ABSTRACT

MARRS, ADAM CHRISTOPHER. Real-Time GPU Accelerated Multi-View Point-Based Rendering. (Under the direction of Benjamin Watson and Christopher G. Healey.)

Research in the field of computer graphics has focused on producing realistic images by accurately simulating surface materials and the behavior of light. Since achieving photorealism requires significant computational power, visual realism and interactivity are typically adversarial goals. Dedicated graphics co-processors (GPUs) are now synonymous with innovation in real-time rendering and have fueled further advances in the simulation of light within real-time constraints. Important rendering effects that accurately model light transport often require evaluating costly multi-dimensional integrals. Approximating these integrals is achieved by dense spatial sampling, and is typically implemented with GPUs as multiple rasterizations of a scene from differing viewpoints. Producing multiple renders of complex geometry reveals a critical limitation in the design of the graphics processor: the throughput optimizations that make GPUs capable of processing millions of polygons in only milliseconds also prevent them from leveraging data coherence when synthesizing multiple views. Unlike its parallel processing of vertices and post-rasterization fragments, existing GPU architectures must render views serially and thus parallelize view rendering poorly. The full potential of GPU accelerated rendering algorithms is not realized by the existing single view design.

In this dissertation, we introduce an algorithmic solution to this problem that improves the efficiency of sample generation, increases the number of available samples, and enhances the performance-to-quality relationship of real-time multi-view effects. Unlike traditional polygonal rasterization, our novel multi-view rendering design achieves parallel execution in all stages of the rendering process. We accomplish this by: (1) transforming the multi-view rendering primitive from polygons to points dynamically at run-time, (2) performing geometric sampling tailored to multiple views, and (3) reorganizing the structure of computation to parallelize view rendering. We demonstrate the effectiveness of our approach by implementing and evaluating novel multi-view soft shadowing algorithms based on our design. These new algorithms tackle a complex visual effect that is not possible to accurately produce in real-time using existing methods. We also introduce View Independent Rasterization (VIR): a fast and flexible method to transform complex polygonal meshes into point representations suitable for rendering many views from arbitrary viewpoints. VIR is an important tool to achieve multi-view point-based rendering, as well as a useful general approach to real-time view agnostic polygonal sampling. Although we focus on algorithmic solutions to the classic rendering problem of soft shadows, we also provide suggestions to evolve future GPU architectures to better accelerate point-based rendering, multi-view rendering, and complex visual effects that are still out of reach.
DEDICATION

For John, Cortana, Gordon, Alyx, Lara, Nathan, and Elena for teaching a young man that dreams are as real as you make them.

For the 32 | neVer forgeT

Sic Parvis Magna
Adam Marrs was born and raised in Virginia, in the United States of America. He completed his undergraduate studies in Computer Science at Virginia Polytechnic Institute and State University (Virginia Tech) in 2007. Adam worked as a Senior Software Developer for digital advertising agencies before beginning his pursuit of computer graphics in 2011 as a Masters Student at North Carolina State University. Adam’s passion for computers and graphics comes from a love of computer games, digital art, and storytelling in interactive digital worlds. These interests have fueled his passion for advancing the field of rendering to provide richer, more beautiful interactive experiences. During his doctoral career, his research focused on accelerating real-time soft shadowing, reflections, multi-view rendering, and global illumination.
ACKNOWLEDGEMENTS

I thank my committee for their time, advice, and encouragement. Each of you played a crucial role during my time at NC State. I extend a special thanks to Ben and Christopher. Without you two this wouldn’t have been possible.

It takes a village. Thank you to my family, whose encouragement and enduring support makes everything possible: David, Beth, Ashley, Brian, Wei, and Cali.

I owe a special debt of gratitude to Mary, Winifrede, and Tad Devan whose wisdom, foresight, selfless generosity, and love for education changed my life.

I thank the exceptional educators who imparted concrete skills and concepts, revealed the possibilities that life holds, and most importantly inspired me to reach for more. It is impossible to understate the contributions you have made to my life: Carla Hunt, Judy Robertson, Bill Roberts, David Larrick, Teresa Tyler, Matthew Fedde, Steve Schneiter, Libby Bradford, William McQuain, Deborah Higgins, Hal Hankins, and John Garland.
# TABLE OF CONTENTS

**LIST OF TABLES** ........................................ vii

**LIST OF FIGURES** ..................................... viii

**Chapter 1 Introduction** ................................. 1

**Chapter 2 Background** ................................. 5
  2.1 Architecture of Modern Graphics Processors .......... 6
  2.2 Point-Based Rendering .................................. 10
  2.3 Rendering Soft Shadows in Real Time .................. 13
  2.4 Rendering Reflections in Real Time .................... 19

**Chapter 3 Multi-View Point-Based Rendering in Real-Time** .......... 22
  3.1 Motivation ............................................. 22
  3.1.1 Limitations of Existing Methods .................. 23
  3.2 A Multi-View Rendering Design ......................... 26
    3.2.1 Generating Points from Polygons .................. 27
    3.2.2 Parallel Image Rendering ......................... 28

**Chapter 4 View Warped Multi-View Soft Shadows for Area Lights** ....... 33
  4.1 Overview .............................................. 34
  4.2 Algorithm ............................................. 35
    4.2.1 Buffered VWSS ..................................... 36
    4.2.2 Unbuffered VWSS ................................... 39
  4.3 Results ............................................... 40
    4.3.1 Quality vs. Performance .......................... 40
    4.3.2 Performance ...................................... 43
    4.3.3 Quality .......................................... 48
  4.4 Conclusion ............................................ 51

**Chapter 5 View Independent Rasterization** ..................... 52
  5.1 View Independent Transform ............................ 53
  5.2 Multi-View Sampling Rate ................................ 54
    5.2.1 Visualizing Sampling Rate ......................... 56
    5.2.2 Perspective Distortion ............................ 58
  5.3 Sampling Projected Polygons in VIR .................... 61
    5.3.1 Sub-Pixel ......................................... 61
    5.3.2 Peri-Pixel ....................................... 62
    5.3.3 Supra-Pixel ....................................... 63
  5.4 Optimization Strategies ................................ 65
    5.4.1 Point Generation .................................. 65
    5.4.2 Point Rendering .................................. 69
LIST OF TABLES

Table 4.1 GPU performance of soft shadowing algorithms on various scenes. Multi-View algorithms use 24 depth buffers @ 120 spp (and 128 depth buffers @ 640 spp in parentheses). VWSS accelerates multi-view depth buffer rendering time $\sim 2 \times 13 \times$ (highlighted blue) and improves total rendering time up to $3 \times$ (highlighted red). VWSS matches PCSS performance (bold) while producing images with less total error (RMSE). Scenes with complex geometry benefit the most from VWSS. GPU times reported in milliseconds (ms). . . . . . . . . 45

Table 6.1 GPU performance and memory cost of VIR point generation for 128 views (and 8 views in parentheses) for geometry of varying complexity. Point storage is measured in megabytes (MB). GPU time is measured in milliseconds (ms). 74

Table 6.2 Performance results of various multi-view point rendering methods. Each method operates on 600,000 points. Fastest times highlighted in blue. Implementations in GPU Compute are faster in all cases. GPU times are measured in milliseconds (ms). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75

Table 6.3 GPU performance results of rendering soft shadows using View Independent Rasterization (VIR) coupled with our multi-view point-based rendering approach (compared to serial Multi-View Rasterization in parentheses). Both techniques compute soft shadows using 128 depth buffers of $1024^2$ resolution. The VIR sampling step is highlighted in red. Unlike MVR, VIR is able to sample the geometry once and reuse point data if geometry is not animated (see the Total Time Static column). A fully dynamic comparison of VIR and MVR recomputes the point data every frame, and is shown in the Total Time Dynamic column. VIR performance improvements are highlighted in blue, and increase significantly as geometric complexity increases. All GPU times are reported in milliseconds (ms). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1</td>
<td>A conceptual representation of modern GPU architectures [39]. <em>Left:</em> the common shader core. <em>Right:</em> the GPU Compute threading model. Thread groups are shown in orange.</td>
<td>6</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>A conceptual model of the GPU graphics pipeline [39]. Fixed function stages are shown in green and programmable stages are in blue.</td>
<td>7</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>The results of standard rasterization and overestimating conservative rasterization. For each method, pixels that produce samples for the triangle are colored grey.</td>
<td>9</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>Point-based rendering as illustrated in [29]. <em>Left:</em> Naive forward mapping of points. <em>Right:</em> Splatting distributes a point’s contribution across an area of nearby pixels.</td>
<td>11</td>
</tr>
<tr>
<td>Figure 2.5</td>
<td>Pre-production images from Media Molecule’s video game “Dreams” using a point-based renderer [24]. <em>Left:</em> high sampling density where adjacent points typically lie within the same pixel create a sharp, detailed image. <em>Right:</em> reduced point sampling density with a larger splatting radius creates a blurred, painterly image for artistic effect.</td>
<td>11</td>
</tr>
<tr>
<td>Figure 2.6</td>
<td>The pull-push algorithm from [30]. a) Original image. b) An incomplete image with gaps. c) Pull Phase: creates lower resolution approximations. d) Push Phase: uses the low resolution images to reconstruct a complete, but blurred result.</td>
<td>12</td>
</tr>
<tr>
<td>Figure 2.7</td>
<td>The Anatomy of a Shadow.</td>
<td>13</td>
</tr>
<tr>
<td>Figure 2.8</td>
<td>Shadow mapping using a point light source [86]. The closest surface’s depth is stored in a depth buffer, and compared to the depth of surfaces visible to the eye’s view [21].</td>
<td>14</td>
</tr>
<tr>
<td>Figure 2.9</td>
<td>Left: The geometry of shadows cast from an area light. Right: a large area light increases the size of the penumbra and decreases the size of the umbra [21].</td>
<td>15</td>
</tr>
<tr>
<td>Figure 2.10</td>
<td>Left: accurate shadow penumbra produced by distributed ray tracing [15]. Right-Top: soft shadow convergence after 1, 3, 7, 20, and 256 frames using temporal coherence. Right-Bottom: Temporal reprojection used to improve the speed of convergence [72].</td>
<td>16</td>
</tr>
<tr>
<td>Figure 2.11</td>
<td>Left: compared to the reference, PCSS produces soft shadows that are too large [21]. Right: Variance Shadow Mapping leaks light into the umbra even in a basic situation [2].</td>
<td>17</td>
</tr>
<tr>
<td>Figure 2.12</td>
<td>A diagram from [5] of the steps in the graphics pipeline to implement ISM splatting with points generated by GPU tessellation.</td>
<td>18</td>
</tr>
<tr>
<td>Figure 2.13</td>
<td>Left: the geometry of planar reflections and the law of reflection [2]. Right: image-based planar reflections in Slightly Mad Studio’s racing game “Project Cars”.</td>
<td>19</td>
</tr>
<tr>
<td>Figure 2.14</td>
<td>Left: the environment mapping process [2]. Right: The longitude / latitude system used in Blinn and Newell’s original environment mapping scheme [2].</td>
<td>20</td>
</tr>
</tbody>
</table>
Figure 2.15 Cubic environment mapping renders the scene to each axis-align face of a cube. Modern hardware supports sampling this custom texture using a 3-component direction vector. Right: an example of a cubic environment map from [28].

Figure 3.1 Existing hardware designs require a traversal of the input geometry, and execution of the full graphics pipeline, for each view when performing multi-view rendering of polygons.

Figure 3.2 Multi-View Rasterization of soft shadows from [37]. Top: depth buffers rendered from varying light sample positions on the area light. Bottom-Left: the area light, occluder, and receiver. Bottom-Middle: multiple hard shadows produced by only 4 depth buffers. Bottom-Right: the fully resolved soft shadow using 256 depth buffers.

Figure 3.3 A comparison of triangle rasterization implementations rendering multiple depth buffers of a 240,186 polygon character. Rasterizing multiple depth buffers in a single graphics pipeline execution is often slower than serial multi-pass methods due to memory access challenges and limitations of the Geometry Shader.

Figure 3.4 Our point-based multi-view rendering design. Input polygons are converted to point primitives at runtime and then multiple images are rendered in parallel using the points.

Figure 3.5 Images constructed in parallel using generated points in a single GPU execution.

Figure 3.6 Multi-view point rendering exists on a continuum from serial, to partially parallel, to fully parallel execution per traversal of the point data.

Figure 3.7 Serial multi-view point rendering traverses point data once for each rendered image. Top: A unique intermediate image buffer per traversal. Bottom: Employing an accumulation buffer decreases memory use and distributes shading memory bandwidth over traversals.

Figure 3.8 Partially Parallel multi-view point rendering produces multiple images for each point data traversal. Top: A 2 Pass, 4 image example. Bottom: The same a 2 Pass, 4 image example employing accumulation. Note how the number of intermediate images is reduced.

Figure 3.9 Fully parallel multi-view point rendering produces all intermediate images in a single point data traversal. Top: Any number of output images is possible with GPU Compute. Bottom: Implementation in the graphics pipeline is more complex and involves more shader stages. The Geometry Shader creates copies of the original point to render each image.
Figure 4.1  Soft shadowing of a 672,927 polygon scene with complex occlusion and dynamic, skinned geometry at 1080p resolution. Our new VWSS algorithm (middle) computes depth buffers (D) 3.7× faster and the complete image (T) 2.15× faster than Multi-View Rasterization (left). Image quality is compared against a 2,560sp reference and reported using RMSE and HDR-VDP2 perceptual heat maps (bottom). VWSS achieves significantly higher numerical and perceptual quality compared to PCSS (right) with a minimal increase in rendering time.  

Figure 4.2  The data flow of buffered VWSS point generation and depth buffer construction. In Step 2, each compute (or pixel) shader thread warps a point to multiple depth buffers simultaneously, generating several depth buffers in a single traversal of the point data.  

Figure 4.3  Traditional image warping (top) incorrectly leaks light at areas of disocclusion (b) and shadow boundaries (a,c). These artifacts are eliminated by using VWSS (bottom).  

Figure 4.4  The data flow of unbuffered VWSS. Point generation and depth buffer construction steps are combined into a single execution of the graphics pipeline, as points are streamed from the rasterizer directly to pixel shader threads.  

Figure 4.5  The numerical image quality (RMSE) versus performance (ms) for the Warriors scene (top) and Cliffside scene (bottom) using MVR (red), VWSS (blue), and PCSS (grey). MVR and VWSS use 24, 32, 64, 96, and 128 depth buffers, with quality improving and performance declining as the number of buffers grows. MVR and VWSS reach diminishing quality returns at 96 depth buffers.  

Figure 4.6  The perceptual quality (HDR-VDP2) and performance of VWSS (left) and PCSS (right) both using 160spp for the Warriors scene (top) and Cliffside scene (bottom). Heatmaps indicate the probability differences between each image and the reference will be perceived. Red areas indicate a higher probability and blue areas indicate lower probability.  

Figure 4.7  GPU performance of shadowing algorithms as geometric complexity increases using 24 depth buffers @ 120spp (top) and 128 depth buffers @ 640spp (bottom). Multi-View Rasterization (red) exhibits an expected linear rise in rendering time as geometric complexity increases. The performance of VWSS (blue) is weakly linked to scene geometry, enabling VWSS to match or exceed PCSS performance (top) and produce a large number of depth buffers in a fraction of the time it takes MVR (bottom).  

Figure 4.8  Top: GPU performance comparison of buffered and unbuffered VWSS implementations. The unbuffered VWSS implementation incurs a performance penalty due to its fully parallel approach. Bottom: performance of compute versus graphics pipelines when performing buffered VWSS depth buffer point rendering. The graphics pipeline is significantly slower than GPU compute.
Figure 4.9  Top: The thin sword is not always captured by a 32 sample PCSS blocker search and creates noticeable holes in the sword’s shadow (left). VWSS using an identical number of samples per pixel (middle) has no such failure, delivers quality closer to MVR (right), and retains rendering time similar to PCSS. Bottom: PCSS is unable to handle occluder fusion properly in some scenarios (left), while VWSS shadows are similar to MVR. ................................. 48

Figure 4.10  Shading modified to emphasize self shadowing. Top: VWSS exhibits self shadow banding similar to PCSS. Bottom: MVR shows no banding artifacts, while Normal Offset Depth Biasing applied to VWSS addresses the self shadowing and disconnected contact shadows. ............................................. 49

Figure 4.11  Top: VWSS output for the Volund scene as the area light (yellow) grows. All area lights use one light sample per two units of light area. Bottom: A close-up comparison of shadows generated by MVR (left) and VWSS (right) reveals very similar results even for large lights. ................................. 50

Figure 5.1  Left: the view plane is uniformly sampled and the plane of a polygon in the view frustum receives variable sampling under perspective projection. Right: a special case, where the view plane and the polygon have uniform sampling rates when parallel, differing only by a ratio proportional to distance (under perspective projection). .................................................. 52

Figure 5.2  View Independent Rasterization as implemented in the GPU graphics pipeline. 54

Figure 5.3  Left: the minimum point-to-polygon geometric distance $d_v$ is found and $s_{mv}$ is computed. Right: $s_{mv}$ determines the polygonal sampling rate for view independent rasterization. .................................................. 55

Figure 5.4  Sampling type visualizations. a) the original texture mapped geometry. b) the basic mode displaying a single color for ideal sampling (black), undersampling (red), and oversampling (cyan). c) an extended visualization mode for oversampling using a single hue and normalized input values. d) a second extended oversampling mode, using 7 perceptually distinct hues and an additional high contrast color to indicate high oversampling rates. ................................. 57

Figure 5.5  VIR sampling using the minimum geometric distance from a view center to each polygon. a) Parallel geometry is sampled ideally. b) Undersampling occurs as the polygon rotates and moves away from the view center. c) Severe undersampling occurs with further rotation and movement towards the view extent. ................................. 58

Figure 5.6  The geometry of perspective projection distortion. ................................. 59

Figure 5.7  VIR sampling using Equation 5.3 and the perspective distortion factor $w$. Visualization of oversampling is disabled to ease in the detection of undersampling. By applying the perspective distortion factor when computing $s_{mv}$, no undersampled (red) pixels exist. ................................. 60

Figure 5.8  Sub-pixel polygon sampling methods on the VIR raster grid. ................................. 61

Figure 5.9  Peri-pixel (left) and supra-pixel (right) polygons on the VIR raster grid. ................................. 63
Figure 5.10 Edge aliasing and undersampling of supra-pixel polygons. Top: the Multi Hue Visualization Mode. Bottom: the oversampling visualization is disabled to ease in the detection of undersampling artifacts. Left: undersampling caused by aliasing between the VIR and view-dependent raster grids. Right: undersampling is eliminated with conservative rasterization.

Figure 5.11 An illustration of our optimal perspective distortion factor approximation strategy. This low polygonal complexity / high projected area example exhibits extreme oversampling that is unusual in practice, but is ideal for illustration purposes. Left: VIR using the default perspective distortion factor. Notice the high oversampling at the center of the image. Middle: VIR using a less aggressive distortion factor. The number of points generated by VIR is reduced and oversampling is significantly decreased, but undersampling is introduced in the upper right corner of the image. Right: adapting the influence of a point to additional surrounding pixels in areas of high perspective distortion eliminates the undersampling problem while retaining a low overall amount of oversampling across the image.

Figure 5.12 An illustration of our modified VIR implementation in the graphics pipeline that generates multiple discrete point-based LoDs while still traversing the geometry only once.

Figure 5.13 An illustration of Spatial Coherence Point Sample Culling.

Figure 6.1 As with VWSS, multi-view rendering using View Independent Rasterization restructures depth buffer computation to improve point reuse and parallelization on GPUs.

Figure 6.2 A comparison of texel coverage approximations. a) standard polygon rasterization, b) point-based rendering, and c) conservative polygon rasterization.

Figure 6.3 A comparison of soft shadowing algorithms. Top: soft shadows rendered using multi-view rasterization of polygons. Bottom: VIR paired with our GPU Compute parallel view rendering approach. Both methods produce 128 high resolution depth buffers and similar high quality shadow penumbra, but VIR takes only a fraction of the time.

Figure 6.4 A quality comparison of soft shadows rendered by MVR and VIR. Top Row: the 2M polygon Lucy statue. Middle Row: the 1M polygon Buddha. Bottom Row: the 1.5M polygon San Miguel Trees. Each scene is rendered using MVR (left column) and VIR (middle column). Note the significant reduction in rendering time when using VIR. RMSE numerical quality and HDR-VDP2 perceptual quality measures are included (right column). Although HDR-VDP2 reports areas containing differences with a high probability of detection in the San Miguel Trees scene (bottom right), these differences are related to the sampling bias of the MVR reference, are not a representation of error, and are actually a beneficial quality of VIR in this test case. See the next page for a more information.
Chapter 1

Introduction

The field of computer graphics is concerned with “the pictorial synthesis of real or imaginary objects from their computer-based models” [27]. Rendering is defined as the process by which the data describing a three dimensional scene is converted into a two dimensional image [66]. Since its inception in the 1960’s, much work in the field of computer graphics has focused on rendering images of astonishing realism by accurately simulating the interaction of light with surfaces composed of nearly every kind of material. This style of rendering is referred to as photorealistic rendering [19]. As the field matured, interactivity in the form of computer games, architectural design tools, advertising, and complex simulations became possible. This form of computer graphics is called real-time rendering, and is primarily concerned with producing images at a rapid rate such that the viewer is unable to perceive individual images, and instead is immersed in an interactive activity [2]. Achieving photorealism requires significant computational power, and the constraints inherent to real-time rendering are typically adversarial to photorealism. As a result, accomplishing photorealism in real-time rendering has become a driving force behind advances in the field of computer graphics. Our research focuses on decreasing the time required to produce images that include photorealistic effects, thus increasing the achievable visual quality of real-time graphics.

Dedicated graphics co-processors, called graphics processing units (GPU), are now synonymous with innovation in the domain of real-time rendering. Over their lifetime, the capabilities of graphics processors have expanded from simple accelerated vertex transformations to incredibly fast distributed rasterization and fully programmable parallel data pipelines [51]. GPUs are painstakingly optimized to render polygonal surface representations from a single viewpoint by leveraging a high throughput design with modest memory caching structures. For the first time, it is possible to process millions of polygons and create images of exceptional geometric complexity in only milliseconds [23]. The rapid adoption and near ubiquity of GPU processors motivated further advances in accurately simulating light in real-time applications [32, 37, 59].
Important visual effects that demand accurate models of light transport, such as soft shadows and reflections, are commonly evaluated using multi-dimensional integrals [41]. Solving these integrals in real-time, given the existing computational constraints, is not currently possible. Instead, sampling techniques are employed to approximate a subset of the full light transport equation. Sampling techniques are commonly implemented in real-time with GPUs as multiple rasterizations of a scene from differing viewpoints. Algorithms employing this approach are commonly referred to as *multi-pass* or *multi-view* [19]. Unfortunately, achieving noise-free results using sample-based approximations requires a large number of samples (or images from unique viewpoints) [49]. Dense sampling of this kind is prohibitively costly for real-time applications, since producing multiple renders of complex geometry reveals a critical limitation in the architecture of existing graphics processors: the throughput optimizations that make GPUs capable of processing millions of polygons also prevent it from leveraging data coherence when synthesizing multiple views. Unlike its parallel processing of vertices and post-rasterization fragments, existing GPU architectures must render views serially and thus parallelize view rendering poorly. The highly detailed and complex input geometry must be re-traversed for every view, missing a critical opportunity to reuse coherent polygonal information and post-rasterization data for similar views. Computing advanced multi-view lighting effects is faster now than before GPUs arrived, but the existing single view design does not realize the full potential of GPUs for accelerated rendering. In this dissertation, we tackle this problem from an algorithmic perspective in order to improve the efficiency of sample generation, increase the number of available samples, and enhance the visual quality of multi-view effects in real-time.

We introduce a GPU accelerated multi-view rendering design that avoids the limitations of GPU-based polygonal rasterization when producing multiple views of a scene. Inspired by the shrinking number of pixels covered by the average polygon, and thus the decreasing importance of polygonal connectivity information [29], we achieve a more flexible rendering design by instead using points as the primary multi-view rendering primitive. We improve upon the GPU’s existing serial design by: (1) transforming polygonal surfaces to a point-based representation suitable for rendering multiple views, (2) performing this conversion to points dynamically in real-time using fixed function GPU hardware, and (3) utilizing points to reorganize the structure of computation and enable the efficient parallel rendering of many images. Every stage of our proposed approach executes in a data parallel fashion, such that image generation is now treated like the processing of vertices or fragments in typical polygonal rendering.

In Chapters 4, 5, and 6 we demonstrate the algorithmic flexibility, real-time performance, and visual quality of our rendering design through the implementation and evaluation of novel algorithms based on our multi-view approach. The *View Warped Soft Shadows* (VWSS) algorithm addresses the serial rendering bottleneck when computing soft shadows cast by dynamic occluders from area light sources. VWSS establishes a useful new point on the qual-
ity/performance continuum by: (1) simplifying the multi-view soft shadowing problem using a view-dependent sampling strategy, (2) avoiding common artifacts by tailoring points to many nearby views, and (3) significantly increasing the efficiency of rendering depth buffers by reorganizing the rendering workload for coherence. VWSS renders depth buffers up to $13 \times$ faster and computes soft shadows over $3 \times$ faster than the current state of the art for accurate shadows, on typical real-time dynamic occluder geometry. VWSS exhibits similar performance and less error than existing approximate algorithms of choice, without the serious failure case artifacts that plague single-view approximations. Both the Root Mean Squared Error (RMSE) numerical comparison measure and the HDR-VDP2 [53] perceptual image comparison measure rate VWSS’s output as significantly closer to the MVR reference than PCSS.

For lighting effects with less meaningful spatial locality and coherence between views, we introduce View Independent Rasterization (VIR): a fast and flexible method to transform complex polygonal meshes into point representations suitable for rendering many views from arbitrary viewpoints. By leveraging the decades of optimizations made to fixed function GPU rasterization hardware, while also retaining the flexibility to customize polygon sampling with programmable shaders, VIR presents a new approach to GPU-based view-independent point generation from polygons suitable for real-time applications. We detail how to implement VIR in support of multi-view point-based rendering, compute an appropriate multi-view sampling rate, and mitigate challenging edge cases. We also outline a number of optimization strategies to further improve performance for future practical applications. In our tests, VIR processes two million polygons and generates nearly 600,000 points tailored for 128 views in only 5.48 milliseconds. We accelerate the rendering of depth buffers for soft shadows using VIR, and evaluate it using the same methodology as VWSS. The transformation of complex, two million polygon geometry into points and the rendering of 128 high resolution depth buffers is accelerated up to $8 \times$ compared to serial MVR. The visual quality of VIR soft shadows is excellent, and even more stable than VWSS due to the view independent point cloud that VIR generates.

Given the promising results of VWSS, VIR, and multi-view point-based rendering on the GPU, we also discuss the application of our algorithms and rendering design beyond shadows, including visual effects such as reflections, defocus blur, and diffuse global illumination. Finally, we discuss the limitations of existing GPU hardware architectures and suggest modifications to evolve them to better support the acceleration of point rendering, the parallelization of image rendering, and ultimately improve visual effects approximated by multi-view sampling algorithms. In summary, the main contributions of this dissertation include:

1. An evaluation of the limitations of the current state of the art in GPU-based multi-view rendering using polygonal and point-based rendering primitives.

2. A novel multi-view GPU rendering design that parallelizes and accelerates the rendering
of complex visual effects approximated by image-based sampling algorithms.

3. View Warped Soft Shadows: a novel real-time GPU rendering algorithm, based on our multi-view design, that uses a view-dependent point set, accelerates depth buffer generation, casts high quality soft shadows of dynamic geometry from area light sources, and represents a new practical option for applications rendering soft shadows in real-time.

4. View Independent Rasterization: a novel, flexible, real-time GPU-based sampling algorithm capable of generating view independent point data sets from polygonal geometry appropriate for an arbitrary number of unique viewpoints.

5. A novel GPU accelerated algorithm leveraging VIR that generates multiple point-based levels of detail (LoD) of polygonal geometry in a single traversal of the source geometry.

6. Detailed performance analyses and visual quality comparisons, using numerical and perceptual measures, of our algorithms versus the current state of the art.

7. Point generation and rendering optimization strategies designed for optimal multi-view rendering on existing GPU architectures.

8. Suggestions for changes to existing GPU hardware architectures to better support our new rendering design, improve point rendering, prioritize multi-view rendering, and further accelerate the rendering of complex visual effects.
Chapter 2

Background

This chapter provides a brief review of background knowledge and common terminology related to modern graphics hardware architecture, point-based rendering, and the real-time rendering of soft shadows and reflections. Since each of these topics has a rich history, we focus specifically on the subset of terminology and research that is directly relevant to or inspires our work.

Since the introduction of the first Graphics Processing Unit (GPU) in 1999, the capabilities of dedicated graphics hardware has expanded significantly [62]. Fueled by fierce competition between hardware vendors, the demand for higher graphical complexity in a maturing real-time video games market, and the addition of General Purpose GPU Computation (GPGPU or GPU Compute) to hardware architectures, the newfound flexibility of modern GPUs simultaneously improved the potential for new algorithmic approaches to classic problems and escalated the nuance-laden difficulty of unlocking that potential as improved performance. We review the common concepts and design elements of modern GPU architectures to provide a universal language and context for reasoning about algorithmic performance on modern hardware platforms.

Point-Based Rendering challenges the universally accepted use of polygons as the primary rendering primitive by instead modeling objects as a collection of infinitesimal three-dimensional points [50]. The use of points as the primary rendering primitive has been demonstrated in offline systems as a viable strategy to accelerate the rendering of complex effects including soft shadows and reflections [29]. Leveraging points as a more efficient surface representation serves as a foundational concept in our approach to accelerate challenging real-time workloads. We review concepts and terminology, discuss common challenges, and identify relevant open problems associated with point rendering.

The accurate simulation of light accounts for photons arriving directly from light sources, as well as from indirect sources, and requires the evaluation of complex multi-dimensional integrals [41]. Today, due to decades of investment in graphics hardware designed primarily to simulate illumination from direct light sources only, sampling techniques that serially rasterize...
Figure 2.1: A conceptual representation of modern GPU architectures [39]. Left: the common shader core. Right: the GPU Compute threading model. Thread groups are shown in orange.

Multiple views are still commonly employed to approximate these integrals for challenging visual effects [28, 32, 33]. Soft shadows and reflections are two such effects, where each serial render increases the effective scene complexity and decreases data coherence. As a result, these effects are ideal test cases for our multi-view point-based rendering approach. We review concepts and terminology associated with existing accurate and approximate approaches for rendering soft shadows and reflections.

2.1 Architecture of Modern Graphics Processors

The design and implementation of our work is informed by the existing capabilities of modern graphics processors. Conceptually, a GPU can be understood in two ways: (1) as an array of small processors that manipulate large amounts of similar data in parallel; or (2) as a sequential set of pipelined stages, each executing in parallel, responsible for specialized predetermined tasks when manipulating polygonal data to render images.

Recently, GPU vendors introduced the ability to execute user authored programs, known as kernels or shaders, on the GPU’s array of processors. Shader programs are built upon a common shader core, which defines a baseline of functionality available to all shaders [39]. Illustrated on the left of Figure 2.1, this common design allows for many shader programs of widely varying inputs, outputs, and purposes to execute in parallel on a set of identical hardware cores. This flexible design makes the GPU an ideal, scalable parallel processing environment. Large workloads are executed using the threading model depicted on the right of Figure 2.1. Shown in blue, individual units of work are assigned to threads. A thread is executed using a
Figure 2.2: A conceptual model of the GPU graphics pipeline [39]. Fixed function stages are shown in green and programmable stages are in blue.

single GPU processor. Shown in orange, threads are organized into conceptual groups. Thread groups are then sent to the GPU, scheduled, and executed in vendor defined quantities (32-thread warps on NVIDIA and 64-thread wavefronts on AMD). User authored programs that execute using this threading model are typically referred to as Compute shaders.

The Programmable Graphics Pipeline

Figure 5.2 depicts the stages of the graphics pipeline, each of which receives input data from the previous stage, performs specialized processing, and outputs result data to the next stage. The flexibility introduced by the common shader core also improved the existing graphics pipeline. There are now two types of graphics pipeline stages. Fixed function stages perform specific tasks using purpose-built hardware units, but can only be configured through predetermined parameters. The rasterizer is an example of a fixed function stage. Programmable stages perform a wider range of tasks by executing user authored shader programs based on the common shader core. By replacing previously fixed-function stages (Vertex, Pixel) with programmable shader stages, as well as adding entirely new stages (Tessellation, Geometry), modern architectures avoid performance bottlenecks by adapting available processing resources to varying workload demands on the fly. Currently, there are five programmable graphics pipeline stages. Each stage is responsible for a unique type of parallel processing.

The vertex shader performs processing of geometric vertices. Many vertex shader programs are executed in parallel, and each vertex shader processes a single vertex. The hull shader is an optional stage located immediately after the vertex shader. The hull shader is the entry point and first programmable stage of the tessellation pipeline. The primary responsibility of the hull shader is to process specialized vertices received from the vertex shader, known as control points, and produce tessellation factors. Control points define an input primitive, or patch, to be tessellated. Tessellation factors specify how an input patch is to be subdivided. The hull shader is composed of two functions: (1) the control point function; and (2) the patch constant function. The control point function is executed for each input control point, similar to a vertex shader.
Unlike the vertex shader, however, the control point function can create up to a maximum of 32 new control points per input patch. The patch constant function is executed once per input patch and produces the tessellation factors. The fixed function tessellator accepts the tessellation factors, subdivides the input patch, and outputs a set of weights representing the post-tessellation structure of the input patch. These weights are tailored to the input topology; therefore, a set of barycentric coordinates are produced for triangular patches. The domain shader is the final stage of the tessellation pipeline and is responsible for processing post-tessellation vertices output by the tessellator. The domain shader operates as a post-tessellation vertex shader, but unlike the vertex shader it has access to all control points and patch constant information generated by the hull shader.

The last programmable stage before rasterization is the geometry shader stage. In this optional stage, vertex information for the entire primitive, as well as adjacent primitives, is available. The geometry shader includes unique functionality for manipulating primitives of varying topology, the ability to create new geometry of arbitrary topology on the fly, the ability to store (or stream out) vertex data pre-rasterization, and the ability to designate primitives be rasterized to multiple different target images. Creating new geometry in the graphics pipeline is referred to as amplification. The geometry shader guarantees the output order of primitives matches the input order, which can severely affect performance [2].

The final programmable stage of the graphics pipeline is the pixel shader stage. The pixel shader stage, whose name is somewhat of a misnomer, processes polygon fragments produced by the rasterizer. Lighting, shading, and post-processing effects are commonly implemented in pixel shader programs. Importantly, the pixel shader stage includes extended functionality available to compute shaders, but not available to other graphics pipeline stages. This enables pixel shaders to perform a wider set of advanced operations at the end of the graphics pipeline, including image processing algorithms such as Screen Space Ambient Occlusion (SSAO) [59] and approximate motion blur effects using velocity buffers [2].

Conservative Rasterization

Rasterization is the process of converting geometric primitives into the discrete pixels that form an image [27]. The newest GPU architectures include fixed-function support for several rasterization algorithms. Shown on the left of Figure 2.3, standard rasterization produces a sample (and invokes a Pixel shader) for any part of a polygon that covers the centroid of a pixel on the raster grid. For some applications, this behavior does not provide the necessary representation of the input geometry post-rasterization. Conservative Rasterization refers to two methods of either overestimating or underestimating the pixel area covered by a polygon [1]. Overestimating the footprint of a polygon produces a sample whenever the polygon intersects any part of
Figure 2.3: The results of standard rasterization and overestimating conservative rasterization. For each method, pixels that produce samples for the triangle are colored grey.

a pixel, while underestimating requires the pixel’s entire area be contained within a polygon. The overestimating method is implied when the specific type of conservative rasterization is not specified. Overestimating conservative rasterization is shown on the right side of Figure 2.3.

Memory Structures

Modern GPU architectures include several memory structures for data storage. Two dimensional arrays used for storing image data are referred to as *textures*. The individual pixels of textures are known as *texels*. Textures can be designated as input to any programmable stage. Textures written by the graphics pipeline are referred to as *render targets*. To avoid undefined behavior from memory contention, a texture cannot simultaneously be a shader input and a render target. One dimensional arrays used for storing data of arbitrary format are called *buffers*. *Append / Consume buffers* are a stack-like buffer storage variation that maintains an internal atomic counter enabling optimized parallel insertion and deletion operations [39]. Append / Consume buffers were initially used for GPU-based particle systems, but are also useful when performing point-based rendering.

The addition of GPU Compute to hardware architectures introduced read-modify-write support for many memory resource types, including textures and buffers. Race conditions caused by concurrent threads accessing the same memory resource are resolved using *atomic operations*, which serialize memory access and perform uninterruptible operations. Atomic operations that manipulate 32-bit floating point values are not yet widely supported among existing GPU architectures, but this is expected to change in the near future.
2.2 Point-Based Rendering

*Point-Based Rendering* introduces an alternative to the ubiquitous use of polygons as the primary rendering primitive by modeling three-dimensional objects as a collection of infinitesimal three-dimensional points. The motivation for this change is based on the insight that a polygonal surface representation is not the most efficient when a polygon covers only a single pixel of an output image [50]. Understanding the design of rendering algorithms further reveals the relationship which makes point-based rendering useful.

Rendering algorithms are broadly classified into two categories based on how object visibility is determined: (1) image-order and (2) object-order [29]. *Image-order* algorithms use a *gather* approach where each pixel of an image searches the scene for the closest intersecting geometric primitive. Ray tracing is an example of an image-order algorithm [85]. *Object-order* algorithms use a *scatter* strategy by examining each geometric primitive and determining which pixels are intersected. Rasterization and the Z-buffer are common object-order algorithms [12]. The efficiency of image-order and object-order algorithms is determined by the quantity of work and the coherence of data within each step of the algorithm. Historically, computer generated images have consisted of many more pixels than geometric primitives. In this scenario, each geometric primitive covers many pixels and exhibits high coherence during object-order traversal. As a result, object-order algorithms have received dedicated hardware acceleration and significant optimization efforts.

The object-order optimization of GPU architectures resulted in extremely high polygon throughput, and spurred a rapid increase in the number of geometric primitives used when rendering images. In the past decade, the geometric complexity of interactive applications has increased to match or even exceed the number of output pixels [23]. Compounding this increase in complexity, the demand for more advanced lighting effects, such as soft shadows or reflections, in real-time applications became higher than ever. Critically, these effects require more information than what is directly available in any single viewpoint. Existing GPU architectures implement distributed rasterization algorithms to sample geometric primitives for a single view at a time. These samples are called *view-dependent*. As a result, sampling scene geometry from multiple views or directions using the GPU’s rasterizer must be performed serially. This causes multiple traversals of the complex geometry, a decrease in the coherence of the object-order algorithm, and a linear increase in the GPU’s triangle setup and rasterization tasks.

To more efficiently represent surfaces and render multi-view effects, the sampling of primitives should be agnostic of view, or *view-independent*. Compared to the view-dependent image-based approach GPUs currently employ, using points to store view-independent information is far less redundant [30]. The accuracy of a set of points used to approximate a given surface can be determined by the geodesic distance between the points [29]. Since reducing the error
Figure 2.4: Point-based rendering as illustrated in [29]. Left: Naive forward mapping of points. Right: Splatting distributes a point’s contribution across an area of nearby pixels.

by a factor of two requires four times more points [29], more efficient representations have been proposed which approximate the surface using a small circular or elliptical area around the point sample [65]. To render an image, the point sample representation is forward mapped to the raster grid in a process known as splatting [91, 83]. This is illustrated in Figure 2.4. Since each point contains all required rendering information, operations performed by GPU hardware to interpolate values across polygons, such as clipping, scan conversion, and texture mapping, are no longer necessary. A view-independent point-based representation also has an automatic level of detail (LOD), since a surface is more coarsely represented by simply reducing the number of points [50]. Recently, game developers have presented experimental fully programmable point-based graphics systems for interactive games [24]. Figure 2.5 shows pre-production results from Media Molecule’s Playstation© 4 title “Dreams”. Various artistic effects are achieved by varying the point sampling density and the pixel area of a point’s influence during splatting.

Figure 2.5: Pre-production images from Media Molecule’s video game “Dreams” using a point-based renderer [24]. Left: high sampling density where adjacent points typically lie within the same pixel create a sharp, detailed image. Right: reduced point sampling density with a larger splatting radius creates a blurred, painterly image for artistic effect.
Challenges and Open Problems

The potential efficiency improvements and simplicity of a point-based representation are not without drawbacks. Since points no longer store polygonal connectivity information, it is possible for point rendered images to lack complete information about surfaces when point data is not sufficiently dense for the target image’s resolution [29]. Illustrated by Figure 2.6b, this causes visible “holes” where some data for a surface is not present in the image. The hole-free representation of a surface for an arbitrary view is referred to as \textit{watertight} rendering. Achieving watertight rendering is the primary challenge associated with point-based rendering. Although it has been shown that the watertight sampling density of a surface can be computed by using the determinant of the Jacobian of a unit-sized patch transformed to screen space [50], this approach is still challenging to accomplish efficiently on existing GPU architectures.

Until now, there have been two main approaches to achieving watertight point-based rendering on GPUs: (1) process highly dense, oversampled point sets, or (2) augment low density point sets with smarter splatting logic and hole-filling post processes [55]. Historically, limited storage and memory bandwidth have made high density point solutions infeasible for practical real-time use. Consequently, much of the existing research in point rendering focuses on using low density point sets rendered using various techniques, such as Elliptical Weighted Average (EWA) filtering [91], that construct a resampling kernel during splatting [45]. A common post-process hole-filling algorithm is referred to as \textit{pull-push} [30]. This approach produces lower resolution approximations of the final image and then uses those approximations to fill gaps.

Finally, a lingering problem impeding point-based rendering is the limited performance and availability of read-modify-write operations exposed to programmable shaders. As with past attempts to accelerate point-based rendering [8], fundamental architectural changes are still required to optimize for points [84]. Despite the GPU’s rapid evolution, the potential for significant performance gains from dedicated silicon designed for points is still on the table.
2.3 Rendering Soft Shadows in Real Time

Shadows are perceptually important elements of visual scenes [44]. Visual cues encoded in shadows were identified as early as Renaissance painters and studied more recently by cognitive psychologists. Shadows provide static and dynamic visual clues about the location, shape, and arrangement of objects in an environment, as well as various characteristics about light sources [52]. The shadowing effect is created by objects preventing light energy, or photons, from reaching other objects in an environment.

The anatomy of a shadow is illustrated in Figure 2.7. Objects that block photons are referred to as *occluders* or *shadow casters*. Objects on which shadows are cast are referred to as *receivers* [22]. It is common for an object to simultaneously be both an occluder and a receiver. The shadow created by an occluder consists of a fully unlit region, the *umbra*, as well as a partially lit region, the *penumbra*. The penumbra is a gradient from completely unlit to lit, starting at the edge of the umbra and extending outward [20]. The penumbra’s shape is produced by a combination of several factors including: (1) occluder shape; (2) occluder proximity to the receiver; (3) light source shape; and (4) light source proximity to the occluder. An important visual effect of the penumbra is called *contact hardening* [22]. As an occluder gets closer to a receiver the shadow penumbra width decreases in size. This relationship causes the shadow to appear “harder” as the occluder gets closer to making contact with a receiver.

![Figure 2.7: The Anatomy of a Shadow.](image-url)
Point Lights

Early attempts at rendering shadows in real time modeled light sources as infinitesimal points, known as point lights. Photons emitted from point lights are modeled as rays traveling unimpeded from the point light source to the closest occluding surface. Image-based implementations of point light shadows store light rays as depth values in a discrete two dimensional array, called a depth buffer or depth map. Shadows are then computed by comparing the distance to a point in the eye’s view against the distance stored in the depth buffer for that point. This process is commonly known as shadow mapping [86] and is shown in Figure 2.8. When using a point light source, the shadow computation ignores the possibility of penumbra. This is referred to as hard shadowing due to the binary shadows it produces; where areas are either “in” shadow or “out” of shadow. Often the sampling rate of the depth buffer and eye’s view do not match. This difference is known as shadow aliasing and manifests visually as sharp, often jagged, edges such as those shown in Figure 2.8. Recent advanced algorithms have resolved a majority of the aliasing artifacts which plague hard shadowing approaches [48] [89]; however, these methods are computationally expensive and ultimately still lack shadow penumbra.

Figure 2.8: Shadow mapping using a point light source [86]. The closest surface’s depth is stored in a depth buffer, and compared to the depth of surfaces visible to the eye’s view [21].
Area Lights

Nearly all light sources in the physical world emit photons across some area in three-dimensional space. As a result, the point light source used in hard shadowing algorithms is not sufficient to produce accurate shadows [22]. More accurate algorithms model light sources as planes with non-zero area, called area lights. The geometric relationship of shadows cast from area light sources is illustrated in Figure 2.9. While still an approximation, shadows computed from area lights exhibit convincing shadow penumbra. Due to the soft edges exhibited by the penumbra, this type of shadow rendering is often referred to as soft shadowing.

![Figure 2.9: Left: The geometry of shadows cast from an area light. Right: a large area light increases the size of the penumbra and decreases the size of the umbra [21].](image)

Accurate Methods

The primary task of soft shadowing algorithms is to determine the proportion of an area light visible to a receiving surface. This proportion is known as the visibility factor [22]. Despite the massive increase in parallel processing capability over the past decade, this area visibility problem remains difficult due to occluder fusion: the non-trivial interaction of multiple occluding surfaces preventing the individual processing and combination of occlusion data [20].

Accurate soft shadowing methods solve for the visibility factor using either distributed ray tracing [15] or by accumulating a large number of rasterized depth buffers from multiple viewpoints on the area light [32, 37]. Shown in Figure 2.10, these methods converge to the correct solution given a sufficiently dense sampling rate. Multi-View Rasterization (MVR) is the best proxy for a ground truth solution that real-time applications can consider, but its
Figure 2.10: Left: accurate shadow penumbra produced by distributed ray tracing [15]. Right-Top: soft shadow convergence after 1, 3, 7, 20, and 256 frames using temporal coherence. Right-Bottom: Temporal reprojection used to improve the speed of convergence [72].

requirement of multiple traversals of occluder geometry still prevents real-time performance in most scenarios. To achieve acceptable quality with rasterization, anywhere from 256 to 1,024 depth buffers may be required to achieve convergence [22].

Temporal Methods

Another category of accurate shadowing algorithms is built upon the insight that rendered frames are often very similar over an interval of time. Algorithms leveraging *temporal coherence* reuse data from previous frames to reduce calculations for subsequent frames [74]. This reduction in per frame computation comes at the cost of requiring multiple frames to resolve the correct solution. Iteratively refined hard shadowing methods leverage previously calculated screen-space occlusion information by reprojecting results of previous frames into the current frame using motion vectors [71]. Convergence to a high quality solution can be faster than the human eye is able to perceive. Shown in Figure 2.10, a temporal approach for soft shadowing distributes multi-view scene traversal and rasterization tasks over multiple frames and uses spatial filtering to hide low coherence situations [72]. Temporal soft shadowing methods regularly achieve real-time performance, but image quality is tightly coupled to frame-to-frame similarity. Modern interactive games typically present a worst case frame-to-frame coherence scenario caused by a large number of moving objects, lights, an unpredictable camera, and a relatively large amount of time between frames (up to 33 milliseconds). Consequently, image quality is limited to an unrefined solution the majority of the time. Using approximate soft shadowing algorithms, such as *Percentage Closer Soft Shadows*, as an initial solution before temporal refinement has also been proposed [75].
Figure 2.11: Left: compared to the reference, PCSS produces soft shadows that are too large [21]. Right: Variance Shadow Mapping leaks light into the umbra even in a basic situation [2].

**Approximate Methods**

Most existing real-time applications abandon accurate methods for approximations that produce flawed but plausible results. The fastest approximate soft shadowing algorithms accept a single depth buffer as input, and even work well in specific cases, but often fail in other common situations due to the complex behavior of the penumbra. For this reason, no single approximate method is best for all scenarios.

Early soft shadowing approximations stemmed from attempts to anti-alias the boundary of a hard shadow. *Percentage Closer Filtering (PCF)* is commonly used to filter depth buffers by comparing depth texel values with the depth of the surface being rendered [67]. Statistics based methods, such as Variance [18], Convolution [3], and Exponential Shadow Mapping [4] apply various mathematical techniques that alter the values stored in depth buffers to achieve the goal of representing a range of values. Shown on the right of Figure 2.11, statistics based filtering methods erroneously leak light into the shadow umbra, have fixed size penumbra, and do not reproduce contact hardening effects.

*Adaptive filtering* techniques vary the size of a filtering kernel to approximate the contact hardening effect [26, 17, 60]. Percentage Closer Soft Shadows (PCSS) and methods based on it, such as Variance Soft Shadows [17] and Moment Soft Shadow Mapping [63], compute an average depth for nearby occluders and use the parallel planes equation to estimate filter kernel size. This approximation prevents these algorithms from correctly handling occluder fusion for occluders at different depths and makes major assumptions about the occluding geometry that are easily invalidated. Shown in the center of Figure 2.11, PCSS can produce incorrect results even in simple situations [22].
Imperfect Shadow Mapping

*Imperfect Shadow Mapping (ISM)* approximates visibility for *indirect* illumination using many low resolution depth buffers, generated by splatting an approximate point-based representation of the scene geometry [68]. Due to the low frequency characteristics of indirect illumination, the errors caused by ISM’s low resolution, approximate visibility data are largely unnoticeable.

ISM generates its *view-independent* point representation either offline or at run-time using GPU tessellation hardware [5]. Figure 2.12 illustrates the steps to accomplish ISM splatting with GPU tessellation point generation at runtime. The number of points generated by the hardware tessellator is determined by the triangle’s surface area. To achieve real-time performance, ISM constructs depth buffers by splatting each point to *one* randomly chosen depth buffer. This crude approximation creates incomplete depth buffers, which are hole-filled by the pull-push post-processing algorithm discussed earlier.

![Figure 2.12: A diagram from [5] of the steps in the graphics pipeline to implement ISM splatting with points generated by GPU tessellation.](image)

ISM has several drawbacks for real-time applications. Although the use of the GPU tessellation pipeline to generate points is clever, it introduces efficiency problems for tessellated dynamic objects. In this case, the object must be (1) traversed and tessellated, (2) stored to GPU memory, and (3) re-traversed and transformed into points. Additionally, ISM is not appropriate to render shadows cast by direct light sources, since it is unable to capture the high frequency fine details required to avoid undesirable artifacts. Finally, post-processing hundreds of higher resolution depth maps using pull-push is a costly, bandwidth intensive operation.

In addition to the terminology and algorithms discussed here, the interested reader may also find comprehensive surveys of shadow rendering algorithms helpful [35, 22, 73].
2.4 Rendering Reflections in Real Time

To achieve realistic images, the interaction between light energy and each rendered surface is modeled. When photons interact with a surface they either: (1) scatter into the surface, known as transmission, or (2) scatter away from the surface, which is referred to as reflection. The general mathematical model of light scattering is known as the bidirectional scattering distribution function (BSDF). When focusing on just reflection, the ratio of incoming to outgoing energy is described mathematically by the bidirectional reflectance distribution function (BRDF) [2].

Planar Reflections

The left side of Figure 2.13 illustrates the geometry of reflection for planar surfaces. It demonstrates the law of reflection which states, “the angle of incidence (θᵢ) is equal to the angle of reflection (θᵣ)” [2]. Due to this relationship, planar surfaces constitute a special case where photons reaching the eye from reflected surfaces are identical to photons arriving along the incident ray for geometry which has been physically reflected about the surface plane.

Conveniently, this relationship can be exploited by image-based rendering techniques, making planar surfaces easier and more efficient to render than more complex cases [2]. Reflected objects are simply rendered twice, where the geometry and relevant light sources are transformed about the reflection plane during the second render. In the shading step for the planar reflector, the reflection image is sampled when computing incoming radiance. The right side of Figure 2.13 provides an example of this technique from the real-time racing simulation game “Project Cars”. Notice the high quality reflection of the car and background building in the foreground water puddles. Given the availability of dense screen-space data structures such as the

![Figure 2.13: Left: the geometry of planar reflections and the law of reflection [2]. Right: image-based planar reflections in Slightly Mad Studio’s racing game “Project Cars”.](image-url)
G-Buffer [69], aggressive approximations of planar reflections have been presented [79]. However, due to the limited data available in these view specific data structures, these techniques often still require fall-back strategies [78].

Non-Planar and Environment Mapped Reflections

Reflections produced by non-planar objects are unable to use the simplifying assumptions which make planar reflections efficient. Consequently, different techniques have been developed that store incoming radiance values in look-up tables, called environment maps [2]. The concept of environment mapping was first introduced using a longitude / latitude system to map reflection vectors into polar coordinates [7]. The process of environment mapping is depicted on the left side of Figure 2.14 and contains the following steps:

1. Generate or load an image representing the environment.
2. Compute the surface normal and reflection direction for each visible surface.
3. Use the reflection direction to compute an index into the environment map image.
4. Sample and use data from the environment map image to represent incoming radiance.

Figure 2.14: Left: the environment mapping process [2]. Right: The longitude / latitude system used in Blinn and Newell’s original environment mapping scheme [2].

A key drawback of Blinn and Newell’s original longitude / latitude mapping method is the non-uniform distribution of information across the texels of the environment map image. This design causes a number of issues including texture seams and severe texel magnification near the equator. To address these problems, different mapping functions were developed to more effectively utilize the environment map’s resolution. These solutions include sphere mapping [58], cube mapping [28], and parabolic mapping [36]. The cube mapping approach is the most
Figure 2.15: Cubic environment mapping renders the scene to each axis-align face of a cube. Modern hardware supports sampling this custom texture using a 3-component direction vector. Right: an example of a cubic environment map from [28].

A popular method, is supported by modern graphics hardware, and is widely used in real-time applications today. This approach is a significant improvement since it exhibits improved sampling density uniformity in all directions [2]. Shown in Figure 2.15, the cubic environment map is generated by rendering scene geometry to the six axis-aligned faces of a cube. For applications generating environment maps dynamically at run-time, the multiple traversals of scene geometry required to produce the cube map can be prohibitively expensive to accomplish at real-time speeds.
Chapter 3

Multi-View Point-Based Rendering in Real-Time

This chapter introduces the multi-view point-based rendering framework that serves as the conceptual foundation for the research presented in subsequent chapters. First, we describe the motivation behind our point-based strategy. Next, we introduce an algorithmic framework for efficiently generating points from polygonal geometry in real time and parallelizing multi-view image rendering. We detail how our approach realizes untapped potential of existing GPUs by avoiding the inefficiencies of existing rendering methods, while laying the groundwork for future hardware opportunities to further optimize point-rendering for complex visual effects.

3.1 Motivation

The accurate simulation of light requires evaluating complex multi-dimensional integrals for each visible surface [41]. Sampling techniques are commonly employed to approximate these integrals; however, effects such as soft shadows, defocus blur, and reflections typically require a large number of samples to produce noise-free results [49]. Since dense sampling is prohibitively costly for real-time applications, filtering approaches instead attempt to reduce noise by locally sharing information between a small set of samples [77]. Despite achieving acceptable results in many cases, filtering approaches are ultimately limited by the quantity of input samples they operate on. As a result, we are motivated to improve the efficiency of sample generation to increase the number of available samples, and enhance the visual quality of all approaches.

Our algorithms target real-time performance in geometrically complex, dynamic environments using a collection of classic Image-Based Rendering (IBR) techniques [7, 86, 28, 67, 32, 82] since these are still best suited for the compute-to-memory trade-offs inherent in existing graphics hardware. In this context, a sample is typically an image rendered from a relevant view in
support of the desired effect. A depth buffer used in shadow mapping and the faces of a cube map used in environment mapping are examples. These samples are distinctly different than polygonal samples, which are discrete representations of geometric surfaces as opposed to the content of an entire view. At the moment, the fastest method of generating polygonal samples is GPU-based rasterization. Our algorithms utilize the rasterizer for its decades of optimized silicon; however, our general approach accommodates any polygonal sampling method should another prove more capable as hardware architectures evolve (for example, procedural sampling of geometry or ray tracing).

3.1.1 Limitations of Existing Methods

To accelerate multi-view image rendering, we first identify the inefficiencies present in the design of existing rendering architectures. Currently, fixed function rasterization hardware is highly optimized to produce a watertight rendering of polygonal surfaces for a single view. Geometry is streamed through a set of parallel pipelines, where processing is performed for many polygons concurrently. Each input polygon is (1) transformed into the target view, (2) sampled appropriately for the given image resolution, and (3) stored as an image. This design enables extremely high polygon throughput while requiring relatively modest caching structures; however, its bias of throughput over coherence is also a severe limitation. In multi-view scenarios where several images contain large amounts of similar data, the cost of being unable to keep previously processed data cached for fast retrieval during subsequent executions is a significant limiting factor. Unlike its parallel processing of vertices and post-rasterization fragments, this existing design must render views serially and thus parallelizes view rendering poorly. Shown in Figure 3.1, input geometry must be traversed for every view, missing a critical opportunity to reuse coherent polygonal information and post-rasterization data for similar views.

![Figure 3.1](image_url)

Figure 3.1: Existing hardware designs require a traversal of the input geometry, and execution of the full graphics pipeline, for each view when performing multi-view rendering of polygons.
As a thought experiment, consider the case of rendering soft shadows from an area light source using Multi-View Rasterization. Randomly positioned points are distributed across an area light. MVR uses each of these light samples as an eye location and rasterizes geometry between the eye and the receiver to produce many depth buffers. To compute the soft shadow visibility factor, MVR averages the results of shadow mapping using each depth buffer. The bottom row of Figure 3.2 illustrates this scenario.

The depth buffers produced for soft shadow rendering are often very similar, if not identical in some areas, since views distributed across an area light source are typically close together. The top row of Figure 3.2 illustrates the contents of four depth buffers for a triangular occluder and a square area light. Notice how areas of each depth buffer are similar. As an area light grows the coherence between depth buffers decreases; however, a larger area light also produces a larger penumbra region. Noise-free rendering of a large penumbra demands more, closely packed light samples that increase coherence. Illustrated on the bottom-right of Figure 3.2, a smooth penumbra requires a minimum of 256 depth buffers. This scales poorly, since typical occluders in modern real-time applications consist of hundreds of thousands of polygons.
Figure 3.3: A comparison of triangle rasterization implementations rendering multiple depth buffers of a 240,186 polygon character. Rasterizing multiple depth buffers in a single graphics pipeline execution is often slower than serial multi-pass methods due to memory access challenges and limitations of the Geometry Shader.

As graphics architectures evolved, several new solutions emerged for exploiting multi-view data locality and avoiding repetitive computation in the graphics pipeline. Figure 3.3 compares these solutions by rendering a varying number of depth maps of a 240,186 polygon dynamic character. *Single Instanced Draw Call /w GS* uses GPU instancing to automatically execute the graphics pipeline multiple times, and routes polygons to different render targets for each execution using the Geometry Shader. *Single Draw Call w/ GS* executes the graphics pipeline only once, and uses Geometry Shader amplification to rasterize each polygon to many depth buffers simultaneously. *Single Draw Call w/ Instanced GS* is identical to the previous approach, except the invocation of Geometry Shaders is instanced, removing the need for loop flow control in the shader. Surprisingly, none of these solutions are effective and simply rasterizing each depth buffer serially (shown in red) is fastest because of memory access challenges and limitations of the Geometry Shader. Highly customizable software polygon rasterizers utilizing GPU Compute are an attractive alternative to work around these graphics pipeline limitations; however, their performance is unable to match hardware supported rasterization [46]. The recent renewed interest in virtual reality, coupled with new low-level rendering APIs, has inspired advances in the direction of hardware supported multi-view rendering [80], but these initial efforts require significant improvement before practical use.
3.2 A Multi-View Rendering Design

Motivated by these limitations, we seek a more efficient – yet still accurate – alternative to polygonal rasterization that exploits today’s GPU hardware. Inspired by the shrinking number of pixels covered by the average polygon, and thus the decreasing importance of polygonal connectivity information [29], we achieve a more flexible rendering design by instead using points as the primary multi-view rendering primitive. The atomic nature of points is a natural fit for the Same Instruction Multiple Data (SIMD) nature of the GPU’s processor array, especially outside of the traditional graphics pipeline when using GPU Compute.

Since the vast majority of real-time content, authoring tools, and practitioner skills are reliant on polygonal primitives, it is imperative our new rendering design support existing content creation standards without significant modification. To achieve this, scene content is authored as polygons and transformed into a point representation at runtime. By deferring the conversion of assets into points until runtime, our design supports all existing content, gains the flexibility to tailor generated point primitives to the most appropriate representation for the specific rendering goal, and enables a reorganization of the three rendering steps described in Section 3.1.1. Our point-based multi-view rendering design is illustrated below in Figure 3.4.

Figure 3.4: Our point-based multi-view rendering design. Input polygons are converted to point primitives at runtime and then multiple images are rendered in parallel using the points.

The first step, referred to as dynamic point generation, computes an appropriate polygonal sampling rate for some set of output views, samples the input polygons, and stores points in an arbitrary data structure. Critically, unlike the GPU’s graphics pipeline, this design traverses the input polygons only once and samples the surfaces for all views at once. The second step, known as parallel image rendering, renders many images simultaneously by performing splatting using the generated point representation.
Currently, point-based rendering is rarely used for real-time multi-view effects since 1) *dense point sets* appropriate for every possible view cannot be regenerated in real-time for animation, nor be rendered to all views within real time frame budgets; while 2) *sparse point sets* require elaborate splatting and reconstruction algorithms that do not parallelize effectively on existing GPUs, and ultimately suffer from low quality approximations of the original geometry. Existing graphics hardware solves direct illumination by coupling dense sampling tailored to the current view with extremely simple reconstruction that parallelizes well, fully exploiting the GPU’s raw power. Inspired by this approach, our design solves for multi-view approximated indirect illumination effects by enabling a *high density point set tailored per frame* to the current multi-view configuration, coupled with relatively *simple reconstruction* kernels.

### 3.2.1 Generating Points from Polygons

At the turn of the century, the introduction of 3D laser scanning technology re-invigorated interest in point-based rendering methods [29]. The acquisition of highly detailed point sets from real-world objects became possible, and fueled research in the simplification and efficient rendering of incredibly dense point data sets [65, 91, 8]. We refer to point acquisition from 3D scanning as *static point generation*, since points are produced by an offline pre-processing step. Detailed surveys of static point generation [29] and rendering [45] are available. Despite the exciting ability to accurately and efficiently render real objects with exceptional detail, rendering static point sets is impractical and undesirable for many real-time applications. For these applications points must be generated at runtime, typically from polygonal representations. We refer to this process as *dynamic point generation*.

The process of dynamically generating points, or *polygonal samples*, of geometry is one of the most fundamental concepts in three-dimensional computer graphics. Determining the intersection of a view ray with a polygon is the foundation of the object-order rasterization and image-order backwards ray tracing algorithms. However, for performance reasons, real-time implementations of these *view-dependent* algorithms typically solve the ray-polygon intersection problem for a specific view, and ignore how the scattering behavior of light creates the need to solve this problem over an entire hemisphere for each rendered surface. As a consequence, until now *multi-view* and *view-independent* dynamic point generation has received limited research attention. Early work produced points in real time by rendering many orthogonal views of each scene object [30]. Later, Bærentzen used depth peeling [25] in three axis-aligned directions to capture the many depth layers of each view using the GPU [9]. This performed much faster than other methods without GPU acceleration; however, point generation still required multiple seconds of processing time and prohibited interactive performance. More recently, an extension of Imperfect Shadow Mapping (ISM) used the GPU’s new tessellation unit to generate points
from triangles in the geometry stage of the graphics pipeline [5], but is limited by a relatively simplistic point generation heuristic.

A primary focus of this dissertation is multi-view dynamic point generation for various effects that exhibit different levels of multi-view coherence. To cover these topics in detail, we present new view-dependent and view-independent dynamic point generation approaches in the chapters that follow. Chapter 4 details a multi-view view-dependent algorithm that enables up to a $13 \times$ performance improvement in depth buffer rendering and $3 \times$ improvement to overall soft shadow rendering. In Chapter 5, we introduce a new view-independent algorithm, View Independent Rasterization, that leverages fast fixed-function rasterization hardware to sample polygons at runtime without many of the downsides of existing view-independent methods. We then discuss how to apply View Independent Rasterization to multi-view effects such as soft shadows, defocus blur, and environment mapped reflections. Each of these novel point generation techniques employs a parallel image rendering approach that we describe next.

### 3.2.2 Parallel Image Rendering

After point generation, images are rendered using the point representation. Unlike the multi-view rasterization process illustrated in Figure 3.1, multiple images are constructed in a single execution of the GPU. This process is the second step in our multi-view point-based rendering framework and is shown in Figure 3.5 below:

![Figure 3.5: Images constructed in parallel using generated points in a single GPU execution.](image)

Images are constructed by forward mapping, or splatting, points into each output view in parallel. Point rendering is performed by either a) streaming points directly to the Pixel Shader stage, or b) storing points to a separate buffer and/or data structure and dispatching GPU compute threads. Our goal is to keep point rendering kernels as simple as possible: reading a
Figure 3.6: Multi-view point rendering exists on a continuum from serial, to partially parallel, to fully parallel execution per traversal of the point data.

point’s world-space location, and then for each view, applying the corresponding view-projection matrix, snapping the projected location to the nearest neighbor texel in the view’s buffer, and performing z-buffering. Atomic functions resolve race conditions created by multiple points projecting to the same texel. Although this approach implicitly associates a point sample’s area of influence with the extent of a single texel, gaps in constructed images are avoided because of the sampling density calculations completed during dynamic point generation. Other more advanced splatting techniques from the point rendering literature [91] could be employed here; however, they are less efficient and usually unnecessary.

Due to the rapid evolution of graphics architectures towards generalized computation, existing hardware includes a surprising number of ways to accomplish parallel image rendering with points, using either the graphics pipeline or GPU compute. We classify and compare multi-view point rendering approaches based on the amount of parallelism they achieve during each traversal of the point data. Shown in Figure 3.6, image rendering exists on a continuum, and is accomplish by either producing one image per traversal (serial), 2) multiple images per traversal (partially parallel), or 3) all images per traversal (fully parallel). Based on the available feature set of existing GPU hardware, there are no fewer than ten ways to render all points to all images. We detail these implementation options below:

Methods 1 - 4: Serial

Similar to the multi-view rasterization of polygons depicted in Figure 3.1, a serial point rendering approach constructs images by traversing point data once for each rendered image. Illustrated on the top of Figure 3.7, the simplest serial approach uses multiple intermediate image buffers and computes final color values by reading all images in a final lighting step. The storage and peak memory bandwidth required by the final lighting step are a limiting factor when many intermediate images are necessary. A common alternative strategy maintains a data structure – the accumulation buffer – that aggregates intermediate results, bounds total
memory use, and distributes shading memory bandwidth over several passes. This method, illustrated on the bottom of Figure 3.7, allows serial rendering passes to reuse the same image buffer at the cost of increasing overall memory traffic.

Each of these strategies can be implemented using the graphics pipeline or GPU compute. When implemented in the graphics pipeline, serial point rendering loads and projects points in the Vertex shader, passes points through the rasterizer and Pixel shader, and z-buffers as usual with the GPU’s Raster Operator (ROP) hardware. When implemented in GPU compute, serial dispatches of Compute shader threads load, project, shade, and z-buffer all points. As we demonstrate in following chapters, GPU compute point rendering performance significantly improves upon the graphics pipeline since points are not processed by the rasterizer.

Figure 3.7: Serial multi-view point rendering traverses point data once for each rendered image. Top: A unique intermediate image buffer per traversal. Bottom: Employing an accumulation buffer decreases memory use and distributes shading memory bandwidth over traversals.
Methods 5 - 8: Partially Parallel

Large quantities of fast global video memory (12 GB and rising) are now available on existing graphics hardware. Illustrated in Figure 3.8, this change in storage capacity makes rendering more than one intermediate image per traversal practical. Rendering multiple images simultaneously reduces the total memory traffic incurred by loading points and increases memory access coherence during point rendering. As with serial rendering, a partially parallel strategy can also bound memory usage by modifying point traversal passes with an accumulation buffer. When implementing this method in the graphics pipeline, points are loaded in the Vertex shader, but projection is deferred until either the Geometry or Pixel shader stages. This important change prevents the rasterizer from culling points that project outside any single view.

Figure 3.8: Partially Parallel multi-view point rendering produces multiple images for each point data traversal. Top: A 2 Pass, 4 image example. Bottom: The same a 2 Pass, 4 image example employing accumulation. Note how the number of intermediate images is reduced.
Methods 9 & 10: Fully Parallel

Illustrated in Figure 3.9, a fully parallel point rendering approach constructs all intermediate images in a single traversal of the point data. In this method, the memory bandwidth incurred from loading points is minimized and coherence during point rendering is maximized. GPU memory bandwidth ultimately limits the number of images that are efficiently rendered by each point data traversal; therefore, the fully parallel method is not appropriate in memory bandwidth limited environments (legacy and low-end GPUs).

Unlike previous methods, implementations of this method differ due to limitations in the hardware’s architecture. GPU compute is able to render any number of intermediate views simultaneously, but is not able to use the texture sampling hardware for mipmapming and filtering. When using the graphics pipeline and the Geometry Shader, each input point is copied for each output view and rasterized to the corresponding image. The Geometry Shader is restricted to 32 copies per execution. If the number of output views exceeds 32, the graphics pipeline implementation must fallback to partially parallel execution. The Geometry Shader also has well known performance problems when performing data amplification [2]; however, the Pixel Shader stage later in the pipeline is capable of using texture sampling hardware.

![Diagram of Method 9](image1)

![Diagram of Method 10](image2)

Figure 3.9: Fully parallel multi-view point rendering produces all intermediate images in a single point data traversal. Top: Any number of output images is possible with GPU Compute. Bottom: Implementation in the graphics pipeline is more complex and involves more shader stages. The Geometry Shader creates copies of the original point to render each image.
Chapter 4

View Warped Multi-View Soft Shadows for Area Lights

In this chapter, we apply our multi-view point-based rendering concept to an important visual effect: soft shadows cast from area lights. As we discussed in Chapter 2, shadows provide important perceptual cues. These cues are even stronger when shadows exhibit natural characteristics such as penumbras [52]. Recent advances in the precomputation and compression of visibility data for static occluders has enabled real-time shadowing algorithms to focus on the subset of occluders and lights that are animated or otherwise dynamic [70, 61]. For these dynamic elements, Multi-View Rasterization (MVR) is the best proxy for a ground truth solution that real-time applications can consider to accurately reproduce soft shadows; however, its requirement of multiple traversals of occluder geometry prevents real-time performance [22].

We introduce a novel algorithm, called View Warped Soft Shadows (VWSS), that addresses the MVR bottleneck for dynamic occluders and area lights by applying our multi-view point-based design. We establish a useful new point on the quality/performance continuum by (1) simplifying the soft shadowing problem using a view-dependent sampling strategy, (2) avoiding common artifacts by tailoring points to many nearby views, and (3) significantly increasing the efficiency of rendering depth buffers by reorganizing the rendering workload for coherence. VWSS is simple to implement and integrates easily into existing rendering engines. Over the course of this chapter, we provide a practical analysis of point rendering implementations on existing GPU hardware with recommendations to achieve optimal performance, and compare our new algorithm with existing accurate and approximate soft shadowing methods.
4.1 Overview

The core of our VWSS algorithm is conceptually similar to image warping, which can reduce the computation required to produce novel images by performing geometric transformations on previously rendered data [87, 57, 54, 82]. Image warping is most successful when the source and derived images are highly similar, since traditional rasterization emits points with a distribution and sampling density optimized for a single view and output resolution. Since views distributed across an area light source are often spatially proximal, the associated depth buffers contain highly similar data ideal for this type of reuse. We leverage the discretize-and-reuse strategy of image warping by traversing occluder geometry only once, rasterizing a central view of the area light to produce a point cloud of view-dependent data, and rendering a complete set of depth buffers in parallel using the points.

When rendering images with warping, the data necessary to produce a complete novel image is not always available in previously rendered images. Previous work addresses this problem by filtering [82, 90] or storing additional data in the source [54, 76]. Inspired by layered depth images [76], our algorithm avoids common artifacts associated with warping (e.g. at areas of disocclusion) by storing all front facing rasterized fragments along each ray of a central view enlarged to contain all light sample views.

Figure 4.1: Soft shadowing of a 672,927 polygon scene with complex occlusion and dynamic, skinned geometry at 1080p resolution. Our new VWSS algorithm (middle) computes depth buffers (D) 3.7× faster and the complete image (T) 2.15× faster than Multi-View Rasterization (left). Image quality is compared against a 2,560spp reference and reported using RMSE and HDR-VDP2 perceptual heat maps (bottom). VWSS achieves significantly higher numerical and perceptual quality compared to PCSS (right) with a minimal increase in rendering time.
To our knowledge, this is the first application of warping to accelerate secondary visibility determination for many views. Existing work employing warping focuses on reusing complete images to accelerate rendering of the primary view. Temporal warping techniques are a critical tool in achieving the low latency and high performance required for virtual reality headsets and stereoscopic displays [57, 6]. A recent depth of field algorithm constructs a light field by warping the visible surfaces of the primary view and filtering the results [90]. Unlike previous work that benefits from warping, the denser sampling of our design eliminates the need for post-processing or filtering, and computes secondary lighting effects at practical speeds.

Currently, commercial real-time applications typically budget a 2–4 millisecond window for all shadow related computation [13]. Applications including soft shadows must choose between impractically slow but accurate multi-view methods or fast but approximate single-view solutions with objectionable artifacts [64]. View Warped Soft Shadows is much more accurate than existing fast single-view methods, and much faster than existing accurate multi-view solutions. VWSS renders depth buffers up to $13 \times$ faster and computes soft shadows over $3 \times$ faster than MVR for typical real-time dynamic occluder geometry. These improvements enable VWSS to facilitate a more practical quality/performance tradeoff and produce soft shadows in the required time window. For example, when using 24 depth buffers and 120 samples per pixel (spp), VWSS exhibits similar performance and less error than Percentage Closer Soft Shadows (PCSS) without the serious failure case artifacts that plague single-view approximations. Additionally, the perceptual image comparison measure HDR-VDP2 [53] rates our algorithm’s output significantly closer to MVR (see Figure 4.1).

We demonstrate our algorithm running on realistic game content in real-time and compare its speed and quality to MVR and PCSS. Our experimentation focuses on local area lights; however, it is straightforward to apply our concept to cascaded shadowing solutions for global light sources [47]. Since our approach is based on shadow mapping, the associated limitations are thoroughly understood. We also study efficient point rendering — critical for our algorithm — on modern GPUs. When provided identical point data workloads, we observe notable performance improvements using GPU compute instead of the graphics pipeline.

4.2 Algorithm

Accurate soft shadowing requires visibility data from many views across an area light. These views are quite similar and promise computational efficiencies, but current graphics pipeline solutions for exploiting data locality and avoiding repetitive computation are ineffective (see Figure 3.3). Highly customizable software triangle rasterizers utilizing GPU compute [46] are an attractive alternative to these limitations; however, their performance is unable to match hardware supported MVR. We therefore seek a more efficient yet accurate alternative to triangle
rasterization that exploits today’s GPU hardware.

*View Warped Soft Shadows* is a point-based warping approach capable of quickly and accurately producing multiple depth buffers. This is accomplished by 1) transforming occluder geometry from triangles to a specialized point set in a single graphics pipeline execution; and 2) constructing multiple depth buffers simultaneously by warping and z-buffering the points in parallel (see Figure 4.2). Since views on an area light are similar, a carefully formed point cloud enables the rendering of depth buffers with accuracy sufficient for high quality soft shadowing. Point primitives map well to the SIMD design of the GPU, affording efficient and more flexible parallelization of rendering workloads.

We present buffered and unbuffered implementations of this concept. The buffered implementation is faster and capable of efficiently generating a larger number of depth buffers, but requires a GPU memory allocation large enough to store intermediate point data. The second unbuffered implementation is slower and generates fewer depth buffers, but does not require memory management.

### 4.2.1 Buffered VWSS

Our buffered implementation, illustrated in Figure 4.2, rasterizes occluder geometry, stores generated points in an unstructured linear (D3D11 Append/Consume) buffer, and then renders depth buffers by traversing the point data.

![Figure 4.2: The data flow of buffered VWSS point generation and depth buffer construction. In Step 2, each compute (or pixel) shader thread warps a point to multiple depth buffers simultaneously, generating several depth buffers in a single traversal of the point data.](image)
Point Generation

Traditional sampling approaches for image warping are insufficient when computing multi-view visibility for area lights. Shown at the top of Figure 4.3, warping of the view-dependent samples stored in a typical z-buffer creates noticeable light leaking artifacts. These artifacts result from lack of data at 1) areas of disocclusion and 2) the shadow map boundary. We solve the first problem by storing all front-facing samples produced by the rasterizer (inspired by layered depth images [76]); and the second by using a central view enlarged to contain the area light matched with a render target scaled to produce depth buffers that receive a comparable number of samples to MVR. This is illustrated on the bottom of Figure 4.3.

To create a single view that contains the area light and all light sample frusta, we initially locate the eye $v_e$ at the center of the area light $l_c$. We ensure the eye’s vertical field of view $fov_v$ and horizontal field of view $fov_h$ match the respective fields of view of the light sample

![Figure 4.3: Traditional image warping (top) incorrectly leaks light at areas of disocclusion (b) and shadow boundaries (a,c). These artifacts are eliminated by using VWSS (bottom).](image_url)
views. We then translate $v_e$ along the negated light normal vector $\hat{\mathbf{n}}$ by the distance $d_{l_e \rightarrow v_e}$, calculated using Equations 4.1 and 4.2. $w$ and $h$ are the width and height of a rectangular area light. To achieve the best performance, the field of view aspect ratio of the light samples and containing view should mimic the shape of the area light.

$$v_e = l_c - (d_{l_e \rightarrow v_e} \cdot \hat{\mathbf{n}}) \tag{4.1}$$

$$d_{l_e \rightarrow v_e} = \frac{\max(w, h)}{2 \cdot \tan\left(\frac{\max(fov_v, fov_h)}{2}\right)} \tag{4.2}$$

To ensure VWSS samples each depth buffer at a rate similar to MVR and avoids light leaking, VWSS pairs its containing view with a render target larger than the resulting depth buffers. Given a target depth buffer of resolution $m^2$, the scaled resolution $v_{res}$ is computed for VWSS’s containing view by applying Equation 4.3, where $z_n$ is the central light frustum’s near plane. By locating $z_n$ in the same plane as the light sample view’s near plane, we avoid creating samples behind the light source which would result in false occlusion.

$$v_{res} = \left[ m \cdot \frac{z_n + d_{l_e \rightarrow v_e}}{z_n} \right]^2 \tag{4.3}$$

VWSS disables z-buffering, rasterizes the resulting containing view in the graphics pipeline, and appends the world-space 3D position of each front facing fragment to an unstructured linear buffer in Pixel Shaders. During this stage, our approach does not actually write to a render target; therefore, VWSS avoids allocating depth buffer memory and instead simply applies Equation 4.3 to the viewport dimensions it supplies to the rasterizer stage.

**Parallel Depth Map Rendering (Warping)**

Next, VWSS constructs depth buffers corresponding to each light sample by warping the points produced in the previous stage. Both the graphics pipeline and GPU compute are capable of exploiting locality by projecting each point into multiple depth buffers during a single thread execution. As a result, unlike triangle rasterization, it is possible to more efficiently construct $n$ depth buffers with less than $n$ traversals of our point data (illustrated in Figure 4.2).

VWSS invokes a GPU thread for each point. In each thread, we employ a basic warping method that reads a point’s world-space location, applies the view-projection matrix corresponding to each light sample, snaps the projected location to the nearest neighbor pixel, and performs z-buffering. Atomic functions resolve race conditions caused by arbitrary points projecting to the same depth buffer texel. Although this implicitly limits a point sample’s area of influence to a single depth buffer texel, the careful construction of the point data allows us to sidestep more complex surface splatting techniques [91].
As described in Chapter 3, we classify and compare multi-view point rendering approaches based on the amount of parallelism achieved by each traversal of the point data (see Figure 4.8 for performance results). Rendering depth buffers from points is accomplished by either producing 1) one depth buffer per traversal (serial), 2) multiple depth buffers per traversal (partially parallel), or 3) all depth buffers in a single traversal (fully parallel). Memory bandwidth ultimately limits the number of depth buffers that are efficiently rendered by each point data traversal; therefore, the fully parallel approach is not appropriate in bandwidth constrained environments. Our experimental results indicate that generating more than 32 $1024^2$ resolution depth buffers in one traversal saturates memory bandwidth and reduces the efficiency of even the most capable graphics hardware currently available.

### 4.2.2 Unbuffered VWSS

Similar to many transparency algorithms [88], the exact amount of memory buffered VWSS requires for point storage is unknown until render time, since view, depth complexity and dynamic geometry vary. This can require frequent reallocation of GPU memory, reducing the achievable performance of the buffered implementation.

We introduce an alternative VWSS implementation for scenarios where bounded memory is desirable and fully parallel depth buffer construction is viable. Shown in Figure 4.4, this method combines the point generation and depth buffer construction steps described previously into a single execution of the graphics pipeline. We perform point generation as detailed above; however, instead of storing points, Pixel Shader threads immediately warp each point to all target depth buffers. This avoids the intermediate storage of points by leveraging the direct

![Figure 4.4](image-url)  
Figure 4.4: The data flow of unbuffered VWSS. Point generation and depth buffer construction steps are combined into a single execution of the graphics pipeline, as points are streamed from the rasterizer directly to pixel shader threads.
connection between the rasterizer and pixel shader stage. As shown on the left of Figure 4.8, this strategy is slower than a buffered implementation using GPU compute, potentially due to cache thrashing and memory bandwidth bottlenecks as points are simultaneously streamed from the rasterizer and written to depth buffers. Despite this, the advantage of simplified memory management is well worth the small performance penalty when rendering a modest number of depth buffers.

4.3 Results

We implemented VWSS in a deferred renderer, because these are popular in cutting-edge commercial game engines. The complete VWSS algorithm computes soft shadows by 1) rendering a G-Buffer, 2) generating the intermediate point cloud for each area light, 3) rendering depth buffers from the point data, and 4) reading depth buffers, averaging visibility results, and shading in a final fullscreen pass. Steps 1 and 4 are identical for MVR and VWSS.

Our experimental test environment uses Windows 10, Direct3D 11.1, Intel i7-4790k CPU @ 4.0 GHz, and a NVIDIA Maxwell Titan X GPU. We report GPU times averaged from 1,000 frames of execution. All depth buffers use 32-bit floating point precision and light sample depth buffers are $1024^2$ resolution. Our tests use scenes of varying polygonal complexity that contain skinned and animated assets from Unity Technologies’ real-time Blacksmith demonstration. Our scenes include complex occlusion from static and dynamic occluders with over 500,000 dynamic polygons, which is typical in high-end game titles [81]. We list the geometric complexity of our four primary test scenes in Table 4.1, and images of them can be found in Figures 4.1, 4.3, 4.6, and 4.11. MVR and VWSS use identical area light sample positions with Poisson distribution. After an informal survey of game developers, we chose as a real-time point of comparison a 160spp PCSS implementation with a 64-sample blocker search and 96-sample PCF filter. All VWSS performance numbers are captured from buffered VWSS unless otherwise specified.

We compare the visual quality of all algorithms against a reference image created using MVR with 512 rasterized depth buffers and 2,560 visibility samples per pixel. As a single numerical measure of image quality, we use Root Mean Squared Error (RMSE). To indicate perceptual image quality, we use the image comparison measure HDR-VDP2 [53], which includes various filters modeling human vision.

4.3.1 Quality vs. Performance

Figure 4.5 shows the quality versus performance tradeoff of MVR, VWSS, and PCSS for the Warriors and Cliffside test scenes. The Warriors scene is composed of four dynamic characters and 510,523 skinned animated polygons. The Cliffside scene is 672,927 polygons and mixes
Figure 4.5: The numerical image quality (RMSE) versus performance (ms) for the Warriors scene (top) and Cliffside scene (bottom) using MVR (red), VWSS (blue), and PCSS (grey). MVR and VWSS use 24, 32, 64, 96, and 128 depth buffers, with quality improving and performance declining as the number of buffers grows. MVR and VWSS reach diminishing quality returns at 96 depth buffers.
Figure 4.6: The perceptual quality (HDR-VDP2) and performance of VWSS (left) and PCSS (right) both using 160spp for the Warriors scene (top) and Cliffside scene (bottom). Heatmaps indicate the probability differences between each image and the reference will be perceived. Red areas indicate a higher probability and blue areas indicate lower probability.
static, animated, and skinned polygons with complex occlusion relationships. VWSS produces an image with significantly less error than PCSS, while performing at similar speeds when using a comparable number of samples per pixel. Each point on the MVR and VWSS curves use an increasing number of depth buffers (24, 32, 64, 96 or 128), reducing speed but improving quality. Note that PCSS appears as a single data point in plots since it uses only one depth buffer. VWSS’s quality-to-performance curve is similar in shape to MVR’s; however, overall performance is much faster, making it a more practical alternative.

Figure 4.6 displays the output of VWSS and PCSS both using 160spp for the Warriors and Cliffside test scenes. The HDR-VDP2 heatmaps visualize the probability that an observer will perceive the differences between each algorithm’s output and the reference image. Red areas indicate a higher probability and blue areas indicate lower probability. VWSS renders an image with much higher perceived quality in nearly the same time as PCSS. In the following sections, we present detailed performance and quality data from our experimental results.

4.3.2 Performance

Table 4.1 displays GPU performance data for each rendering step of MVR, VWSS, and PCSS. Each test scene increases geometric and point data complexity to demonstrate how performance scales from a moderately complex single character to a realistic scene with multiple characters, complex geometry and non-trivial occlusion. Table 4.1 reports two ends of the performance spectrum: a real-time frame budget setting using 24 depth buffers and 120spp, and a high quality configuration using 128 depth buffers and 640spp.

Figure 4.7 plots the overall performance of these two settings for each algorithm as geometric complexity increases. VWSS accelerates multi-view depth buffer rendering time $\sim 2\times-13\times$ and improves total rendering time up to $3\times$. Since VWSS traverses occluder geometry a single time, VWSS’s performance is weakly linked to geometry, much like PCSS. As a result, VWSS often matches PCSS performance while producing images with less total error (RMSE). Scenes with complex geometry or settings that generate many depth buffers benefit the most from VWSS.

Figure 4.8 compares different implementations of the VWSS concept. The top plot compares the buffered and unbuffered implementations described in Section 4.2. The fully parallel implementation of the unbuffered approach is slower than the buffered version; however, at the real-time frame budget setting the 6% performance penalty is worth the advantage of simplified memory management. The bottom plot compares depth buffer point rendering implementations using GPU compute and the graphics pipeline for buffered VWSS. Each approach processes identical point data workloads. Due to the Geometry Shader, the graphics pipeline is unable to render points in a partially parallel manner without decreasing performance. Unexpectedly, even when not invoking Geometry Shaders, a serial implementation using the graphics pipeline
is still significantly slower than even serial GPU compute, especially as the number of rendered depth buffers increases. This demonstrates that passing point data through the graphics pipeline as vertices rather than directly loading them in compute (Figure 4.8, bottom), or as output generated from rasterized triangles (Figure 4.8, top), drastically reduces performance.
Table 4.1: GPU performance of soft shadowing algorithms on various scenes. Multi-View algorithms use 24 depth buffers @ 120 spp (and 128 depth buffers @ 640 spp in parentheses). VWSS accelerates multi-view depth buffer rendering time $\sim 2\times-13\times$ (highlighted blue) and improves total rendering time up to $3\times$ (highlighted red). VWSS matches PCSS performance (bold) while producing images with less total error (RMSE). Scenes with complex geometry benefit the most from VWSS. GPU times reported in milliseconds (ms).

<table>
<thead>
<tr>
<th>Scene</th>
<th># of Triangles</th>
<th># of Points</th>
<th>Algorithm</th>
<th>G-Buffer</th>
<th>Point Generation</th>
<th>Depth Buffers</th>
<th>Lighting</th>
<th>Total</th>
<th>SD</th>
<th>×Faster Depth</th>
<th>×Faster Total</th>
<th>RMSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Challenger</td>
<td>90,007</td>
<td>40,541</td>
<td>MVR</td>
<td>0.261</td>
<td>0.860 (5.396)</td>
<td>1.300 (7.822)</td>
<td>2.421 (13.485)</td>
<td>0.161 (1.709)</td>
<td></td>
<td></td>
<td></td>
<td>1.497 (0.407)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>VWSS</td>
<td>0.262</td>
<td>0.115</td>
<td>0.286 (1.118)</td>
<td>1.259 (7.757)</td>
<td><strong>1.922</strong> (9.260)</td>
<td>0.421 (0.943)</td>
<td>2.15 (4.38)</td>
<td>1.26 (1.46)</td>
<td><strong>1.801</strong> (0.774)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PCSS</td>
<td>0.262</td>
<td>0.021</td>
<td>1.590</td>
<td><strong>1.873</strong></td>
<td>0.092</td>
<td></td>
<td></td>
<td></td>
<td><strong>2.753</strong></td>
</tr>
<tr>
<td>Volund</td>
<td>240,186</td>
<td>57,188</td>
<td>MVR</td>
<td>0.300</td>
<td>1.358 (10.816)</td>
<td>1.308 (8.242)</td>
<td>2.966 (19.371)</td>
<td>0.149 (2.928)</td>
<td></td>
<td></td>
<td></td>
<td>1.995 (0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>VWSS</td>
<td>0.293</td>
<td>0.109</td>
<td>0.261 (1.216)</td>
<td>1.272 (7.952)</td>
<td><strong>1.935</strong> (9.588)</td>
<td>0.131 (0.795)</td>
<td>3.67 (8.16)</td>
<td>1.53 (2.02)</td>
<td><strong>1.428</strong> (0.772)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PCSS</td>
<td>0.291</td>
<td>0.052</td>
<td>1.767</td>
<td><strong>2.110</strong></td>
<td>0.105</td>
<td></td>
<td></td>
<td></td>
<td><strong>2.503</strong></td>
</tr>
<tr>
<td>Warriors</td>
<td>510,523</td>
<td>101,002</td>
<td>MVR</td>
<td>0.400</td>
<td>4.095 (22.101)</td>
<td>1.280 (7.469)</td>
<td>5.775 (29.990)</td>
<td>0.298 (5.876)</td>
<td></td>
<td></td>
<td></td>
<td>1.437 (0.403)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>VWSS</td>
<td>0.397</td>
<td>0.471</td>
<td>0.342 (1.253)</td>
<td>1.120 (6.938)</td>
<td><strong>2.330</strong> (9.098)</td>
<td>0.481 (0.440)</td>
<td>5.04 (12.82)</td>
<td>2.48 (3.30)</td>
<td><strong>1.627</strong> (0.702)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PCSS</td>
<td>0.394</td>
<td>0.157</td>
<td>2.060</td>
<td><strong>2.611</strong></td>
<td>0.144</td>
<td></td>
<td></td>
<td></td>
<td><strong>3.608</strong></td>
</tr>
<tr>
<td>Cliffside</td>
<td>672,927</td>
<td>347,060</td>
<td>MVR</td>
<td>0.683</td>
<td>5.148 (30.320)</td>
<td>1.271 (7.679)</td>
<td>7.102 (38.834)</td>
<td>0.540 (6.243)</td>
<td></td>
<td></td>
<td></td>
<td>1.129 (0.255)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>VWSS</td>
<td>0.660</td>
<td>0.746</td>
<td>0.636 (2.910)</td>
<td>1.261 (6.823)</td>
<td><strong>3.303</strong> (11.525)</td>
<td>0.575 (0.821)</td>
<td>3.73 (8.29)</td>
<td>2.15 (3.37)</td>
<td><strong>1.316</strong> (1.129)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PCSS</td>
<td>0.681</td>
<td>0.153</td>
<td>2.246</td>
<td><strong>3.080</strong></td>
<td>0.161</td>
<td></td>
<td></td>
<td></td>
<td><strong>3.232</strong></td>
</tr>
</tbody>
</table>
Figure 4.7: GPU performance of shadowing algorithms as geometric complexity increases using 24 depth buffers @ 120spp (top) and 128 depth buffers @ 640spp (bottom). Multi-View Rasterization (red) exhibits an expected linear rise in rendering time as geometric complexity increases. The performance of VWSS (blue) is weakly linked to scene geometry, enabling VWSS to match or exceed PCSS performance (top) and produce a large number of depth buffers in a fraction of the time it takes MVR (bottom).
Figure 4.8: Top: GPU performance comparison of buffered and unbuffered VWSS implementations. The unbuffered VWSS implementation incurs a performance penalty due to its fully parallel approach. Bottom: performance of compute versus graphics pipelines when performing buffered VWSS depth buffer point rendering. The graphics pipeline is significantly slower than GPU compute.
4.3.3 Quality

Despite its real time speed, VWSS produces soft shadows of remarkable quality, by both numeric and perceptual measures. Figures 4.1 and 4.6 show that even at speeds similar to PCSS, VWSS produces imagery with quality comparable to MVR. VWSS achieves this by building depth buffers very similar to MVR’s, while (like PCSS) using only the depth samples of a single view.

Not only is VWSS more accurate than PCSS overall, but it also avoids the hard failure cases that plague PCSS and similar single-view techniques. Figure 4.9 illustrates two such cases. First, the PCSS blocker search step is not always capable of detecting thin occluders represented by only a few depth buffer texels. This creates noticeable holes in the rendered shadow. Even when using the same number of samples per pixel, VWSS’s higher quality point representation prevents failures of this kind. Second, the PCSS blocker search step computes an average of the depth values found. This approximation prevents PCSS from properly handling occluder fusion in some scenarios. By z-buffering points in many depth buffers, VWSS handles occluder fusion identically to MVR, but in a fraction of the time.

Figure 4.9: Top: The thin sword is not always captured by a 32 sample PCSS blocker search and creates noticeable holes in the sword’s shadow (left). VWSS using an identical number of samples per pixel (middle) has no such failure, delivers quality closer to MVR (right), and retains rendering time similar to PCSS. Bottom: PCSS is unable to handle occluder fusion properly in some scenarios (left), while VWSS shadows are similar to MVR.
Aliasing is a common quality challenge associated with all image based shadowing algorithms, and VWSS benefits from the wealth of research available on the topic. Shown in Figure 4.10, VWSS exhibits self shadow banding artifacts similar to PCSS when using a constant depth bias value during shadow mapping. This similarity exists since VWSS still samples occluder geometry from only a single view. Many techniques exist to reduce self shadowing artifacts. Figure 4.10 demonstrates Normal Offset Depth Biasing [38] as one possible solution.

Figure 4.10: Shading modified to emphasize self shadowing. Top: VWSS exhibits self shadow banding similar to PCSS. Bottom: MVR shows no banding artifacts, while Normal Offset Depth Biasing applied to VWSS addresses the self shadowing and disconnected contact shadows.
The value of any warping technique is demonstrated by examining how rapidly the quality of the resulting image declines as the novel viewpoint moves away from the original viewpoint. The containing view used by VWSS ensures that this decline is very gradual. Figure 4.11 shows soft shadows generated by VWSS as the light source grows to 64× its original area. Light samples are distributed on the area light at a constant rate for all light sizes. Even for large lights, the results are exceptionally similar to MVR.

Figure 4.11: Top: VWSS output for the Volund scene as the area light (yellow) grows. All area lights use one light sample per two units of light area. Bottom: A close-up comparison of shadows generated by MVR (left) and VWSS (right) reveals very similar results even for large lights.
VWSS displays less temporal and geometric aliasing than PCSS since it uses multiple depth buffers; but more aliasing than MVR, since it does not resample occluder geometry for each depth buffer. Common cascaded shadow mapping [47] and temporal reprojection [72, 75] solutions are straightforward to apply to VWSS to improve aliasing. We include a comparison of geometric and temporal aliasing for MVR, VWSS, and PCSS in the supplementary video, in addition to an example of VWSS with temporal averaging.

4.4 Conclusion

VWSS establishes a useful new point on the quality/performance continuum for applications rendering soft shadows in real-time. By traversing occluder geometry only once and exploiting parallelism and locality while creating depth buffers, VWSS is much faster than MVR and comparable in speed to PCSS. Since VWSS creates an array of depth buffers similar to MVR’s, its soft shadows are much better than PCSS’s, and comparable in quality to MVR’s.

Although VWSS has similarities to Imperfect Shadow Mapping, it takes a fundamentally different approach aimed specifically at exploiting current hardware to produce high quality shadows cast by direct light sources in real time. Though both VWSS and ISM use point clouds, VWSS produces point data tailored to the current area light with fast rasterization hardware ideal for view-specific rendering. While both algorithms create many depth buffers, VWSS restructures computation to increase GPU utilization, enabling much denser sampling of visibility. The resulting approach takes full advantage of the GPU’s raw parallel power, allowing a relatively simple algorithm to produce shadows at useful quality and speed.
Chapter 5

View Independent Rasterization

As discussed in Chapter 3, multi-view effects exhibit varying amounts of coherence among views when approximating visual effects with image-based samples. Chapter 4 presented a multi-view polygonal sampling method for effects with meaningful spatial locality and coherence across views; however, this approach is not effective when view directions – and therefore visible content – are considerably different. A common example of this scenario is the six axis-aligned faces of a cube map, a view-independent intermediate data structure often used for environment mapped reflections. This challenging case is further complicated by (1) geometrically complex content and the need for (2) multiple unique cube maps for multiple reflective objects.

Although the fixed function rasterization hardware of modern GPUs are highly-optimized point generation machines, the existing view-dependent serial design of the GPU accelerated graphics pipeline struggles to leverage coherent data when rendering cube maps. By design,

Figure 5.1: Left: the view plane is uniformly sampled and the plane of a polygon in the view frustum receives variable sampling under perspective projection. Right: a special case, where the view plane and the polygon have uniform sampling rates when parallel, differing only by a ratio proportional to distance (under perspective projection).
GPU rasterization units produce *view-dependent* samples of input geometry by transforming all polygons of a *single view* using a *single view-projection plane*. The result is uniform sampling across the view plane and variable sampling across each projected polygon. See the left side of Figure 5.1 for an illustration. Critically, a special case exists when the *view plane* and the plane of a transformed *polygon* are *parallel*. Shown on the right of Figure 5.1, this view-to-polygon relationship creates a scenario where the sampling rate is *uniform* across both planes, and only differs by (1) a ratio proportional to the distance between the planes in a perspective projection; or (2) a scaling factor of the view’s near plane in an orthographic projection. In this situation, the rasterizer can be utilized as a powerful *view-independent polygon sampler* by ensuring the convex hull of a parallel polygon fits within the view volume before rasterization. Note how increasing the projected area *uniformly increases* sampling rate, while decreasing projected area *uniformly decreases* sampling.

We leverage this insight to generate a high density point cloud useful for *many views* in real time. We refer to this novel point generation approach as *View Independent Rasterization (VIR)*. By computing a view-projection plane *unique to each polygon* in the geometry stage of the graphics pipeline, VIR achieves its real-time performance by traversing the source geometry only once and sampling with fast, fixed-function rasterization hardware. In this chapter, we detail how to implement VIR and tune the sampling rate per polygon for multiple views, before exploring important edge cases and optimization techniques.

### 5.1 View Independent Transform

To perform *view independent rasterization* of arbitrary polygons, we compute and apply a unique transformation matrix, $T_{\text{vir}}$, to each polygon. $T_{\text{vir}}$ centers each polygon about the z-axis, aligns the polygon’s plane parallel to the X-Y plane, and is written as the $4 \times 4$ matrix:

$$
T_{\text{vir}} = \begin{bmatrix}
\hat{u}_x & \hat{u}_y & \hat{u}_z & -(\hat{u} \cdot c) \\
\hat{v}_x & \hat{v}_y & \hat{v}_z & -(\hat{v} \cdot c) \\
\hat{n}_x & \hat{n}_y & \hat{n}_z & -(\hat{n} \cdot c) \\
0 & 0 & 0 & 1
\end{bmatrix}
$$

where $\hat{n}$, $\hat{u}$, and $\hat{v}$ are mutually orthogonal unit vectors forming the polygon’s *basis frame*. $\hat{n}$ is the polygon’s unit length geometric normal. The vector $c$ is the translation of the polygon’s centroid to the world origin. By computing the translation from the polygon’s centroid, we place the polygon in the center of the view and maximize the available texel area of the image resolution used during rasterization. Any other point on the polygon, such as a polygon vertex, may be used instead of the centroid. Illustrated in Figure 5.2, we compute and apply $T_{\text{vir}}$ in the Geometry Shader before the rasterization stage. Alternatively, these steps can be performed in
the Hull Shader; however, this requires hardware tessellation support, prevents surfaces from being traversed by the hardware in a single pass, and incurs unnecessary overhead by activating the tessellation pipeline for all polygons.

By applying the view independent transform, we position the VIR camera at the world origin looking down the positive z-axis (in a left-handed coordinate system). Next, we find and apply a projection matrix that sets a conservative multi-view sampling rate for the polygon. The computation of this matrix is discussed in detail in the next section. Finally, we store points generated by the rasterizer using the Pixel Shader stage. In this step, culling of point samples may occur to further tailor the point set to the current frame’s multi-view requirements. We accomplish this using view-frustum and/or back-face comparisons before point storage. For simplicity, we store points in an unstructured, linear (D3D11 Append/Consume) buffer; however, an arbitrary data structure may be used.

5.2 Multi-View Sampling Rate

To accomplish multi-view rendering with VIR, we compute a per-polygon multi-view sampling rate: the density of view-independent polygonal samples necessary to ensure watertight reproduction of the polygon in many views. Since VIR traverses the input geometry only once and performs sampling when the polygon and view planes are parallel, VIR represents the multi-view sampling rate as a single value we denote as \( s_{mv} \).

Due to the potentially high variability between any pair of target views, coupled with the varying size and shape of a polygon within each view, computing an optimal multi-view sampling rate is a challenging task. The view-inspired multi-view sampling approach of View Warped Soft Shadows takes advantage of knowledge about views spread across an area light to side-step this problem; however, the relationship between views and polygons is less predictable in more
general scenarios. In this case, to ensure VIR samples polygons sufficiently – placing at least one polygonal sample in each texel of all relevant views – the multi-view sampling rate must account for (1) the level of detail required of the polygon by each view in the set of views, (2) the variability of polygon size and shape within each view, and (3) the variability of polygon size and shape across the set of all views.

We solve these problems by employing a conservative sampling approach – producing a high density point cloud – and then optimizing point rendering based on the desired effect’s per frame multi-view requirements. Unlike existing real-time point rendering techniques, this approach ensures a high quality representation of the original geometry, guarantees watertight reproduction, and encourages the reuse of point data for multiple purposes by not specializing the data set for a single effect. These advantages are not without a price; conservative sampling comes at the cost of requiring various optimization techniques for rendering points at practical speeds on existing GPU architectures (optimization strategies are discussed in Section 5.4).

Shown in Figure 5.3, our conservative approach computes \( s_{mv} \) for the current multi-view scenario, and then uses it to configure the appropriate density of texels per unit polygonal area during rasterization. We accomplish this by a) configuring the rasterizer’s viewport resolution to the view with the most demanding level of detail requirements, b) finding the shortest geometric distance from a polygon’s surface to the set of all relevant view centers, and c) adjusting the minimum geometric distance by a perspective distortion factor: the maximum perspective distortion found in the most demanding view. We discuss the perspective distortion factor in Section 5.2.2. The multi-view sampling rate \( s_{mv} \) is given by Equations 5.2 and 5.3:

\[
    d_v = \text{distance}(v, p) \tag{5.2}
\]

\[
    s_{mv} = \forall v \in V | \min(s_{mv}, w \times d_v) \tag{5.3}
\]
where $V$ is the set of all view centers, $v$ is a view center, $p$ is a polygon, $w$ is the perspective distortion factor, the function $\min$ takes the minimum of two values, and the function $\text{distance}$ computes the minimum geometric distance from a point to a polygon. The point-to-polygon distance is efficiently computed with a two dimensional projection \[40\].

Once $s_{mv}$ is computed, we apply it to each polygon in the Geometry Shader (prior to rasterization) as part of $T_{\text{persp}}$ or $T_{\text{ortho}}$, a perspective or orthographic projection transform. When using a perspective projection, $s_{mv}$ represents the minimum distance between the polygon and the VIR near plane, which we implement as a translation along the z-axis. For an orthographic projection, $s_{mv}$ scales the projected size of the parallel polygon by manipulating the area of the near plane. Each projection uniformly affects the polygonal sampling rate; however, $T_{\text{ortho}}$ is preferred in most cases since the distribution of samples is constant across the VIR near plane. These transformations are given as $4 \times 4$ matrices in Equation 5.4:

$$
T_{\text{persp}} = \begin{bmatrix}
\frac{2a_r}{\tan(fov_y)} & 0 & 0 & 0 \\
0 & \frac{2}{\tan(fov_y)} & 0 & 0 \\
0 & 0 & \frac{f}{f-n} & s_{mv} \\
0 & 0 & \frac{-n \times f}{f-n} & 1
\end{bmatrix}, \quad T_{\text{ortho}} = \begin{bmatrix}
\frac{1}{s_{mv}} & 0 & 0 & 0 \\
0 & \frac{1}{s_{mv}} & 0 & 0 \\
0 & 0 & \frac{f}{f-n} & 1 \\
0 & 0 & \frac{-n \times f}{f-n} & 1
\end{bmatrix}
$$

where $a_r$ is the aspect ratio of the view, $fov_y$ is the vertical field of view angle, $n$ is the distance to the near projection plane, and $f$ is the distance to the far projection plane.

### 5.2.1 Visualizing Sampling Rate

Due to the view agnostic nature of view independent rasterization’s polygonal sampling, it is useful to visualize how the point samples generated for a polygon map to a given view / projection combination. We define ideal sampling as the set of polygonal samples that map exactly one sample to each texel covered by the polygon. This definition caters to the strict performance requirements of real-time applications, since a single sample per texel is the minimum necessary to render complete images. This form of ideal sampling is also by definition view dependent, and is the output of standard rasterization. VIR achieves ideal sampling for an entire polygon in the special case that makes VIR possible on the GPU: when the polygon is parallel to the target view plane. In all other cases, VIR’s polygonal sampling also includes a combination of undersampling and/or oversampling. Undersampling occurs when a texel is covered by a polygon, but no point sample maps to the texel during splatting. Oversampling occurs when multiple point samples of a polygon map to the same texel covered by the polygon.

We compute $s_t$, the type of sampling (ideal, under, or over) exhibited by our multi-view sampling rate for each texel of a target view, by comparing (1) $n_{ps}$, the number of polygonal
samples generated by VIR that map to each texel using point-based rendering with \( n_{rs} \), the number of samples at each texel of an image rendered using regular rasterization. This is given by Equation 5.5.

\[
s_t = n_{ps} - n_{rs}
\]

\( n_{ps} \) is computed by splatting points into the target view and following the process outlined in Section 3.2.2, with the addition of a per texel counter that is incremented for each point that maps to the texel. \( n_{rs} \) is simply the view’s per texel depth complexity, since standard rasterization places a single sample at every texel. \( n_{rs} \) is computed by either incrementing the GPU stencil buffer during rasterization or by using additive blending after rasterization.

---

**Figure 5.4:** Sampling type visualizations. a) the original texture mapped geometry. b) the basic mode displaying a single color for ideal sampling (black), undersampling (red), and oversampling (cyan). c) an extended visualization mode for oversampling using a single hue and normalized input values. d) a second extended oversampling mode, using 7 perceptually distinct hues and an additional high contrast color to indicate high oversampling rates.
Figure 5.4 demonstrates our sampling type visualization and its various modes for a simple texture mapped plane composed of two triangles. Figure 5.4a shows the original texture mapped polygon with the mesh wireframe overlaid in green. Figure 5.4b demonstrates a basic visualization mode that represents each type of sampling with a single color. An $s_t$ value of zero indicates ideal sampling, and the texel is colored *black*. When $s_t$ is negative undersampling occurs, and the texel is colored bright *red*. Finally, a positive $s_t$ value represents oversampling, and the texel is colored *cyan*. Since our general strategy is to produce a high density point cloud tailored for many views and avoid complex reconstruction, each polygon’s multi-view sampling rate is biased towards oversampling. Due to this, we include two extended visualization modes that illustrate oversampling in more detail. The *Normalized Single Hue Mode*, shown in Figure 5.4c, scales the intensity of a single hue based on the normalized quantity of oversampling present at each texel. The *Multi-Hue Mode*, shown in Figure 5.4d, uses seven perceptually distinct hues to display the first seven levels of oversampling. We determine perceptually distinct hues using ColorBrewer [34]. An additional high contrast color is added at the top of the scale to make excessively high oversampling rates stand out.

### 5.2.2 Perspective Distortion

Generating view-independent samples of a polygon using an $s_{mv}$ value determined by the minimum geometric distance from a view center to a polygon ($d_v$) is not sufficient to guarantee watertight rendering across all parts of a view. Using our sampling visualization, Figure 5.5 demonstrates how a polygon sampled with VIR using only minimum geometric distance exhibits

<table>
<thead>
<tr>
<th></th>
<th>27,900 raster samples</th>
<th>34,565 raster samples</th>
<th>71,906 raster samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>a)</td>
<td>29,070 VIR samples</td>
<td>29,070 VIR samples</td>
<td>29,070 VIR samples</td>
</tr>
<tr>
<td>b)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.5: VIR sampling using the minimum geometric distance from a view center to each polygon. a) Parallel geometry is sampled ideally. b) Undersampling occurs as the polygon rotates and moves away from the view center. c) Severe undersampling occurs with further rotation and movement towards the view extent.
a) ideal sampling when polygons are parallel to the view, but suffers from b) undersampling when polygons rotate and move away from the view center, and 3) severe undersampling when polygons rotate and move near the edge of the target view. This undersampling problem exists because *perspective projection distorts* three-dimensional space across a two-dimensional plane. The convergence of lines of sight in perspective projection *non-uniformly alters* the size and shape of a polygon within a view, and this distortion increases towards the view extent [11]. Notice how the rectangle in Figure 5.5a becomes significantly larger and a different shape as the view is rotated in Figures 5.5b and 5.5c. Reported in each panel of Figure 5.5, standard view-dependent rasterization produces more polygonal samples to accommodate this difference in size and shape; however, the number of samples VIR generates in each example may remain identical when using an $s_{mv}$ value equivalent to the minimum geometric distance. In this example, the polygon’s change in size and shape is caused by a view rotation, and the geometric relationship of the view center and polygon are unchanged. As a consequence, in general, the minimum geometric distance from a view center to a polygon *guarantees* watertight sampling only at the exact center of a view (where the view direction intersects the image plane).

We account for the distortion caused by perspective projection by modifying the minimum geometric distance from a view center to a polygon with $w$, the *perspective distortion factor*. This decreases the value of $s_{mv}$ before VIR, which uniformly increases polygonal sampling and matches the worst-case sampling rate present at a view’s point of maximal perspective distortion in the two viewing dimensions ($x$ and $y$). Shown on the left of Figure 5.6, perspective distortion in one dimension is illustrated geometrically as the difference between the perspective projection of a line onto a circular arc. Shown on the right of Figure 5.6, the distortion factor in two dimensions is the difference between a planar surface (the image plane) and a unit

![Figure 5.6: The geometry of perspective projection distortion.](image-url)
sphere centered about the view center (defined as artificial and natural perspective projection respectively by Da Vinci [16]). The maximum distortion in each quadrant of the image plane occurs along the diagonal \( k \), and the point of maximum 2-D distortion lies at the intersection of \( k \) and the edge of the view frustum, \( m_{xy} \). Similar to the one dimensional case, the maximum distortion in two dimensions is the squared length of the right triangle hypotenuse formed by edges \( r \), \( k \), and \( m_{xy} \). The perspective distortion factor \( w \) is given by Equations 5.6 and 5.7:

\[
|m_{xy}| = \sqrt{|r|^2 + |k|^2} \tag{5.6}
\]

\[
w = \frac{1}{|m_{xy}|^2} \tag{5.7}
\]

where \( r \) is the radius of a unit circle centered at the view center and the distance from the view center to the image plane along the view direction. Perspective distortion changes radially across the image plane, increases along the diagonal \( k \) towards the image extent, and is shown as concentric circles inscribed on the image plane in Figure 5.6.

Figure 5.7 applies the perspective distortion factor in VIR for the example scene in Figure 5.5. To ease in the detection of undersampled pixels, we disable the visualization of oversampling. Pixels colored black represent ideal sampling or oversampling, and pixels colored red represent undersampling. Unlike Figure 5.5, no undersampling exists due to the addition of the perspective distortion factor when computing \( s_{mv} \). VIR’s conservative approach now replaces undersampling with oversampling by generating a larger number of polygonal samples. In this example, the high oversampling shown is an unusual extreme case used for illustration purposes.

![Figure 5.7: VIR sampling using Equation 5.3 and the perspective distortion factor \( w \). Visualization of oversampling is disabled to ease in the detection of undersampling. By applying the perspective distortion factor when computing \( s_{mv} \), no undersampled (red) pixels exist.](image)
5.3 Sampling Projected Polygons in VIR

Standard rasterization on existing GPU hardware guarantees watertight view-dependent sampling of polygonal meshes. Since adjacent polygon edges project to nearby areas in an image of a single view, graphics hardware employs a fixed-point sub-pixel snapping scheme coupled with specific rules for rasterization to ensure solid surfaces are represented without holes once rasterized. For example, this approach handles tricky cases such as highly tessellated meshes composed of many sub-pixel projected polygons (polygons with projected area occupying a single pixel or less) that do not intersect a pixel center.

This approach, and the subsequent watertight guarantee, is no longer valid in view independent rasterization since each polygon is processed in isolation and agnostic of view. As a result, it is possible for some types of polygons to 1) fall between pixel centers on the VIR raster grid and not be sampled at all, or 2) be sampled insufficiently due to aliasing between the view-dependent and view-independent raster grids. We tackle these problems by classifying polygons based on their projected size on the VIR raster grid and adapting sampling using a combination of procedural, stochastic, and conservative rasterization techniques.

5.3.1 Sub-Pixel

Shown in Figure 5.8a, we define a sub-pixel polygon in VIR as any projected polygon whose longest edge is less than one unit in length (of horizontal or vertical interpixel distance) on the VIR raster grid. Using standard rasterization rules, sub-pixel polygons receive a maximum of one sample if they cover a pixel center; however, they will often fall between pixel centers and receive no samples. Figure 5.8b demonstrates a common VIR scenario where a sub-pixel polygon is placed in the middle of the raster grid (when the polygon centroid is used in Equation 5.1), but it is too small to intersect a nearby pixel center. A simple solution to this problem is to
enable hardware *conservative rasterization*. The drawback of this approach, besides the lack of ubiquitous hardware support, is unnecessary oversampling since four samples are generated per polygon even though a single sample is sufficient. An improved alternative approach that also uses hardware rasterization, modifies the translation in Equation 5.1 to place the sub-pixel polygon’s centroid at any nearby pixel center. Shown in Figure 5.8c, this approach guarantees all sub-pixel polygons always receive a sample, avoids unnecessary additional sampling, and does not require conservative rasterization.

Since sub-pixel polygons are often part of geometrically complex surface meshes, several adjacent polygons may project to the same texel of a target view’s image. In this situation, the texel represents all of the sub-pixel polygons with a single color value. Accurate methods perform a weighted average of each polygon’s contribution to compute the final texel color; however, real-time applications typically opt for faster alternatives such as simply choosing one sample from the set of available samples. As a result, it is often *not* necessary for every sub-pixel polygon to produce a sample. VIR leverages this insight by procedurally generating samples of sub-pixel polygons in the Geometry Shader and stochastically choosing whether to store these samples in the point data structure. The flexibility of programmable sampling allows for a wide range of statistics-based sampling methods to be used when determining the probability that a generated sub-pixel polygonal sample will be retained. For example, the probability function may be based on a binomial distribution related to the comparison of a sub-pixel polygon’s projected area and the area of a pixel on the VIR raster grid. The binomial function ensures, with a high probability, that at least one sub-pixel polygon sample will be stored to the point data structure. With this kind of procedural approach, sample generation itself is fully programmable, but is still efficient – despite not using the dedicated raster hardware – since *at most* one sample is produced per polygon.

### 5.3.2 Peri-Pixel

Shown in Figure 5.9a, we define a *peri-pixel* polygon in VIR as any projected polygon whose longest edge is greater than or equal to one and less than two units on the VIR raster grid. Peri-pixel polygons are guaranteed to cover at least one pixel center, and at most three pixel centers using standard rasterization rules. This sampling variability introduces the potential for aliasing and undersampling between the VIR raster grid and the target view’s raster grid. To tackle this problem, we procedurally sample peri-pixel polygons in the Geometry Shader after VIR transformation and projection – similar to procedurally sampled sub-pixel polygons – and store at least three samples per polygon. Due to the flexibility of procedural sampling, the sample positions of peri-pixel polygons are fully programmable and can be adapted to application specific demands. Our conservative implementation stores each polygon vertex *and*
the polygon centroid. Although an additional sample is included, this is an improvement over the seven samples that are generated by conservative rasterization. To improve performance, statistics-based methods may also be applied here, much like sub-pixel polygons. For example, the probability of storing each vertex may be determined by the peri-pixel polygon’s area, the distance of a given vertex to the centroid, and many other criteria.

5.3.3 Supra-Pixel

Shown in Figure 5.9b, we define a supra-pixel polygon in VIR as any projected polygon whose shortest edge is greater than or equal to two units in length on the VIR raster grid. Supra-pixel polygons typically cover many pixel centers on the raster grid, and are most efficiently sampled with rasterization hardware due to the large number of samples that must be generated. For example, the polygons used in Figures 5.5 and 5.7 are classified as supra-pixel. Although sampling of these polygons is straightforward compared to the procedural methods used for other polygon types, aliasing between sample placement on the VIR and view-dependent raster grids is possible, especially if less conservative polygon sampling rates are employed. Shown on the left of Figure 5.10, this aliasing manifests as undersampling at polygon edges. The top of Figure 5.10 uses the Multi-Hue Visualization Mode to display the mix of oversampling and undersampling across a plane composed of two supra-pixel polygons. The bottom of Figure 5.10 disables the visualization of oversampling to highlight the areas of undersampling. We eliminate the edge undersampling of supra-pixel polygons by enabling conservative rasterization on the VIR raster grid. The top right of Figure 5.10 demonstrates how this approach trades undersampling for oversampling. However, since the number of texels covered by the polygon’s interior typically far exceeds that of the edges, conservative rasterization is a suitable solution to improve edge
Figure 5.10: Edge aliasing and undersampling of supra-pixel polygons. Top: the Multi Hue Visualization Mode. Bottom: the oversampling visualization is disabled to ease in the detection of undersampling artifacts. Left: undersampling caused by aliasing between the VIR and view-dependent raster grids. Right: undersampling is eliminated with conservative rasterization.

Especially large supra-pixel polygons that also require high sampling rates pose a unique problem in VIR. In view-dependent rendering, large polygons receive variable sampling across a view and are typically clipped at the edges of the view frustum. VIR does not perform clipping and instead samples the entirety of these large polygons at the highest rate necessary. It is possible that achieving the appropriate VIR sampling rate requires a texel density that VIR cannot accommodate. In these cases, we procedurally subdivide the large polygon into several smaller polygons. We apply the VIR transform and projection matrices to the new polygons, and then sample as described previously. Subdividing large polygons at runtime is not ideal, and by design the connectivity information of large polygons is most beneficial to the design of standard rasterization. When the scene content contains primarily large polygons, multi-view rasterization is a more appropriate choice than VIR or point-based rendering.
5.4 Optimization Strategies

In this section, we describe several optimization strategies that may improve the performance of practical View Independent Rasterization and multi-view point rendering implementations. Each solution focuses on optimizations that existing GPU architectures may benefit from. We categorize our solutions by those that (1) optimize the size of the point data set or (2) improve the performance of point rendering without modifying the input point data. We explore the space of optimization ideas to inspire future research directions in View Independent Rasterization and multi-view point-based rendering. As a result, many of these concepts are in the initial stages of development and have not yet been implemented or fully evaluated.

5.4.1 Point Generation

The sampling approach we present in the previous sections guarantees hole-free results in all scenarios at the cost of overly conservative sampling. We present solutions to reduce oversampling by optimizing the per polygon perspective distortion factor, performing point rendering with smaller point data sets, and generating multiple point clouds of varying levels of detail.

**Optimal Perspective Distortion Factors**

Although the default perspective distortion factor is fast to compute for each view (it is a constant value), it assumes all polygons will cover a texel at the view’s point of maximum 2D distortion. Many polygons project near the center of a target view, and are subject to only small amounts of distortion. Therefore, by tailoring the default perspective distortion factor to a view’s worst-case sampling scenario, some polygons are sampled at a rate several times higher than the minimum rate necessary to guarantee watertightness for a specific set of views. The result is a point data set often denser than necessary, which has the potential to significantly increase the amount of compute resources, memory bandwidth, and atomic operations utilized during point rendering. Introducing additional computation to find the optimal perspective distortion value for each polygon-view pair may yield significant performance improvements by dramatically reducing the quantity of point samples generated during VIR.

To compute the optimal perspective distortion for a polygon-view pair, we (1) project the polygon vertices onto the view plane, (2) find the maximum distance from the image center to the projected vertices, and (3) compute the distortion factor $w$ using Equations 5.6 and 5.7 where $m_{xy}$ becomes the hypotenuse of the optimal right triangle formed by $r$, the unit length view direction vector, and $k$, the vector from the image center to the most distant projected polygon vertex. This additional processing is performed every frame to tailor the perspective distortion factor per polygon as animation may change the polygon’s projected location. This
approach is a classic example of balancing computation and memory resources, and we are encouraged to test this strategy in a practical setting to quantify how beneficial it may be to performance. We expect this approach to reduce the oversampling of polygons that have large projected surface areas, while the additional computation required to perform the vertex projections will not be advantageous for polygons with small projected surface areas.

**Approximating Optimal Perspective Distortion Factors**

In some cases, the cost of the additional computation to find the optimal perspective distortion factor may not actually improve performance. As mentioned previously, we observe that the highest amounts of perspective distortion exist at the image extents, and distortion decreases substantially towards the image center. We can take advantage of this observation and avoid the cost of additional computation by approximating the optimal perspective distortion factor. This simple approach produces a similar result that decreases the size of the point data, reduces oversampling, and optimizes point rendering.

Since the areas of a view that exhibit the highest perspective distortion often cover a small percentage of the output image, we impose an *upper limit* for the perspective distortion factor that is less than the maximum 2D distortion. By creating points in VIR using a less aggressive distortion factor, many fewer points are generated and watertight reproduction of polygons is still guaranteed for a majority of the image. By reducing the VIR perspective distortion factor, oversampling in the center of the view decreases substantially, at the cost of introducing the possibility of undersampling at the image extents. This is illustrated in Figure 5.11, where we show the multi-hue visualization results for a test plane in several locations on the image plane. On the left, VIR uses the default perspective distortion factor for a 90° view, and high levels of oversampling exist in the center of the image. In the middle of Figure 5.11, the same scene is rendered where VIR uses a less aggressive distortion factor (\(|m_{xy}| = \sqrt{2}\) instead of \(\sqrt{3}\)). The number of points decreases by 2.24×, oversampling decreases dramatically in the center of the image, and the majority of the image is free of undersampling. A small amount of undersampling does exist in the upper corner of the image. We highlight this area and magnify it in the inlay on the right of Figure 5.11. Since undersampling artifacts are confined to only a few pixels in size, we eliminate undersampling by *adapting and extending* the influence of a point sample to multiple texels *only when* it projects to areas of the image that exhibit higher perspective distortion than the chosen upper limit. The undersampling free result of this approach is shown on the right of Figure 5.11 and is magnified in the inlays. Although this approach is not strictly correct, since point samples are written to texels they might not actually intersect, the visual impact is limited to only a few texels near the image extents; and major performance improvements are possible due to the decrease in atomic operations executed.
Figure 5.11: An illustration of our optimal perspective distortion factor approximation strategy. This low polygonal complexity / high projected area example exhibits extreme oversampling that is unusual in practice, but is ideal for illustration purposes. Left: VIR using the default perspective distortion factor. Notice the high oversampling at the center of the image. Middle: VIR using a less aggressive distortion factor. The number of points generated by VIR is reduced and oversampling is significantly decreased, but undersampling is introduced in the upper right corner of the image. Right: adapting the influence of a point to additional surrounding pixels in areas of high perspective distortion eliminates the undersampling problem while retaining a low overall amount of oversampling across the image.
Discrete Level of Detail

When the projected size of a polygon varies greatly across a set of views, the minimal multi-view rate at which the polygon must be sampled also varies significantly. Since View Independent Rasterization traverses and samples polygons only once, the denser point set produced for the most demanding view is also used to render the less demanding view(s). The higher resolution representation of the geometry is wasteful when rendering at the lower detail level of a less demanding view. This observation reveals another optimization opportunity.

This oversampling problem is similar to classic level of detail (LoD) problems in traditional polygonal rasterization. Instead of matching point cloud density to a view’s image resolution, polygonal LoD techniques attempt to tailor the complexity of polygonal meshes to closely match the detail level the output image’s resolution is capable of displaying for objects at a given distance or projected size. A widely used LoD solution adapts polygonal level of detail by constructing several discrete versions of a mesh, typically offline, and determining the version to render at run-time based on the distance of the mesh to the view (and often additional factors).

Inspired by this approach, we can solve our sampling variability problem by creating multiple discrete point-based levels of detail during VIR and choosing the appropriately dense version just before point rendering. Although we require several point clouds of the original geometry, we can still generate these in VIR and only traverse the geometry once. We create multiple point-based LoDs in a single geometry pass by duplicating the input polygon in the Geometry shader and rasterizing each duplicate with different multi-view sampling rates. This is illustrated in Figure 5.12. This process replicates the input polygon a limited number of times (five replications or less performs well on current GPUs), and stores each LoD in a separate linear buffer. Before point rendering, we compute the distance from the current view’s eye location to a bounding volume containing the polygon and determine the appropriate discrete point cloud.

![Figure 5.12](image-url)  
Figure 5.12: An illustration of our modified VIR implementation in the graphics pipeline that generates multiple discrete point-based LoDs while still traversing the geometry only once.
to render based on the distance. This is just one example of how to construct multiple LoDs with VIR. A variety of alternative approaches are possible, since VIR sampling is exceptionally fast and flexible due to the combination of fixed function hardware and programmable shaders. For example, point LoDs may be stored progressively or in spatial data structures such as an octree or bounding volume hierarchy.

We are also interested in pursuing the combination of VIR with Geometry Images [31] as a mechanism to achieve multiple pre-filterable discrete LoDs. In this scenario, we envision using VIR to transform and pack polygons onto the raster grid at runtime. A view independent image is then rasterized by the hardware, and LoDs are produced by simply mipmapping the image. Using VIR this way also has potential intersections with object space shading, since polygons are discretized independent of view along with all the required shading data [14] [10].

5.4.2 Point Rendering

When rendering a high density point set to a target view’s image, it is common for multiple point samples to project to the same texel. Since only one of these points (usually the closest) must be stored for each texel to avoid holes, we can theoretically avoid much of the memory bandwidth associated with atomically writing and z-buffering additional points. While the previous section focused on reducing oversampling by tailoring polygonal sampling to optimize the size of the point data; in this section, we describe strategies to reduce memory traffic and increase point rendering performance without modifications to the point set.

Our basic point rendering strategy, described in Chapter 3, renders a single point sample with a single GPU Compute thread. By leveraging the insight that multiple samples may project to the same image location, our general point rendering optimization approach modifies the threading model to instead (1) load several point samples per thread and (2) perform various types of processing to reduce the number of memory bandwidth intensive operations (e.g. atomic read-modify-write) each GPU Compute thread performs.

Spatial Coherence Point Sample Culling

A simple optimization based on this concept has each GPU thread load multiple point samples into its registers, project each point sample into a target view, and compare the destination texel location of the projected points in an all pairs fashion. If any of the projected point samples have the same destination texel, then it is only necessary to write the closest point sample of the samples in the matching set. Since most modern GPU rasterization hardware units process groups of pixels, or tiles, during rasterization, Pixel shaders invoked after View Independent Rasterization are roughly executed in polygon submission order. This means point samples that are nearby on the plane of a polygon (in world space) are stored by Pixel shaders at similar
times. As a consequence, these point samples are also stored in similar memory locations in our linear point data structure. The spatial coherence in both world space and memory space increases the probability that a GPU Compute rendering thread loading multiple points will process points that may project to the same destination texel. This is illustrated in Figure 5.13.

Storing points in spatial data structure makes leveraging the spatial coherence of point samples during rendering even easier. For example, GPU Compute threads can load and render the contents of data structure leaf nodes. These nodes contain spatially coherent points by definition, and are more predictably organized than the linear Append/Consume buffer we currently employ. Regardless of the data storage solution we use, leveraging spatial coherence during point rendering trades additional computation for a reduction in memory bandwidth traffic. This trade is a good match for the capabilities of existing GPUs since compute capability far exceeds memory bandwidth. In our initial tests we see a 25% reduction in atomic read-modify-write operations when loading two points per GPU Compute thread. Diminishing returns are reached at 4 points per thread; however, this is specific to the linear buffer and distribution of polygon sizes in these initial tests. We expect spatial data structures to be able to take advantage of higher numbers of points per thread due to their more predictable organization.

**Just-in-Time Point Sample Culling**

Spatial Coherence Point Sample Culling is an implementation idea that fits within a broader category of point rendering optimization strategies that we call *Just-in-Time (JIT) Point Rendering*. These strategies render points, but make adjustments by culling, amplifying, filtering, and writing points to images at the last available moment. In the case of Spatial Coherence Culling, we perform additional computation at point render time to determine if points can be culled before writing to the output image. Several other similar strategies may be used to
accomplish just-in-time culling. For example, the distance between projected point samples may be compared to the texel density of the output image to determine the amount of potential oversampling at a texel. Inspired by existing statistics based point rendering methods [42, 43], it may be possible to use probabilities and binomial distribution functions to predict and reduce oversampling without modification to the point data before rendering begins.
Chapter 6

Multi-View Effects with View Independent Rasterization

In this chapter, we demonstrate and discuss the practical application of View Independent Rasterization (VIR) and multi-view point-based rendering to challenging multi-view effects in real-time. Similar to our evaluation of View Warped Soft Shadows, we implement, evaluate, and report the performance and quality results of rendering soft shadows using VIR coupled with our multi-view point-based rendering approach on existing graphics hardware. As before, Multi-View Rasterization (MVR) serves as the best proxy for a ground truth solution that real-time applications can consider; therefore, we compare VIR results directly to MVR. Due to VIR’s use of the Geometry Shader, which is well known to limit peak GPU performance in some scenarios [2], we assess VIR’s ability to process complex geometry by testing assets that are significantly more complex than those used in our VWSS evaluation. Although many optimization strategies related to VIR and multi-view point rendering are presented in Chapter 5, not all are implemented – nor are they relevant to the workloads we test – so we also include a discussion of the limitations of our implementation.

When considering visual effects beyond shadows, VIR is capable of producing useful polygonal samples for arbitrary viewpoints unlike the view inspired design of VWSS. We discuss the practical application of VIR to additional visual effects including ambient occlusion, defocus blur, environment mapped reflections, and diffuse global illumination. Since the design of existing graphics hardware is heavily tailored to object-order single-view polygonal rasterization, barriers caused by the hardware’s architecture make applying VIR and multi-view point rendering to z-buffered color images challenging. We identify, discuss, and provide work-around solutions to these obstacles as a stopgap until GPU architectures and APIs are updated to provide first-class support for important operations and data types that are currently absent.
### 6.1 Multi-View Soft Shadows

We demonstrate View Independent Rasterization coupled with multi-view point-based rendering by applying it to the rendering of soft shadows cast from an area light source. This new soft shadowing algorithm performs multi-view point rendering similarly to VWSS, but builds a *high density view independent point cloud* of the complex geometry using View Independent Rasterization. We set the field of view for each area light view to 45\(^\circ\) to maximize the texel density of each depth buffer and support shadows with fine details. As a result of the relatively small field of view, the maximum perspective distortion in each view is minimized. Our implementation approximates the optimal perspective distortion factor for the light sample views using the strategy discussed in Chapter 5. We use simple point rendering kernels implemented in Compute shaders *without* optimizations such as spatial coherence culling, just-in-time culling, or discrete levels of detail. Figure 6.1 illustrates the flow of data through our implementation. Figure 6.1 depicts point rendering with 32 views per point traversal since this optimizes memory bandwidth utilization and avoids bottlenecks during parallel view rendering on most existing GPUs; however, our implementation supports an arbitrary number of views per traversal.

![Figure 6.1: As with VWSS, multi-view rendering using View Independent Rasterization restructures depth buffer computation to improve point reuse and parallelization on GPUs.](image)

<table>
<thead>
<tr>
<th>Step 1 - Point Generation</th>
<th>View Independent Rasterization and Multi-View Point Rendering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geometry Traversal</td>
<td>View Independent Rasterization</td>
</tr>
<tr>
<td>Occluder Geometry</td>
<td>View Independent Point Representation</td>
</tr>
<tr>
<td>1</td>
<td>Parallel Point Representation</td>
</tr>
<tr>
<td>Pixel Shader (p)</td>
<td>Append / Consume Buffer</td>
</tr>
<tr>
<td>Light View Transforms ([1 - 32])</td>
<td>Depth Buffer ([1])</td>
</tr>
<tr>
<td>Light View Transforms ([33-64])</td>
<td>Depth Buffer ([32])</td>
</tr>
<tr>
<td>Depth Buffer ([33])</td>
<td>Final Lighting</td>
</tr>
<tr>
<td>Depth Buffer ([64])</td>
<td></td>
</tr>
</tbody>
</table>

---

**Figure 6.1:** As with VWSS, multi-view rendering using View Independent Rasterization restructures depth buffer computation to improve point reuse and parallelization on GPUs.
Our evaluation methodology is similar to our assessment of VWSS. We capture performance data as an average of 1,000 frames of execution, all depth buffers use 32-bit floating point precision, and depth buffers are 1024\(^2\) resolution. Our experimental environment again uses Direct3D 11.1 running on an Intel i7-4790k @ 4.0 GHz with a NVIDIA Maxwell Titan X GPU, but instead uses Windows 8.1. We observe no significant difference in runtime performance between Windows versions 8.1 and 10. Visual quality is again evaluated using RMSE as a numerical measure and HDR-VDP2 as a perceptual measure. The most important difference in our evaluation of VIR is the use of high complexity geometry as occluders in the test scenes. The number of polygons in each test scene is reported in Tables 6.1 and 6.3, and images of each are provided in Figure 6.4.

### 6.1.1 Performance

First, we evaluate the performance of VIR’s transformation of polygons into points (Step 1 in Figure 6.1). We run the algorithm on multiple workload sizes by varying the geometric complexity of each test scene’s occluder geometry. The geometric complexity of test models starts around 50,000 polygons and increases to 2 million polygons. We report results in Table 6.1 and include the number of polygons processed, the number of points generated, the storage required for the generated points, and the GPU time required to perform the transformation. We include the storage required by the generated points since all stages of the algorithm (loading polygons, passing data between shader stages, and storing point samples in global memory) may potentially contend for limited bandwidth resources during execution. We capture GPU times of VIR tailoring point sampling to a large set of views (128), and a small set of views (8).

In Table 6.1, the values in parentheses report the small set’s results. As demonstrated by the

<table>
<thead>
<tr>
<th>Triangles</th>
<th>Points</th>
<th>Point Storage</th>
<th>GPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>53,715</td>
<td>83,989 (83,261)</td>
<td>0.96 (0.95)</td>
<td>0.46 (0.24)</td>
</tr>
<tr>
<td>106,973</td>
<td>125,136 (124,420)</td>
<td>1.43 (1.42)</td>
<td>0.68 (0.33)</td>
</tr>
<tr>
<td>256,621</td>
<td>229,840 (229,342)</td>
<td>2.63 (2.62)</td>
<td>0.98 (0.69)</td>
</tr>
<tr>
<td>502,667</td>
<td>369,328 (368,983)</td>
<td>4.22 (4.22)</td>
<td>1.62 (1.14)</td>
</tr>
<tr>
<td>1,023,609</td>
<td>563,878 (563,665)</td>
<td>6.45 (6.45)</td>
<td>2.97 (1.99)</td>
</tr>
<tr>
<td>2,005,071</td>
<td>599,916 (599,865)</td>
<td>6.87 (6.87)</td>
<td>5.41 (3.21)</td>
</tr>
</tbody>
</table>

Table 6.1: GPU performance and memory cost of VIR point generation for 128 views (and 8 views in parentheses) for geometry of varying complexity. Point storage is measured in megabytes (MB). GPU time is measured in milliseconds (ms).
results, tailoring point sampling to fewer light views improves performance. This is expected since computing the VIR transform for many views generates a larger number of instructions per shader due to the increased number of distance computations executed. This in turn requires more GPU registers per thread, and how effectively a shader thread is parallelized across GPU processor cores is heavily influenced by the number of registers it uses. Although tailoring for fewer views is fastest, VIR still turns in exceptional performance results when processing the larger set of 128 views. VIR processes the most complex test occluder of two million polygons, and tailors nearly 600,000 points to 128 views in only 5.41 milliseconds.

Next, we evaluate the performance of various multi-view point rendering implementations. As with VWSS, our multi-view implementation projects all points into all views, but the point data is now produced by VIR. A main goal of this analysis is to compare the performance of multi-view point rendering executed in the graphics pipeline against execution in GPU compute. Since the GPU has evolved to be more flexible than ever before, which approach is best is not immediately clear. Even within the graphics pipeline, multi-view point rendering may be accomplished a number of ways by using instancing features and the Geometry shader. We have tested each combination of instancing and amplification features in the graphics pipeline to render points to multiple views in a single traversal. Somewhat surprisingly, none of these features actually improve multi-view point rendering performance in a meaningful way.

Table 6.2 reports performance results for our fastest GPU Compute and graphics pipeline implementations. Each rendering method is labeled according to the categorization presented.

<table>
<thead>
<tr>
<th>Rendering Method</th>
<th>Depth Buffers Rendered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial w/ Accumulation Buffer (lowest memory)</td>
<td>8 16 32 64 128 256</td>
</tr>
<tr>
<td>Compute, Method 3</td>
<td>0.73 1.49 3.11 5.79 13.00 24.42</td>
</tr>
<tr>
<td>GFX, Method 4</td>
<td>1.89 3.87 7.37 14.76 29.43 58.88</td>
</tr>
<tr>
<td>Serial w/o Accumulation Buffer</td>
<td>0.49 1.00 2.00</td>
</tr>
<tr>
<td>Compute, Method 1</td>
<td>3.76 8.60 16.20</td>
</tr>
<tr>
<td>GFX, Method 2</td>
<td>1.44 2.87 5.71 11.58 23.61 47.68</td>
</tr>
<tr>
<td>Fully Parallel w/o Accumulation Buffer</td>
<td>0.33 0.60 1.16</td>
</tr>
<tr>
<td>Compute, Method 9</td>
<td>8.38 36.62 307.15</td>
</tr>
<tr>
<td>GFX, Method 10</td>
<td>1.97 5.05 11.39 22.78 45.56 91.12</td>
</tr>
</tbody>
</table>

Table 6.2: Performance results of various multi-view point rendering methods. Each method operates on 600,000 points. Fastest times highlighted in blue. Implementations in GPU Compute are faster in all cases. GPU times are measured in milliseconds (ms).
in Chapter 3. We directly compare graphics pipeline and GPU Compute implementations, as well as implementations that utilize accumulation buffers for applications that must minimize memory consumption. We report results for the two ends of the spectrum: a) *serial*: rendering one view per point traversal (Methods 1, 2, 3, 4); and b) *fully parallel*: rendering all views in a single traversal of the point set (Methods 9 and 10). Due to a limitation of the Geometry shader, Method 10 is only able to render a maximum of 32 views in a single point traversal. Additional views are not currently possible; however, we include values (in grey) for 64, 128, and 256 views extrapolated from a linear performance curve should this limitation be removed.

Methods using accumulation buffers always perform slower than those without due to the additional memory intensive read and write operations executed during the accumulation process. Furthermore, in all cases *GPU Compute implementations execute faster* than those using the graphics pipeline. At first glance, this is an unexpected result given the additional fixed function hardware the graphics pipeline uses for loading and caching vertices. However, we observe similar results in VWSS as shown in Figure 4.8. We theorize the performance delta between the graphics pipeline and GPU Compute is a result of thread scheduling, the pipeline requiring points to pass through the rasterizer stage, and those points being rasterized as degenerate triangles (in order to invoke Pixel shaders). Although we are unable to confirm our theory due to the proprietary nature of GPU rasterizers, this type of additional work fits the profile of the observed performance reduction.

Table 6.3 presents the performance results of our full soft shadowing algorithm using View Independent Rasterization and our fastest GPU Compute multi-view point rendering implementation. We organize our implementation as a deferred renderer, since those are popular in cutting edge game engines. Table 6.3 reports the performance of each step of the deferred algorithm, and also includes the performance of each step of Multi-View Rasterization (MVR) in parentheses for direct comparison. We test a range of geometric complexities, including multiple “Lucy” models of various levels of detail. The VIR sampling step is highlighted in red. We report the Total Time to compute an image using either algorithm in two columns, Total Time *Static* and Total Time *Dynamic*. Unlike MVR, VIR is able to sample the geometry once and *reuse the point data* to render depth buffers if the geometry is not animated. The Total Time *Static* column is the total time required to render an image when leveraging this reuse, and enables even greater performance improvements. The Total Time *Dynamic* column is a fully dynamic comparison of VIR and MVR that recomputes the point data every frame. The performance improvements of our rendering approach are highlighted in blue. Even in the fully dynamic case, soft shadows cast by complex geometry are rendered up to $8 \times$ faster than MVR.
Table 6.3: GPU performance results of rendering soft shadows using View Independent Rasterization (VIR) coupled with our multi-view point-based rendering approach (compared to serial Multi-View Rasterization in parentheses). Both techniques compute soft shadows using 128 depth buffers of $1024^2$ resolution. The VIR sampling step is highlighted in red. Unlike MVR, VIR is able to sample the geometry once and reuse point data if geometry is not animated (see the Total Time Static column). A fully dynamic comparison of VIR and MVR recomputes the point data every frame, and is shown in the Total Time Dynamic column. VIR performance improvements are highlighted in blue, and increase significantly as geometric complexity increases. All GPU times are reported in milliseconds (ms).

<table>
<thead>
<tr>
<th>Model</th>
<th># of Triangles</th>
<th># of Points</th>
<th>G-Buffer</th>
<th>VIR</th>
<th>Depth Buffers</th>
<th>Lighting</th>
<th>Total Time Static</th>
<th>Total Time Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Faster</td>
<td>Faster</td>
</tr>
<tr>
<td>Lucy</td>
<td>53,715</td>
<td>83,989</td>
<td>0.10 (0.10)</td>
<td>0.46</td>
<td>1.64 (2.21)</td>
<td>1.38 (1.46)</td>
<td>3.12</td>
<td>1.21 (1.46)</td>
</tr>
<tr>
<td></td>
<td>106,973</td>
<td>125,136</td>
<td>0.13 (0.13)</td>
<td>0.62</td>
<td>2.07 (4.10)</td>
<td>1.38 (1.49)</td>
<td>3.58</td>
<td>1.59 (1.49)</td>
</tr>
<tr>
<td></td>
<td>256,621</td>
<td>229,840</td>
<td>0.21 (0.21)</td>
<td>1.03</td>
<td>3.54 (15.91)</td>
<td>1.35 (1.53)</td>
<td>5.10</td>
<td>3.46 (17.65)</td>
</tr>
<tr>
<td></td>
<td>502,667</td>
<td>369,328</td>
<td>0.35 (0.35)</td>
<td>1.95</td>
<td>5.56 (32.73)</td>
<td>1.38 (1.50)</td>
<td>7.29</td>
<td>4.74 (34.58)</td>
</tr>
<tr>
<td></td>
<td>1,023,609</td>
<td>563,878</td>
<td>0.60 (0.60)</td>
<td>3.31</td>
<td>8.32 (65.86)</td>
<td>1.40 (1.50)</td>
<td>10.32</td>
<td>6.58 (13.63)</td>
</tr>
<tr>
<td></td>
<td>2,005,071</td>
<td>599,916</td>
<td>1.02 (1.02)</td>
<td>5.48</td>
<td>7.55 (120.09)</td>
<td>1.37 (1.46)</td>
<td>9.94</td>
<td>12.33 (122.57)</td>
</tr>
<tr>
<td>Buddah</td>
<td>1,087,476</td>
<td>164,831</td>
<td>0.43 (0.43)</td>
<td>2.93</td>
<td>2.50 (52.80)</td>
<td>1.36 (1.49)</td>
<td>4.29</td>
<td>12.99 (55.72)</td>
</tr>
<tr>
<td></td>
<td>1,597,951</td>
<td>121,768</td>
<td>0.35 (0.35)</td>
<td>3.77</td>
<td>1.9 (37.00)</td>
<td>3.50 (3.68)</td>
<td>5.75</td>
<td>7.14 (41.03)</td>
</tr>
</tbody>
</table>

77
6.1.2 Quality

Shown in Figures 6.3 and 6.4 on the following pages, the visual quality of soft shadows produced by our algorithm is excellent, and quite similar to the reference MVR solution despite taking significantly less time to render. The characteristics of high quality shadowing are all present, including contact hardening, a soft penumbra gradient that transitions seamlessly between levels of occlusion, proper self shadowing, and contiguous shadows without light leaking artifacts. Due to the high density, view independent point cloud our algorithm generates, shadows are also temporally stable under animation even when resampling the geometry every frame.

The output of our algorithm exhibits a minor “bloating” of the shadows when compared to the multi-view rasterization reference. This is highlighted in the magnified inlays of Figure 6.3. The bloating artifact is a result of the differences in how texel coverage is approximated in polygonal rasterization and point-based rendering. Figure 6.2 illustrates these differences and compares a) standard polygon rasterization, b) point-based rendering, and c) conservative polygon rasterization. All of these approaches are coarse approximations of the actual amount a texel is covered by a polygon, and as such no one is definitively correct. Standard rasterization is biased towards undersampling, since only texels whose central point is contained within the polygon are considered covered. Conservative rasterization is biased towards oversampling, since all texels intersecting a polygon are marked as covered. Point-based rendering using nearest neighbor reconstruction presents a middle ground between the two polygon rasterization options. The density of point samples produced by VIR compared to the density required by the image resolution and view-projection transformation determines the bias of point rendering towards either under or oversampling. Consequently, the tailored point sets of VIR are often oversampled compared to standard rasterization and cause the visual bloat. If our algorithm’s shadows were compared to MVR using conservative rasterization, they would instead appear

![Figure 6.2: A comparison of texel coverage approximations. a) standard polygon rasterization, b) point-based rendering, and c) conservative polygon rasterization.](image-url)
Figure 6.3: A comparison of soft shadowing algorithms. Top: soft shadows rendered using multi-view rasterization of polygons. Bottom: VIR paired with our GPU Compute parallel view rendering approach. Both methods produce 128 high resolution depth buffers and similar high quality shadow penumbra, but VIR takes only a fraction of the time.
Figure 6.4: A quality comparison of soft shadows rendered by MVR and VIR. Top Row: the 2M polygon Lucy statue. Middle Row: the 1M polygon Buddha. Bottom Row: the 1.5M polygon San Miguel Trees. Each scene is rendered using MVR (left column) and VIR (middle column). Note the significant reduction in rendering time when using VIR. RMSE numerical quality and HDR-VDP2 perceptual quality measures are included (right column). Although HDR-VDP2 reports areas containing differences with a high probability of detection in the San Miguel Trees scene (bottom right), these differences are related to the sampling bias of the MVR reference, are not a representation of error, and are actually a beneficial quality of VIR in this test case. See the next page for a more information.
slightly smaller than the rasterization reference. As the resolution of the target image increases, the magnitude of the bloat decreases, since the influence of an individual texel declines and the various texel coverage approaches converge.

Figure 6.4 shows soft shadows rendered using Multi-View Rasterization (left column), our VIR and GPU Compute multi-view point rendering algorithm (middle column), as well as numerical and perceptual image comparison measures (right column). The colored heat map overlay is the output of HDR-VDP2, where red indicates that differences between the images are detected a high percentage of the time and blue indicates differences are rarely perceived. Each row of the figure shows a test case using different occluder geometry, including the two million polygon “Lucy” model (top row), the one million polygon Buddha statue (middle row), and the one and a half million polygon San Miguel Trees (bottom row). In each test case, the numerical difference between the MVR reference and our algorithm’s output is low.

The perceptual differences highlighted by HDR-VDP2 provide a better understanding of where the algorithms differ. The perceptual measure emphasizes areas where a) the bloat artifact alters self shadowing or contact hardening (see Lucy and Buddha); as well as where b) the differences in VIR’s conservative sampling and standard rasterization’s undersampling bias change how thin geometry is represented (see the San Miguel Trees). Although the output of HDR-VDP2 for the San Miguel Trees scene includes many areas where a high probability of detection exists for the differences between the images (colored red), this does not indicate a failure of VIR. On the contrary, standard rasterization often fails to represent thin projected geometry since few pixel centers are covered and sub-pixel coverage is underestimated. Conservative rasterization eliminates this problem, but includes all surfaces regardless of their actual sub-pixel importance. VIR’s approach to sampling coupled with point-based rendering is a potentially better solution for representing thin geometry that exists between these two extremes. This warrants more investigation, and a comparison with accurate offline algorithms that densely sample sub-pixel polygonal coverage.

6.1.3 Limitations

As with VWSS and MVR, the lighting phase of our algorithm – and any image-based multi-view soft shadowing algorithm – is a performance bottleneck caused by memory bandwidth limitations. To keep our focus on accelerating depth buffer rendering, we work around this problem by computing screen space lighting at a lower resolution and upscaling the lighting result to the final resolution using bilinear interpolation. Although this provides improved performance, the lighting result is noticeably softer at sharp discontinuities as a consequence of the interpolation. Since our algorithm is based on the shadow mapping algorithm, we inherit its limitations. These limitations are well known, thoroughly studied, and our approach can incorporate the
myriad of improvements to the basic algorithm from decades of research. It may be possible to eliminate the quality limitations of shadow mapping by ray tracing a spatial data structure constructed from VIR’s point samples; however, we have not yet pursued this direction. Testing with a wider variety of geometry, as well as target views with little coherence (e.g. omni-directional soft shadowing), is necessary to further understand implications for production settings.

6.2 Beyond Shadows

Our success accelerating multi-view shadow rendering motivates us to apply our rendering design to additional visual effects whose approximations improve with many samples. Several visual effects may benefit including ambient occlusion, defocus blur, environment mapped reflections, and diffuse global illumination.

Ambient occlusion is a rendering technique that approximates how much ambient light is able to reach surfaces. Computationally, ambient occlusion algorithms determine how much light is blocked from reaching a surface by other nearby surfaces [2]. This often results in areas of shadowing where multiple surfaces make contact. Ambient occlusion is difficult to compute accurately within real-time frame budgets due to the need to perform dense visibility sampling on the hemisphere around the point of interest. Screen space approximations produce reasonable results with practical execution times [59, 56]; however, these techniques are not physically based and still struggle to resolve visual artifacts due to their approximate nature. Much like soft shadowing, accurate ambient occlusion techniques use either ray tracing or many depth buffer images. Our algorithms exhibit excellent depth buffer rendering acceleration that ambient occlusion techniques would naturally benefit from. The dense sampling required for ambient occlusion may still prohibit real-time execution, so we are also interested in the possibility of our rendering design significantly accelerating accurate offline methods.

Soon after the introduction of NVIDIA’s Compute Unified Device Architecture (CUDA), a new algorithm rendered accurate defocus blur by synthesizing a light field in CUDA at runtime from a single rendered view [90]. The algorithm produces high quality defocus blur by using image warping, CUDA’s atomic functions, and post-process filtering to fill holes caused by warping. Using either VWSS’s or VIR’s multi-view sampling approaches, we expect to improve upon their algorithm’s accuracy by removing the post-process hole-filling filtering step.

Reflections of geometrically complex objects from non-planar reflectors are still a challenging problem for real-time applications [2]. For example, nearly all car racing video games still do not include high quality reflections of nearby cars reflected by the car the player is manipulating. Our single geometry traversal sampling method and parallel image rendering design removes the bottleneck caused by multiple traversals of complex geometry (e.g. car models) and we expect it to accelerate the rendering of the many cube maps (and cube map faces) utilized in dynamic
environment mapping. Similarly, diffuse global illumination approximated by many image-based samples should benefit from our parallelization of image rendering. By addressing the geometry bottleneck with View Independent Rasterization, performing basic diffuse shading only once during VIR sampling, and parallelizing image rendering using points, we expect to observe substantial acceleration when integrating our algorithm into existing image-based diffuse global illumination techniques.

**Architectural Obstacles**

Our decision to utilize points as the primary multi-view primitive unlocks significant computational gains; however, frustrating obstacles still remain when using points with existing GPU architectures. Specifically, hardware vendors’ current focus on optimizing their architectures for single-view object-order polygonal rasterization limits the potential of GPU Compute in many point-based rendering applications. For example, z-buffering tasks executed for color images are traditionally performed by dedicated hardware in the GPU’s output merger during the final stages of the graphics pipeline. The hardware resolves concurrency problems – where multiple fragments project to the same texel – and stores the proper depth and color values. This process is especially fast due to bandwidth saving compression methods paired with the dedicated raster operator (ROP) units that employ custom circuitry (called ZROP and CROP) tailored for depth comparison and color blending operations. Unfortunately, these ROP units are exclusively accessible at the end of the graphics pipeline. GPU Compute threads performing multi-view z-buffering are unable to leverage their speed. Instead, z-buffering in GPU Compute must be performed using much slower atomic functions that execute using the Level 2 (L2) cache of each Streaming Multiprocessor (SM). This creates an undesirable choice when rendering many views with points, either: (1) use the fast ROP hardware, but be forced to process views serially, or (2) render views in parallel with GPU Compute, but be limited by relatively slow atomic functions without dedicated hardware support. To solve this problem, we suggest either: (a) enabling access to ROP units from GPU Compute, or (b) adding dedicated hardware to accelerate atomic functions. Given the complexity and proprietary nature of GPU architectures, it is not straightforward to know or predict which of these solutions is best.

Compounding the limitations of GPU Compute’s atomic functions, widely used graphics APIs (e.g. Direct3D 11) only expose support for atomic operations using 32-bit integer storage formats. Although 32 bits of precision is sufficient to represent depth, it is problematic when our goal is to atomically compare depth and color values when rendering color images with points. We work around this problem by packing 16-bit half precision depth and 5-6-5 bit precision RGB color into a single 32-bit value, and then performing rendering using the existing 32-bit atomics. Although this works on existing hardware, the reduction in depth precision creates
noticeable artifacts. Fledgling support for 64-bit atomic functions in GPU Compute is available in OpenGL 4.5 with hardware vendor specific extensions, and we hope this functionality – along with hardware acceleration for atomics – becomes mainstream as soon as possible.

Finally, choosing when to perform texture sampling and filtering during multi-view point-based rendering with VIR is a challenging decision laden in the nuances of how existing GPU architectures are designed. As a consequence of its object-order optimizations, the GPU expects that the geometry moving through the graphics pipeline, and the fragments generated by the rasterizer, are composed of a single (or a small number of) surface material(s). This optimization strategy limits the number of dependent memory resources, such as textures, variables, and filtering parameters, that the GPU must keep readily available during shading. Unfortunately, this object-order optimization maps poorly to other algorithms that encounter surfaces of widely varying materials from pixel to pixel. This shading divergence problem has stifled the adoption of GPU-based ray tracing algorithms in real-time applications, since “über shader” (or “megakernel”) strategies rely on branching flow control to dynamically select the shading properties and memory resources of the appropriate surface material. Large flow control structures bloat the number of instructions of shader kernels, reduce the amount of parallelism due to high register counts, and introduce stalls while fast shaders wait for divergent shaders to complete when executing on the same multiprocessor. The scatter based design of our multi-view point rendering approach experiences similar challenges, since points representing an arbitrary number of surface materials are rendered to any number of target images simultaneously. We again face an undesirable choice, either: (1) dynamically select the surface material at render time, and pay the price associated with flow control; or (2) store multiple pre-sampled and/or pre-filtered colors in the point itself, and pay the memory bandwidth cost of larger rendering primitives. Additionally, this second option introduces the possibility that the type of filtering chosen before view rendering may be inappropriate for any given target view. Much improvement is required for GPUs to resolve this problem; however, the introduction of “bindless textures” and explicit residency management of texture resources in Direct3D 12 are encouraging developments in the right direction. At the moment, we advocate the use of flow control to solve this problem, since we expect this solution’s limitations to be viewed as a higher priority by hardware vendors as GPU ray tracing becomes more commonplace.
Chapter 7

Conclusion

In this dissertation, we present a general multi-view point-based rendering framework that accelerates the simulation of global lighting effects for complex geometry using the GPU. Due to the complexity of advanced lighting algorithms coupled with the nuanced nature of achieving optimal performance from modern GPU hardware, we present, implement, and evaluate multiple strategies to solve the multi-view point generation and point rendering problems. This chapter summarizes the strengths and limitations of our conceptual approach, as well our specific View Warped Soft Shadows and View Independent Rasterization implementations. We conclude with a discussion of future research directions, suggestions for evolving graphics hardware architectures for points, and further improvements to multi-view rendering.

7.1 Strengths

An important strength of our multi-view point rendering framework is the efficient transformation of complex polygonal geometry into multi-view tailored point sets in real time. Our approach leverages heavily optimized fixed function hardware for point generation, but is still exceptionally flexible due to the use of programmable shaders to compute sampling parameters on the fly. Additionally, by dynamically transforming polygons into points at render time, our approach does not require modifications to the authoring process of scene geometry or art assets. All existing content and artist work-flows are automatically supported. Similarly, our straight-forward sampling and rendering approach is designed for the GPU’s strengths while avoiding esoteric data structures and complicated reconstruction methods that are difficult to integrate and optimize in modern real-time game engines.

Our algorithms achieve significantly improved performance by working around the limitations of existing graphics architectures and the implicitly serial rendering design associated with them. By parallelizing image rendering using a more flexible rendering primitive, we exploit the
inherent coherence present in multi-view rendering and realize significant computational gains by keeping rendering data locally cached in GPU registers. For example, existing applications including soft shadows must choose between impractically slow but accurate multi-view methods or fast but approximate single-view solutions with objectionable artifacts. Our View Warped Soft Shadows algorithm facilitates a more practical quality/performance tradeoff by rendering depth buffers up to $13 \times$ faster and computing lighting over $3 \times$ faster than Multi-View Rasterization. When using 24 depth buffers and 120 samples per pixel (spp), VWSS exhibits similar performance and less error than the approximate Percentage Closer Soft Shadows algorithm. View Independent Rasterization achieves similarly impressive multi-view rendering performance improvements for complex geometry that is expected to become more common in the next five to ten years. VIR processes two million polygons and tailors a 600 thousand point data set for 128 views in 5.5 milliseconds. The computation of soft shadows for this kind of complex occluder is improved $4 \times$ to $13 \times$, while maintaining quality nearly identical to Multi-View Rasterization. Since both of our VWSS and VIR methods handle surface visibility and occlusion without major approximations, the severe failure cases of existing approximate multi-view algorithms are avoided entirely. The perceptual image comparison measure HDR-VDP2 rates both of our algorithms’ output as minimally different than the reference, and in many cases significantly closer to Multi-View Rasterization than other existing methods.

Looking to the future, our general approach to multi-view rendering is well suited to benefit from expected improvements in rendering architectures as hardware, graphics APIs, and game engines evolve. Our algorithm’s focus on memory coherence and parallelism position it well for a future where the majority of animated content is densely tessellated and the simulation of light uses primarily global lighting solutions.

7.2 Limitations

There are scenarios where a point-based approach to rendering is not ideal from a real-time performance perspective. When only a few polygons cover a large percentage of the pixels in a view’s output image, the easily cached geometry and brute force of fixed function polygon rasterization hardware will outperform point-based methods. This scenario is becoming less common everyday since the complexity of real-time geometry is rising faster than image resolution [29] and views are not often located directly next to surfaces.

Any multi-view rendering algorithm using many image-based samples makes heavy use of GPU memory and bandwidth, and VWSS and VIR are no exception. As a consequence, the lighting phase remains a challenging performance bottleneck for both of our algorithms — and indeed also for MVR and ISM. In the near term, several techniques might be applied to further reduce memory bandwidth requirements and improve performance. Rendering screen space
lighting at a lower resolution, utilizing dynamically varying lower resolution or precision depth buffers, employing reconstruction filtering techniques, or adopting temporal and/or stochastic methods may improve lighting performance at varying costs to image quality and latency. With the recent release of the Direct3D 12 and Vulkan graphics APIs, asynchronous compute might be used to pipeline and overlap the rendering of intermediate images (e.g. depth buffers or cube maps) and the final lighting workload. Longer term, better compression techniques may emerge for the dynamically created but quite similar depth buffers soft shadowing algorithms generate. We do not integrate any of these optimization solutions in our algorithms since they are not central to our contribution, are applicable to nearly all image-based rendering algorithms, and no single solution is best in all cases. Finally, our implementations use simple data structures to store the point sets generated by both VWSS and VIR. Although this approach sped up development time, simplified the evaluation, and reduced the time to acquire results, it does limit the complexity of scene content that can be managed efficiently. More capable spatial data structures, such as an Octree or Bounding Volume Hierarchy, are important next steps in evaluating multi-view point rendering on large – or even dynamically streaming – content.

The main quality limitations of VWSS and VIR stem from artifacts associated with an image-based sampling approach to approximating visual effects. Specifically, the quality of our soft shadowing implementations is ultimately limited by the shadow mapping algorithm that they extend. These limitations are thoroughly understood, and several decades of refinements to the basic shadow mapping algorithm can be incorporated into both of our implementations. For clarity and simplicity of evaluation, we have not employed all of them. The visual quality of VWSS is not as refined as MVR, which resamples occluder geometry for each depth buffer, but several techniques might improve it. First, as hardware and software changes improve performance, VWSS could generate a more detailed point cloud for use with higher resolution depth buffers. Similarly, including points for backfacing geometry can further improve quality in areas of disocclusion and for thin occluders. Additionally, we are experimenting with refining the point representation judiciously as distance from the light increases. Unlike VWSS, soft shadows rendered with VIR exhibit a nearly identical level of refinement and temporal stability as MVR since VIR’s high density view-independent point set is more complete than VWSS’s view-dependent point set. However, this quality improvement comes at the cost of requiring more sophisticated solutions to manage the larger point set and maintain high performance.

7.3 Future Work

The fixed function point sample generation, memory coherence, and parallelism we exploit in both VWSS and VIR might find application in accelerating other visual effects. For VWSS, high quality image warping is enabled by the viewing constraints typical of real time area lights.
Other applications with similar constraints may also leverage VWSS’s multi-view optimized approach. Soft shadows cast from omni-directional and volume light sources, defocus blur / depth of field effects, and ambient occlusion may all benefit, since they rely heavily on depth buffers and their approximations improve with many views.

For VIR, less constrained global lighting effects, such as diffuse global illumination approximated by many views, are clear candidates for acceleration. As a consequence of VIR’s high density sampling and simple reconstruction approach, there are many avenues for performance optimization beyond what we have implemented and evaluated. Several of ideas in this direction are presented at the end of Chapter 5. Implementing and evaluating these optimization strategies may provide further insight into how GPUs may be exploited for fast multi-view rendering, in addition to how GPUs may evolve to better accelerate these challenging workloads.

In all cases, the ultimate test of a high performance rendering algorithm is integration in an existing real-time game engine, such as Unreal Engine, Frostbite, or Unity. While academic efforts to evaluate rendering algorithms running on actual game content have been largely unsuccessful thus far (as a result of fierce competition in the game industry fueling a culture of proprietary technology and closed source), the recent shifting landscape of the industry has started to do away with these limitations. For example, the full source code of Unreal Engine 4 is now available to the public. Although engine source code is a transformative step towards the ideal evaluation of rendering algorithms, representative geometry and other art assets are still difficult to acquire due to the multi-million dollar investment required to create them.

### 7.3.1 Suggestions for Future Graphics Architectures

High quality computer graphics requires indirect illumination, which is often approximated using multi-view rendering. Unfortunately, today’s GPU architectures do not support multi-view rendering well, even when views are quite similar to one another. For example, existing real-time video game titles precompute or use screen space approximations of ambient occlusion; and reflections of dynamic objects are often not present at all due to the inefficiency of traversing complex geometry several times. We chose points produced frame-by-frame as a way of avoiding the limitations of the graphics pipeline for multi-view rendering and found success. However, current hardware could still use much improvement for point rendering. In particular, dedicated hardware support for high performance 32-bit and 64-bit floating point atomic functions and increased memory bandwidth are critical to achieve even faster multi-view z-buffering in GPU Compute. Although increased memory bandwidth and decreased memory latency is a long-standing, well known bottleneck to GPU performance, atomic functions are a less common performance bottleneck for traditional rendering applications, and as a result are not treated as a first class hardware priority on many platforms. A relatively straight-forward solution
to this problem might expose the existing raster operator hardware (ZROP and CROP) that handles z-buffering and color blending in the graphics pipeline to GPU Compute. Additionally, 64-bit memory resources are essential to render z-buffered, color images with points and atomic functions. Support for these double precision resources in both hardware and software APIs is fledgling, but still not yet common.

Of course, as useful as they are, points are not a direct attack on the multi-view problem. Utilizing a hardware accelerated graphics pipeline to render multiple views in a single pass is essential. In the short term, this may be accomplished by removing limitations of the Geometry shader or even the rasterizer itself. The recent renewed interest in virtual reality, coupled with the release of new low-level rendering APIs, has inspired advances in this direction [80], but further improvement is required. In the longer term, the graphics pipeline should be improved to efficiently support multi-view rendering directly from polygons, without a transformation into points. Another potential alternative may be to create a separate, more efficient pathway designed for indirect effects by providing a flexible compute system with direct access to hardware accelerated processing of geometry and/or points.
REFERENCES


[40] Mark W. Jones. 3D distance from a point to a triangle. Technical report, Department of Computer Science, University of Wales, 1995.


