuah, N. Cheung, and C. Yuen, "Layered coding for mobile cloud gaming using scalable blinn-phong lighting," *IEEE Transactions on Image Processing*, vol. 25, no. 7, pp. 3112–3125, 2016.

[47] G. Bernstein and D. Fussell, "Fast, exact, linear booleans," *Computer Graphics Journal*, vol. 28, no. 5, pp. 1269–1278, 2009.

[48] J. Peng, C. Kim, and C. Kuo, "Technologies for 3D mesh compression: a survey," *Journal of Visual Communication and Image Representation*, vol. 16, no. 6, pp. 688–733, 2005.

[49] S. Lloyd, "Least squares quantization in PCM," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[50] D. Arthur and S.Vassilvitskii, "k-means++: The advantages of careful seeding," in *Proc. of the ACM-SIAM Symposium on Discrete algorithms (SODA'07)*, 2007, pp. 1027–1035.

[51] M. Deering, "Geometry compression," in *Proc. of Conference on Computer Graphics and Tnteractive Techniques (SIGGRAPH'95)*, 1995, pp. 13–20.

[52] T. Cover and J. Thomas, *Elements of information theory.* John Wiley & Sons, 2012.

[53] "Objective perceptual multimedia video quality measurement in the presence of a full reference," ITU Telecommunication Standardization Sector, Standard, 2008.

[54] K. Skarseth, H. Bjørlo, P. Halvorsen, M. Riegler, and C. Griwodz, "OpenVQ: a video quality assessment toolkit," in *Proc. of ACM International Conference on Multimedia (MM'16), OSSC paper*, 2016, pp. 1197–1200.

[55] February 2017, http://x265.org.

[56] C. Green, "Improved alpha-tested magnification for vector textures and special effects," in *ACM SIGGRAPH 2007 Courses*, ser. SIGGRAPH '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1281500.1281665 pp. 9–18.

[57] N. P. Rougier and B. Esfahbod, "Digital typography: 25 years of text rendering in computer graphics," in *ACM SIGGRAPH 2018 Courses*, ser. SIGGRAPH '18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3214834.3214837 pp. 12:1–12:29.

[58] V. Chlumsky, "Shape decomposition for multi-channel distance fields," Master's Thesis, Czech Technical University in Prague, Faculty of Information Technology, 2015.

[59] Z. Qin, M. D. McCool, and C. Kaplan, "Precise vector textures for real-time 3d rendering," in *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, ser. I3D '08. New York, NY, USA: ACM, 2008. [Online]. Available: http://doi.acm.org/10.1145/1342250.1342281 pp. 199–206.

[60] B. Walter, G. Drettakis, and S. Parker, "Interactive rendering using the render cache," in *Proceedings of the 10th Eurographics Conference on Rendering*, ser. EGWR'99. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 1999. [Online]. Available: http://dx.doi.org/10.2312/EGWR/EGWR99/019-030 pp. 19–30.

[61] K. Bala, B. Walter, and D. P. Greenberg, "Combining edges and points for interactive high-quality rendering," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 631–640, July 2003. [Online]. Available: http://doi.acm.org/10.1145/882262.882318

[62] G. Ramanarayanan, K. Bala, and B. Walter, "Feature-based textures," in *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques*, ser. EGSR'04. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2004. [Online]. Available: http://dx.doi.org/10.2312/EGWR/EGSR04/265-274 pp. 265–274.

[63] E. Parilov and D. Zorin, "Real-time rendering of textures with feature curves," *ACM Trans. Graph.*, vol. 27, no. 1, pp. 3:1–3:15, 2008. [Online]. Available: http://doi.acm.org/10.1145/1330511.1330514

[64] P. Sen, M. Cammarano, and P. Hanrahan, "Shadow silhouette maps," in *ACM SIGGRAPH 2003 Papers*, ser. SIGGRAPH '03. New York, NY, USA: ACM, 2003. [Online]. Available: http://doi.acm.org/10.1145/1201775.882301 pp. 521–526.

[65] P. Sen, "Silhouette maps for improved texture magnification," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, ser. HWWS '04. New York, NY, USA: ACM, 2004. [Online]. Available: http://doi.acm.org/10.1145/1058129.1058139 pp. 65–73.

[66] A. Reshetov and D. Luebke, "Infinite resolution textures," in *Proceedings of High Performance Graphics*, ser. HPG '16. Goslar Germany, Germany: Eurographics Association, 2016. [Online]. Available: https://doi.org/10.2312/hpg.20161200 pp. 139–150.

[67] J. Tumblin and P. Choudhury, "Bixels: Picture samples with sharp embedded boundaries," in *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques*, ser. EGSR'04. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2004. [Online]. Available: http://dx.doi.org/10.2312/EGWR/EGSR04/255-264 pp. 255–264.

[68] N. Ray, T. Neiger, B. Lévy, and X. Cavin, "Vector texture maps on the gpu," 2005.

[69] M. Tarini and P. Cignoni, “Pinchmaps: Textures with customizable discontinuities,” pp. 557–568, 2005, (Eurographics 2005 Conf. Proc.). [Online]. Available: http://vcg.isti.cnr.it/Publications/2005/TC05

[70] B. Esfahbod, “Glyphy. software library,” 2012. [Online]. Available: https://github.com/behdad/glyphy

[71] W. Dobbie, “Gpu text rendering with vector textures,” 2016. [Online]. Available: http://wdobbie.com/post/gpu-text-rendering-with-vector-textures/

[72] E. Lengyel, “Gpu-centered font rendering directly from glyph outlines,” *Journal of Computer Graphics Techniques (JCGT)*, vol. 6, no. 2, pp. 31–47, June 2017. [Online]. Available: http://jcgt.org/published/0006/02/02/

[73] W. Hunt, “Thorax truetype. libary,” 2017. [Online]. Available: https://github.com/spiderofmean/thorax_truetype

[74] P. Constable and M. Jacobs, “Truetype fundamentals,” 2018. [Online]. Available: https://docs.microsoft.com/en-us/typography/opentype/spec/ttch01

[75] W. Hunt, M. Mara, and A. Nankervis, “Hierarchical visibility for virtual reality,” 2018.