EVOLVING THE REAL-TIME GRAPHICS PIPELINE FOR MICROPOLYGON RENDERING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Kayvon Fatahalian
December 2010

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Patrick Hanrahan, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

William Dally

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Kurt Akeley

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

The modern real-time graphics pipeline is a versatile parallel architecture that accommodates a wide range of rendering techniques. The architecture is implemented by heavily optimized graphics processors (GPUs) that employ a mixture of application-programmable and fixed-function processing resources, yet its design lends itself to a simple programming model easily understood by non-expert programmers.

A major goal of future graphics systems is rendering geometrically complex, film-quality scenes in real time. Unfortunately, current GPU implementations not only require additional compute capability to handle high-resolution surfaces represented by subpixel-area *micropolygons*, the fundamental graphics pipeline operations of surface tessellation, rasterization, and shading execute inefficiently under this advanced workload.

This dissertation evolves the graphics pipeline architecture and its associated rendering algorithms to increase system efficiency when processing micropolygons. The proposed redesign extends the pipeline with a new parallel algorithm for high-quality, adaptive surface tessellation, making it possible to generate crack-free meshes that represent surfaces accurately, but without excessive numbers of micropolygons. It increases rasterization throughput using micropolygon-parallel processing and analyzes the cost of rasterizer support for motion blur and camera defocus. It also adds pipeline logic to detect and avoid redundant shading computations, reducing shading costs more than eight times. The resulting real-time micropolygon rendering pipeline architecture increases rendering efficiency and, due to its evolutionary nature, maintains the graphics pipeline's simple programming model and the throughput-optimized design of a GPU's programmable processing cores.

Acknowledgments

I heard a lot of things about Pat Hanrahan before coming to Stanford. They are essentially all true. If you get the chance to work with Pat, I suggest you take it. You will walk away thinking differently about your world and the technology within it. Needless to say, I am very thankful Pat took me on as a student. He has gone far beyond the call of duty in support of me and my career during my time at Stanford. For this I am extremely grateful.

I also wish to thank Kurt Akeley, Bill Dally, Alex Aiken, and Juan Alonso for serving on my orals committee. Throughout several projects at Stanford, I have always appreciated Bill's strong support. Without question, Kurt's guidance has had great influence. Conversations with Kurt are highly prized, and a suggestion from Kurt is always taken to heart. I am grateful to Alex for his technical leadership during the Sequoia project and his personal and career advice since.

The micropolygon project succeeded because a fun group of sharp people showed up at Stanford on Tuesdays and were not afraid to argue. In addition to Pat and Kurt, Bill Mark and Henry Moreton made enormous contributions. Much of the work described in this dissertation is directly due to the ideas and efforts of fellow students Solomon Boulos, Matthew Fisher, Edward Luong, James Hegarty, and John Brunhaver. (Solomon deserves special thanks for the significant amount of effort he put in to this project.) Jeremy Sugerman and Jonathan Ragan-Kelley also made their feelings known in many discussions. Intel, NVIDIA, AMD, the Stanford Pervasive Parallelism Laboratory, and the National Science Foundation generously provided financial support.

Over the years, opportunities provided by Karen Leppin, Mark Richer, Nick Triantos, Kiran Bhat, Jessica Hodgins, and Doug James turned into inflection points. At Stanford Ian Buck, Mike Houston, Martin Casado, Andreas Sundquist, and Ren Ng (another special thank you) became role models and also good friends. Annie Bosler continues to motivate and inspire. She also claims to be a big fan.

Finally, my most important thank you goes out to Mom, Dad, and Keon for their love and support. Since I have moved to Palo Alto, we have taken many photos of the four of us at airports. I will be keeping every one of them. Hey guys, I did it.

To Mom, Dad, and Keon

Contents

List of Tables

List of Figures

Chapter 1

Introduction

When I take a look around my surroundings I see many complex surfaces. Highly curved objects lie about my apartment and on my office desk. My clothes are full of complicated folds. Out my window, I see natural surfaces that are rough or bumpy as well as foliage with intricate shape (a towering redwood tree dominates my view to the west). Complex surfaces like these are pervasive in our world. They make environments visually compelling and enrich them with style and character.

The importance of accurately modeling complex surface detail has always been fundamental to the design of offline rendering systems [Cook et al. 1987]. These systems use high-resolution meshes to accurately capture shapes meticulously created by film artists. It is common for film scenes to consist of hundreds of millions of tiny polygons, called *micropolygons*, that are about a pixel or less in area. Because of the small size of micropolygons, rendered images contain no evidence that surfaces are approximated discretely by polygon meshes. While it is acceptable for an offline renderer to take minutes or hours to process a frame, real-time graphics systems must synthesize images in only a few milliseconds. As a result, the geometric complexity in film scenes dwarfs that present in interactive environments. Objects in games have traditionally been represented using low-resolution meshes. Game artists must choose to omit complex objects from environments, or approximate them coarsely using polygons that cover many pixels on screen.

Although today's games do not yet equal the visual quality of early computer-generated animated films, the magnitude of compute resources available to interactive graphics systems is staggering. A graphics processor (GPU) in a high-end desktop PC today has over three orders of magnitude more compute capability (FLOPS/sec) than Pixar's entire render farm for Toy Story 1 [Snider 1995; Henne et al. 1996]. As compute capability continues to increase, more rendering techniques become viable for inclusion in games. A major challenge facing GPU architects is how to most efficiently organize these resources to produce the best images possible in real time.

While advances in animation, lighting, and anti-aliasing are all clearly needed to close the gap between real-time and film rendering [Andersson 2010], I believe one of the most important ways to leverage increasing compute power is to dramatically increase the geometric detail present in interactive scenes. Future graphics systems should provide games the ability to render richly detailed scenes, with artifact-free, micropolygon-resolution surfaces in real time. In pursuit of this goal, this dissertation focuses on reoptimizing the real-time graphics pipeline for micropolygon workloads.

1.1 Why Micropolygons

The stone house pictured at the top of Figure 1.1 features many complex surfaces. Bumpy cornerstones extrude from the house's walls and curved red tiles cover its roof. The detail in these surfaces is easy to observe along the house's silhouettes. The roof tiles and cornerstones also cast accurate shadows that effect illumination of themselves (self-shadowing) and of the house's walls. These details are rendered accurately because house geometry is represented using a high-resolution triangle mesh.

The lower image in Figure 1.1 is a rendering of the same house using the same texturing and soft shadowing techniques, but with scene geometry represented using a low resolution triangle mesh. Now, the house's roof and walls are modeled as flat planes and surface details such as the cornerstones, roof tiles, and even the windows on the left side of the image are represented using texture maps, not mesh triangles.

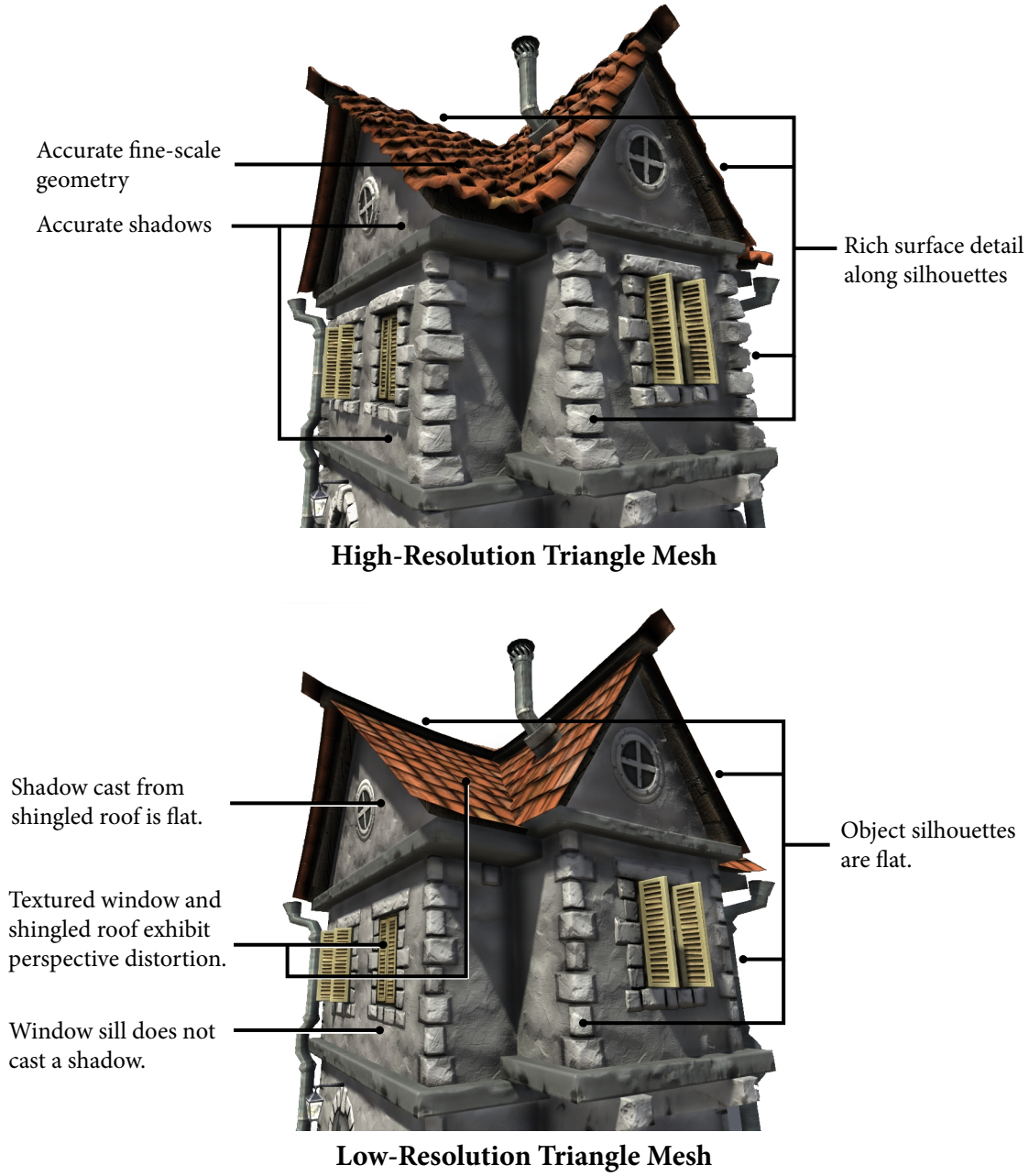


Figure 1.1: Renderings of a stone house using a high-resolution (top) and low-resolution (bottom) triangle mesh. A lack of geometric detail in the bottom image results in visible artifacts. House image by Unigine Engine, © Unigine Corporation (used with permission).

In some areas of the image, texture mapping is a convincing substitute for fine-scale geometric detail. Unfortunately, the illusion does not work along the house's silhouettes, where it is clear that geometry is flat. Similarly, the roof tiles and windows on the left side of the image appear distorted because these features are viewed from a glancing angle. The actual geometry of the house is non-planar in these areas and approximating it with a textured plane does not yield correct perspective or occlusions. These artifacts become even more noticeable when objects are moving on screen. A third artifact caused by the low resolution mesh is incorrect shadowing. Since the cornerstones don't actually protrude from the house, they do not cast shadows on its walls. Also, the shadow cast by the roof onto the wall is flat. It lacks the curved appearance of the shadow in the top rendering of the home.

Rendering methods such as precomputing static illumination, bump mapping, normal mapping and occlusion mapping leverage texture mapping to compensate for a lack of geometric detail in interactive applications [Akenine-Möller et al. 2008]. These techniques have low rendering cost and are used effectively in games. However, they are prone to artifacts like the ones discussed above. Ironically, as shown by the incorrect shadowing in Figure 1.1, artifacts due to texture-based approximations can become more pronounced as the sophistication of shading or lighting simulation increases.

Put simply, complex surfaces must be represented accurately for high-quality rendering. Micropolygons have proven to be a robust and effective (although brute force) surface representation for offline rendering. Ubiquitous use of micropolygons in real-time graphics systems would significantly advance the visual quality of interactive applications.

1.2 Micropolygon Challenges

Graphics applications abstract the process of rendering a picture from a 3D representation of a scene as a pipeline of operations. This pipeline computes how input primitives, such as triangles, influence the color of final image pixels. For interactive rendering, this sequence of operations is defined by a real-time graphics pipeline

architecture like OpenGL [Segal and Akeley 2010] or Direct3D [Blythe 2006]. Since these two architectures are similar in structure and designed for implementation by GPUs, for convenience, I will often refer to real-time graphics pipeline as the GPU pipeline in this dissertation (However, the reader is encouraged to recognize that a GPU is an *implementation* of the GPU pipeline *architecture*).

Because of the high cost of 3D graphics operations, it is critical for GPU implementations of the real-time graphics pipeline to be very efficient systems. Interactive applications benefit from as much performance as a GPU can deliver, and to meet this demand GPUs contain a large collection of programmable and fixed-function processing resources. Under the rendering workloads of current games, a GPU sustains high utilization of its compute resources to maximize rendering performance.

However, micropolygon workloads present new challenges for the GPU pipeline. Clearly, representing surfaces using high-resolution micropolygon meshes requires the GPU pipeline to process many more polygons each frame, increasing rendering cost. But rendering micropolygons using the GPU pipeline is not only expensive, it is inefficient. That is, maintaining the current GPU pipeline architecture and simply scaling up current GPU designs to feature more processing and more bandwidth would yield an inefficient system for real-time micropolygon rendering.

Three important sources of inefficiency are:

1. Parallel, adaptive tessellation is missing from the GPU pipeline. For many reasons it is not practical for an interactive application to precompute and store a high-resolution mesh representation of an entire scene. For example, doing so incurs substantial storage and bandwidth costs, increases the cost of non-rendering operations like simulation and animation that do not require high-resolution meshes, and makes it challenging to provide the appropriate level of mesh detail for all possible views. Instead, micropolygon meshes are generated on-demand in the GPU pipeline from a compact surface representation. This process, called *tessellation*, generates high-quality micropolygon meshes by adapting mesh resolution to surface properties and camera view, such that the surface is represented accurately, but without an exceedingly large number of polygons. Existing tessellation algorithms are not

amenable to high-performance parallel execution and are not expressible using current GPU pipeline abstractions. As a result, there is currently no efficient way to generate micropolygons in a real-time system.

2. Pixel-parallel rasterization is inefficient. *Rasterization* is the process of determining what image pixels a polygon overlaps. Current GPUs achieve high rasterization throughput by executing rasterization computations for many pixels covered by a polygon in parallel. Unfortunately, this parallelization strategy provides little benefit when polygons are less than a pixel in area. Micropolygon rendering increases polygon count and leads to less efficient rasterizer execution. GPU rasterizer implementations must be reoptimized to efficiently process micropolygons.

3. GPU pipeline implementations perform many redundant shading computations. *Shading*, or computing the appearance (color) of a surface, involves all the light and material simulation in a scene. As a result, shading is often the most expensive computation in a GPU pipeline. Unfortunately, when processing surfaces tessellated into micropolygons, implementations of the GPU pipeline must perform many redundant shading computations. Given the high cost of shading, these extra computations severely reduce overall rendering performance.

1.3 Evolving the GPU Pipeline

Despite the challenges described in the previous section, I have chosen to pursue the goal of real-time micropolygon rendering by evolving the existing GPU pipeline and its associated rendering algorithms. The result of this evolution is the modified GPU pipeline shown in Figure 1.2. This pipeline, which I will refer to as the real-time micropolygon pipeline, exhibits two significant extensions to the current GPU pipeline architecture (highlighted in red). These extensions enable new functionality that overcomes the GPU pipeline's inefficiencies in tessellation and shading. Figure 1.2 also highlights several pipeline stages in yellow. The architecture of these stages undergoes little to no change in the real-time micropolygon pipeline, but their implementation is modified to meet the needs of micropolygon rendering.

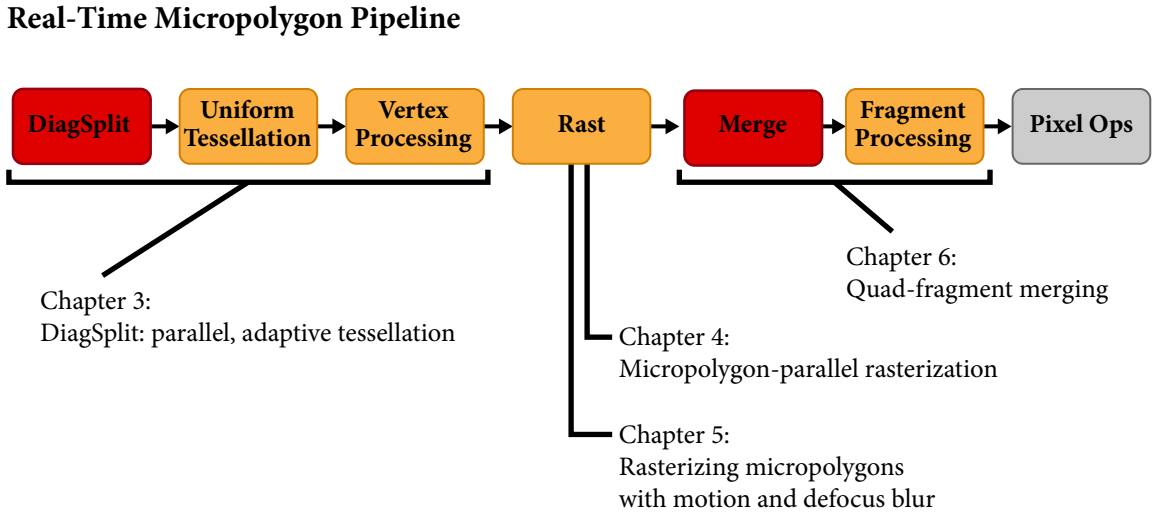


Figure 1.2: A micropolygon rendering pipeline for real-time rendering. Extensions to the current GPU pipeline are highlighted in red. Stages colored yellow exist in the current GPU pipeline, but their implementation is modified to meet the needs of micropolygon rendering. See Figure 7.1 at the end of this dissertation for a complete list of changes made to the GPU pipeline and its associated rendering algorithms.

This dissertation describes numerous solutions that enable the pipeline architecture and pipeline implementation changes summarized in Figure 1.2. It makes the following specific contributions:

1. *Identification of GPU pipeline inefficiencies.* It identifies three significant problems that prevent efficient micropolygon rendering on GPUs: a lack of high-quality, adaptive tessellation, insufficient parallelism in rasterizer implementations, and generation of redundant fragment-shading work.

2. *DiagSplit tessellation.* It provides a new algorithm for parallel, adaptive tessellation called DiagSplit. DiagSplit efficiently generates high-quality micropolygon meshes from parametric surfaces and integrates into the GPU pipeline as an extension of existing tessellation mechanisms. DiagSplit was designed in collaboration with Matthew Fisher, who proposed most of the algorithm's key ideas.

3. *Micropolygon-parallel rasterization.* It reoptimizes rasterization for micropolygon workloads using simpler algorithms and by processing many micropolygons in parallel.

4. *Interleaved sampling technique for rasterization with motion and defocus blur.* It analyzes the cost of adding support for motion and defocus blur effects to a micropolygon rasterizer and provides a new coverage-sampling technique based on interleaved sampling. This technique achieves greater rasterization efficiency than previous methods under high motion or moderate defocus.

5. *Quad-fragment merging.* It introduces quad-fragment merging to identify and remove redundant shading work from the GPU pipeline. When rendering surfaces tessellated into micropolygons, quad-fragment merging reduces the number of shading computations by more than a factor of eight while maintaining high image quality.

The decision to evolve, not replace, the existing GPU pipeline was motivated by performance; modifying the highly-optimized GPU pipeline architecture (and its rendering algorithms), rather than improving the performance of offline rendering systems, seemed a more likely path to real-time micropolygon rendering. However, this strategy has the added benefit of maintaining development continuity for real-time graphics applications. In fact, many optimizations presented here also improve GPU performance when rendering small, but not necessarily subpixel-area, polygons. Whenever possible, I have looked to isolate or compartmentalize change. Many components of the current GPU pipeline and current GPU implementations remain unmodified. As a result, this dissertation describes a path of technology evolution that I hope will allow existing real-time graphics applications to transition gradually from today's level of geometric complexity towards micropolygon rendering as GPU compute power increases.

In future chapters I will suggest that several of the contributions listed above are best realized through changes to GPU hardware and likely will not be implemented by software in a performant system. However, the complex task of building and conducting an end-to-end evaluation of a new graphics processor is beyond this scope

of this dissertation. While the ultimate goal of this work is real-time micropolygon rendering, this dissertation does not attempt to build a GPU that achieves required performance levels. Rather, the new techniques and algorithms proposed here are evaluated in detail using software implementations. Suggested GPU implementation changes are influenced by numerous conversations with GPU architects and they are intended to be consistent with the spirit of highly optimized GPU designs. Of course, it is possible that additional performance problems could be discovered if or when a full implementation of the proposed micropolygon rendering pipeline is built.

1.4 Dissertation Road Map

The organization of this dissertation follows the order of processing in the GPU pipeline. Chapter topics begin at the beginning of the pipeline and work downstream.

Chapter 2 provides an overview of the current real-time graphics pipeline architecture and its current GPU implementations. It enumerates strengths of the pipeline architecture that this dissertation seeks to preserve. It also describes the Reyes pipeline architecture for offline micropolygon rendering. Many of the contributions of this dissertation are influenced by ideas from Reyes.

Chapter 3 describes how the real-time micropolygon pipeline generates micropolygons using the DiagSplit algorithm for parallel, adaptive tessellation.

Chapters 4 and **5** focus on parallel algorithms for rasterizing micropolygons. Chapter 4 provides a more efficient implementation of existing GPU rasterizer functionality. Chapter 5 adds functionality for simulating motion blur and camera defocus effects.

Chapter 6 focuses on shading. It describes quad-fragment merging and then briefly contrasts the relative merits of this approach with those of shading mechanisms used in the Reyes pipeline.

Chapter 7 discusses the resulting micropolygon pipeline architecture as a whole. It highlights key interactions between new pipeline components, lists important design principles, and suggests immediate areas of future work.

Chapter 2

Graphics Pipelines

In both real-time and offline rendering, graphics systems must strike a balance between the conflicting goals of enabling high performance, achieving high image quality, and providing a simple, but versatile, interface for describing rendering computations. This balance is achieved by abstracting the rendering process as a pipeline of operations on intuitive structures such as vertices, polygons, and pixels. This chapter provides an overview of two widely used graphics pipeline architectures. The first is the real-time graphics pipeline (GPU pipeline). This architecture is used by interactive applications, is accelerated by GPU implementations, and serves as the starting point for the micropolygon rendering pipeline proposed in this dissertation. The second is the Reyes pipeline, which produces high-quality images and is used heavily in offline rendering for film. Reyes is designed to render surfaces represented by micropolygon meshes. Not surprisingly, many ideas from Reyes were considered when evolving the GPU pipeline for micropolygon rendering.

The ensuing discussion provides a high-level view of the structure and operation of these two pipelines. Detailed descriptions of the architecture and implementation of key pipeline stages are deferred until necessary in the subsequent chapters.

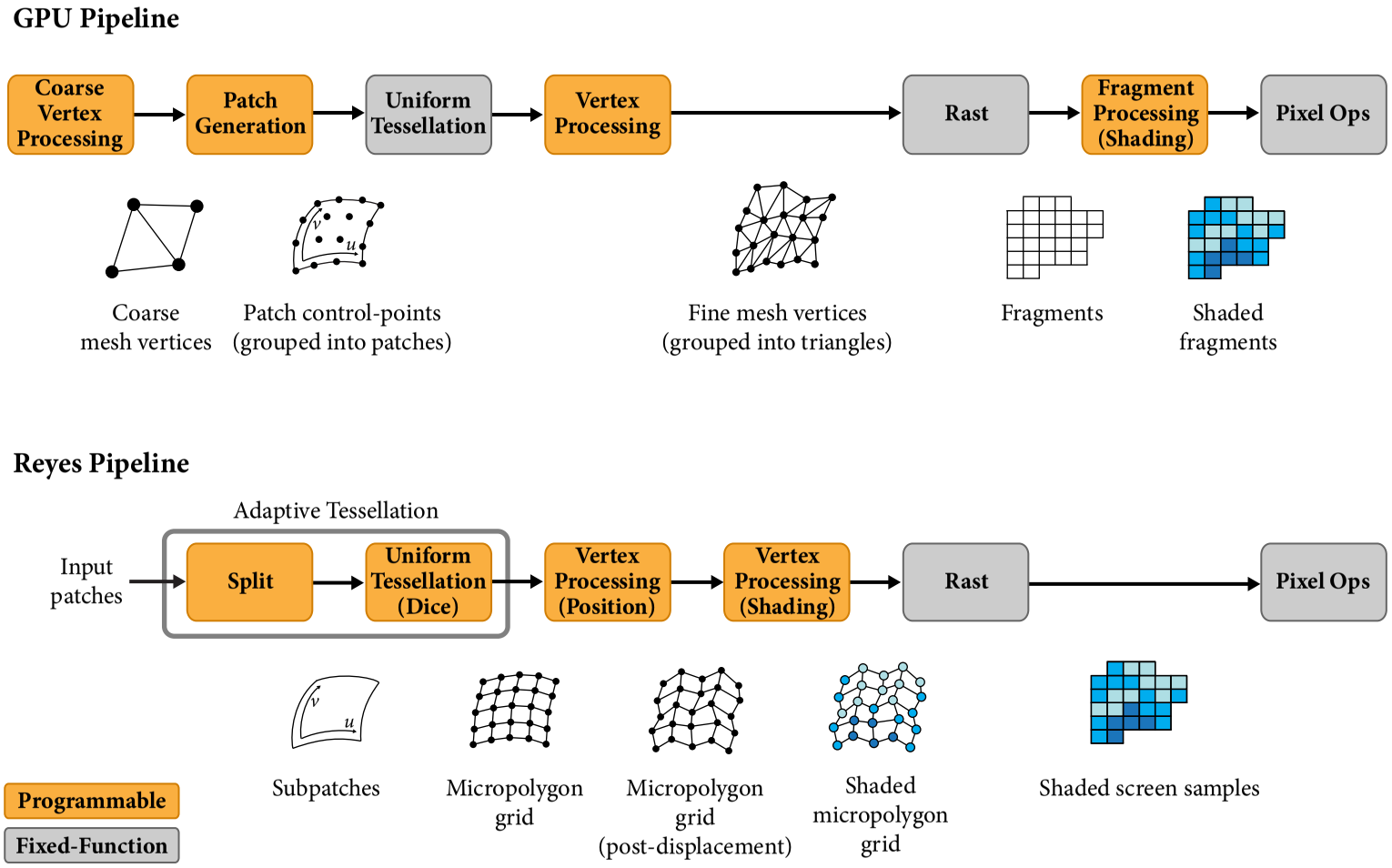


Figure 2.1: The GPU pipeline for real-time rendering and the Reyes pipeline for offline rendering.

2.1 The GPU Pipeline

2.1.1 GPU Pipeline Architecture

Pipeline Structures

The upper half of Figure 2.1 shows a seven-stage GPU pipeline that is similar to the pipeline architectures defined by OpenGL 4 [Segal and Akeley 2010] and Direct3D 11 [Mic 2010b]. This pipeline has support for surface tessellation, which is a new feature in the latest generation of these architectures. The stage names established in Figure 2.1 will be used consistently throughout this document. They are chosen for clarity and sometimes differ from the official names adopted by OpenGL and Direct3D. For simplicity, the illustration omits two pipeline stages (the Geometry Shader and Stream Output) that are not essential to the discussion of pipeline behavior in this dissertation. Each GPU pipeline stage generates or operates on one of five types of entities: parametric patches, vertices, triangles, fragments, or pixels. Entities are communicated between pipeline stages in data streams (black arrows). The process of generating a picture by manipulating the five types of GPU pipeline entities is summarized below.

Graphics applications and games have traditionally provided the GPU pipeline with a stream of triangles as input. However, when configured with tessellation functionality enabled, the GPU pipeline renders surfaces represented by parametric patches (e.g., Bezier patches). The pipeline tessellates each surface patch into a triangle mesh. The Uniform Tessellation stage emits the mesh as a stream of vertices. Then, Vertex Processing evaluates the parametric surface's position at each vertex and projects the result onto the screen. As illustrated in Figure 2.2, surface evaluation during Vertex Processing may utilize displacement mapping to produce fine-scale surface detail. After Vertex Processing vertices are regrouped into a stream of triangles. Rasterization (Rast) computes the screen coverage of each triangle and generates a stream of fragments. Each fragment represents a region of a triangle that overlaps one screen pixel (in the context of this dissertation, a pixel is a square region of the screen). Next, Fragment Processing performs a shading computation for each

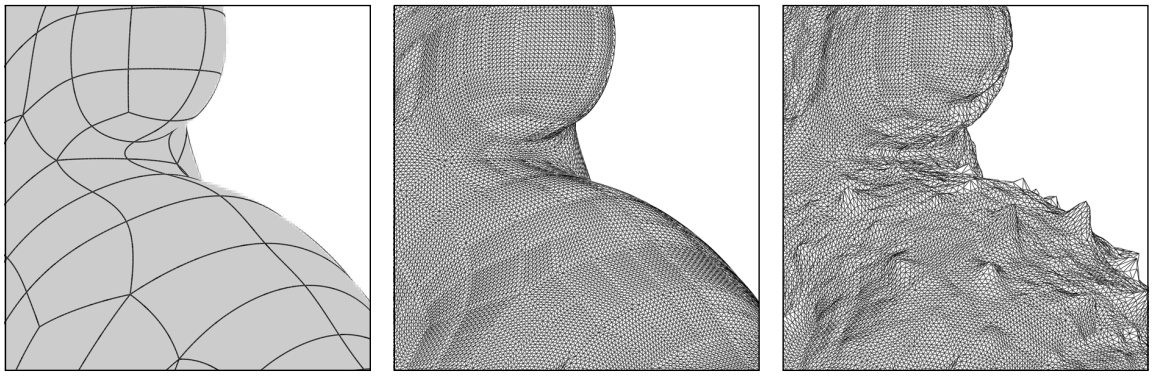


Figure 2.2: Tessellation of a complex surface represented using displaced Bezier patches. Left: Patch primitive boundaries. Center: Triangle mesh with vertex positions computed by direct evaluation of the Bezier surface. Right: Final detailed surface; vertex positions have been displaced in the direction of the Bezier surface normal (displacement magnitudes are obtained from a texture map). Surface evaluation, including both Bezier evaluation and displacement, is performed in the Vertex Processing stage of the GPU pipeline.

fragment (shading computes the color of the triangle at the overlapped pixel). Last, fragments visible from the virtual camera (they are not occluded by other fragments that are closer to the camera at the same pixel) are blended into the frame buffer, changing the value of image pixels (Pixel Ops).

In a GPU pipeline with tessellation, the Coarse Vertex Processing and Patch Generation stages (not mentioned in the preceding description) are important because they allow surface manipulations to be performed on the GPU, but at a much lower resolution than the final rendered mesh. In practice, applications do not directly provide the GPU pipeline with parametric patch primitives. Instead, they provide a low-resolution mesh representation of a surface. Given this coarse mesh, the Coarse Vertex Processing stage performs per-vertex manipulations, such as animation (via skinning or blend shapes) or physical simulation, that can be carried out cheaply at low frequency. The Patch Generation stage then constructs parametric patch inputs from the resulting vertices. For example, given skinned vertices from a coarse mesh face and its one-ring, the Patch Generation stage can be used to compute control

points for a Bezier patch. The Coarse Vertex Processing and Patch Generation stages are useful regardless of whether patch primitives are ultimately tessellated into small polygon or micropolygon-resolution meshes. However, since this dissertation makes no modifications to these pipeline stages they are rarely discussed in the subsequent chapters and are omitted from illustrations of the real-time micropolygon pipeline (Figures 1.2 and 7.1).

Shader Programming

Figure 2.1 colors pipeline stages to indicate whether they are fixed-function stages whose behavior defined by the pipeline architecture (gray boxes) or programmable stages whose behavior is defined by application code (yellow boxes). To program the GPU pipeline an application must set the configuration of fixed-function stages and provide programs (called "shader programs") that execute within environments associated with each programmable stage.

Shader programs are expressed using high-level graphics programming languages like Cg [Mark et al. 2003], HLSL [Mic 2010b], and GLSL [Kessenich 2009]. They execute on individual entities from a stage's input stream and emit entities to the stage's output stream. For example, a shader program that defines the behavior of Fragment Processing accepts one rasterized fragment as input and produces one shaded (colored) fragment as output. Shader programs are C-like functions that may contain data-dependent control-flow, manipulate complex data structures, and access large data buffers (e.g., textures) in addition to stream entities.

Although the shader programming model is flexible, shader programs are subject to pipeline constraints to ensure performant execution. For example, the GPU pipeline abstracts each shader program invocation as an independent sequence of logic that executes serially and in complete isolation from the processing of other stream entities. This abstraction permits data-parallel program execution on multiple stream entities without violating program correctness. Also, shader programming abstractions explicitly differentiate data access to streams, buffers, and textures, allowing GPUs to implement custom data paths for each form of communication.

GPU Pipeline Benefits

Over the past two decades, the GPU pipeline has transitioned steadily from its earliest incarnation as an entirely fixed-function architecture (OpenGL 1.0, introduced in 1992) to the programmable pipeline described above. Its ubiquitous use in interactive graphics stems from how successive pipeline evolutions have consistently met the needs of both application developers and system implementers. It is modern form, the GPU pipeline provides the following key benefits.

- **Programming simplicity**. The high-level, graphics-specific pipeline abstraction makes programming simple. A programmer that understands how a picture is made using the five graphics entities described above has basic knowledge of how to use the GPU pipeline. Shader programming abstractions keep programming simple by providing sequential execution semantics and a per-element execution model.

- **Versatility**. Programmable stages allow the GPU pipeline to accommodate a wide variety of rendering techniques. Graphics applications are free to define the surface representations, material models, and lighting techniques used for rendering.

- **High performance**. The pipeline abstraction and accompanying shader programming model provide significant optimization opportunities for GPU implementers. Most notably, they expose large amounts of parallelism (both task parallelism across stages and data-parallelism within stages), provide semantics for different forms of data access, and isolate fixed-function operations. As discussed in the next section, GPU designers exploit these properties to achieve efficient, high-performance GPU pipeline implementations.

For interested readers, Segal and Akeley summarize the original motivations and design goals for the OpenGL 1.0 pipeline architecture in [Segal and Akeley 1994]. Additional background on the history and design of the GPU pipeline is available in more recent articles by Blythe [2006; 2008] and Fatahalian and Houston [2008].

2.1.2 GPU Pipeline Implementation

GPU implementations of the real-time graphics pipeline employ sophisticated optimizations to achieve high rendering performance given limits on processor transistor count and power consumption.

Some key GPU optimizations are algorithmic. For example, it is common for GPUs to avoid expensive Fragment Processing work by detecting and discarding occluded fragments from the pipeline immediately following rasterization [Morein 2000]. This is a valid optimization because eliminating these hidden fragments prior to Fragment Processing does not impact the final rendered image (recall the GPU pipeline architecture only specifies a logical sequence of operations to make a picture; implementations may reorder processing as needed, so long as their results are consistent with the architectural specification). Additional examples of algorithmic optimizations implemented by GPUs include compression of pipeline data structures (e.g., the frame buffer) to reduce rendering bandwidth requirements, and caching of Vertex Processing results to avoid duplicating computation when a vertex is shared by multiple triangles in a mesh.

In addition to algorithmic optimizations, many important GPU optimizations involve the design and capabilities of its processing resources. Modern GPUs feature a heterogeneous collection of programmable and fixed-function processing resources heavily optimized to execute specific aspects of the GPU pipeline workload.

Throughput-Optimized Programmable Cores

A majority of a GPU's resources are organized into programmable processing cores whose primary responsibility is to execute pipeline shader programs. Unlike CPU processing cores, which provide excellent performance to applications employing only a few threads of control, GPU cores are designed to maximize overall pipeline throughput when executing many shader program invocations simultaneously. GPU processing cores feature large numbers of floating-point ALUs and achieve high ALU utilization by pushing throughput-computing techniques such as SIMD execution and hardware multi-threading to extreme scales [Fatahalian and Houston 2008; Lindholm

et al. 2008; Fatahalian 2010].

For example, a processing core in an NVIDIA GeForce GTX 480 GPU contains 32 ALUs, each capable of performing a single-precision floating-point multiply-accumulate operation per clock [NVI 2009]. The core efficiently uses these ALUs by invoking the same shader program on different GPU pipeline entities in parallel. For example, the pipeline's vertex processing shader program is simultaneously run on multiple vertices. In addition to processing groups of 32 entities in parallel, the core also interleaves execution of instructions from up to 48 different groups [NVI 2010a]. This form of fine-grained hardware multi-threading allows the core to avoid stalls caused by high-latency memory operations (e.g., texture fetches) by running instructions from any of the 48 groups of entities. The NVIDIA GeForce GTX 480 GPU features 15 of these programmable processing cores [NVI 2009] which provide over 1.3 teraflops of aggregate computing performance: more than nine times the peak capability of a high-end, six-core Intel Core i7 CPU. Similarly, an ATI Radeon HD 5870 GPU uses 20 cores and 64-wide SIMD processing to provide over 2.7 teraflops of peak performance for shader programs [Houston 2008; Fowler 2010].

GPU processing cores trade-off single-threaded performance for the advantages of increased compute density and large-scale multi-threaded execution. Each NVIDIA core described above processes over 1,500 pipeline entities at a time using 32 ALUs (overall, the 15 cores process over 23,000 entities). In this design, logic associated with any one of these shader program invocations takes longer to complete because it shares processor resources many other invocations. As a result, GPU processing cores excel at processing large batches of data-parallel work but they do not efficiently execute latency-sensitive or non-data-parallel components of the graphics pipeline. Rather than limit peak compute capability by designing a more versatile programmable core, GPU architects choose to delegate these computations to fixed-function processing.

Fixed-Function Processing

The fixed-function components of a GPU provide power and area-efficient implementations of pipeline operations not assigned to programmable cores. Fixed-function operations fall loosely into one of the following three categories.

First, fixed-function processing accelerates compute-intensive operations that would require many instructions if implemented on programmable cores. For example, the pipeline's rasterization stage is implemented efficiently in custom hardware using reduced-precision fixed-point operations. Texture filtering and transcendental operations (sin, cos) are additional examples of expensive graphics operations accelerated using fixed-function processing.

Next, GPUs rely on fixed-function processing to perform computations that map poorly to regular, data-parallel execution. Important pipeline tasks like culling, grouping vertices into primitives, data compression, and frame-buffer update involve irregular control, synchronization, and communication when implemented in parallel. These operations cannot be implemented efficiently on the GPU's programmable cores.

Last, fixed-function logic assumes responsibility for dynamically organizing and scheduling pipeline computations onto the GPU's collection of compute resources. For example, scheduling tasks include distributing work to programmable cores, packing entities into blocks of work for data-parallel processing, and coordinating access to input and output streams. Low-latency, fine-grained pipeline scheduling is important because it allows GPUs to run efficiently despite wide and unpredictable variation in pipeline workload characteristics. Variation in load is common as applications frequently change the shader programs used to define stage behavior.

In summary, although a large fraction of a modern GPU's resources reside within programmable processing components, fixed-function processing continues to plays a critical role in the efficiency of modern GPUs. Not only does it provide efficient implementations of compute heavy tasks, it ensures programmable components of GPU are utilized well. Many of the GPU pipeline changes I propose in this dissertation also stand to benefit from implementations that make judicious use of fixed-function processing.

2.2 The Reyes Pipeline

2.2.1 Reyes Architecture

The Reyes rendering architecture was developed at Pixar in the mid-1980s to meet the explicit goal of rendering high-quality images of geometrically complex scenes for use in film [Cook et al. 1987]. Reyes remains the industry standard for film rendering today, and Pixar's implementation of Reyes, RenderMan [Apodaca and Gritz 2000], has served as the core rendering system for all of Pixar's feature-length animated films. A simplified Reyes pipeline is shown below the GPU pipeline in Figure 2.1. The illustrations are aligned to facilitate comparison of the two architectures.

In the Reyes pipeline, there is no equivalent of the GPU pipeline's Coarse Vertex Processing and Patch Generation stages. Reyes assumes final surface patch primitives are generated using separate modeling and animation software and provided directly to the rendering pipeline. Like the GPU pipeline, Reyes tessellates input primitives into polygons. However, Reyes uses an adaptive, two-phase algorithm to produce high-quality tessellations and it seeks to tessellate all surfaces, regardless of their geometric complexity, into high-quality micropolygon meshes (a variant of this tessellation scheme is integrated into the GPU pipeline in Chapter 3). Following tessellation, a programmable Vertex Processing stage allows fine-scale surface detail to be added to the resulting mesh using per-vertex displacements.

The GPU and Reyes pipelines also differ notably in their approach to shading. While the GPU pipeline computes surface shading once for each rasterized fragment, the Reyes pipeline performs a shading computation for each mesh vertex prior to rasterization. Since the sampling rate of both surface position and shading is determined by the density of mesh vertices, Reyes requires micropolygon meshes to achieve high-quality shading (it is often desirable to perform at least one shading computation per image pixel; Reyes requires meshes to contain one vertex per pixel to achieve this shading density). The original Reyes architecture contained only a single Vertex Processing stage for computing both surface displacement and shading at each mesh vertex. However, as shown in the figure, subsequent evolutions of the Reyes pipeline provide separate stages for these two operations. This design allows for optimizations

such as culling to take place after the final position of micropolygon vertices is known, but prior to shading.

Following Vertex Processing, Reyes computes micropolygon-screen coverage (Rast). Then, the pipeline interpolates surface color stored at each micropolygon vertex to determine the color of covered frame-buffer pixels (Pixel Ops). While the Reyes pipeline's Rast and Pixel Ops stages are conceptually similar to their counterparts in the GPU pipeline, they support more advanced rendering features. For example, the Reyes rasterizer's micropolygon-screen coverage computations simulate the effect of camera motion blur and defocus (the cost of adding this feature to a rasterizer is investigated in Chapter 5). Also, unlike the GPU pipeline, which processes all geometry in the order it is received from the application, Reyes guarantees accurate rendering of transparent surfaces by blending rasterized surface samples at the same screen location into the frame buffer in depth order.

2.2.2 Reyes Implementation

Although the design of Reyes coincided with efforts at Pixar to develop parallel hardware systems for graphics [Levinthal et al. 1987], modern Reyes systems (including, to my knowledge, implementations of RenderMan used to create Pixar's feature films) are implemented as software applications running on commodity CPUs. In the film domain, artifact-free rendering quality, flexibility to support advanced rendering techniques, and integration into production work flows (features that save artists time) take precedence over realizing the most highly optimized rendering system. As a result, the Reyes pipeline has not been subjected to the same level of sustained, aggressive optimization as the GPU pipeline.

Even so, reducing Reyes rendering costs is important to film production. Recent RenderMan releases enable support for multi-core CPU execution and researchers have explored the possibility of accelerating Reyes performance using GPUs. For example, Wexler et al. [2005] generate and shade micropolygons using a software Reyes implementation and then render the resulting geometry using the GPU pipeline (to accelerate rasterization and frame-buffer operations). More recently, aspects of the

Reyes pipeline [Patney and Owens 2008], or even simplified versions of the entire pipeline [Zhou et al. 2009; Eisenacher and Loop 2010], have been ported to run on the GPU's programmable cores. However, as was the case with the GPU pipeline, many Reyes pipeline operations do not map efficiently to regular data-parallel processing, resulting in inefficient execution. GPU-based implementations of Reyes do benefit from the high compute capability of GPUs and can achieve higher rendering performance than CPU-based RenderMan implementations, but none of these efforts demonstrate real-time performance on complex film-quality (or even game-quality) scenes.

2.3 Alternative Graphics Pipelines

Historically, the GPU pipeline architecture was the only programming interface that provided graphics applications access to the substantial compute capability of GPUs. However, current parallel programming languages like CUDA [NVI 2010a] or OpenCL [Khronos 2010] allow developers to write software that runs as a "compute mode" program on a GPU's programmable cores (the ports of Reyes described above are examples of such programs). In addition, upcoming compute-optimized multi-core processors like Intel's Larrabee [Seiler et al. 2008] or heterogeneous multi-core chips from Intel (Sandy Bridge) and AMD (Fusion) will also provide teraflops of computing performance and be programmable using traditional parallel programming languages and APIs.

The increasing availability of compute-rich parallel processing platforms has spurred interest in exploring alternatives to the rendering methods used by the GPU and Reyes pipelines. For example, high-performance ray tracing using GPUs and multi-core CPUs is an area of active work [Wald et al. 2007; Aila and Laine 2009; Parker et al. 2010]. Graphics researchers [Pharr et al. 2007; Mark 2008; Seiler et al. 2008] and prominent game developers like Tim Sweeney have argued that a fixed pipeline architecture that requires applications to use specific, pipeline-accelerated rasterization, z-buffering, and anti-aliasing algorithms to achieve good performance diminishes the overall visual experience expert developers can achieve [Sweeney 2009]. Sweeney

calls for graphics platforms to provide flexible processors and programming models that allow game-engine developers complete freedom to choose and implement the algorithms used for rendering.

Clearly, the freedom to develop new high-performance renderers from scratch, or programmatically create custom graphics pipelines using frameworks like GRAMPS [Sugerman et al. 2009], provides exciting opportunities for future innovation in real-time graphics. However, ubiquitous programmable parallelism does not imply that the high level of efficiency achieved through tight co-design of algorithms, heterogeneous hardware processing resources, and the GPU pipeline abstraction is no longer necessary. In fact, looming challenges presented by chip power and area constraints make efficiency increasingly important. While the GPU pipeline's inefficient behavior when rendering micropolygons could be interpreted as further evidence of the need to discard the fixed-pipeline architecture and start anew, the remainder of this dissertation shows that the pipeline can be modified to correct these problems, enlarging its application scope to include micropolygon rendering while also retaining the significant implementation benefits of high-level, domain-specific abstractions.

Chapter 3

DiagSplit Tessellation

The micropolygon rendering pipeline must tessellate surface primitives into micropolygon meshes. As will become increasingly evident throughout this dissertation, performing good tessellation is very important for efficient micropolygon rendering. Clearly, one goal of tessellation is to generate micropolygon meshes that capture complex surface detail accurately. But tessellation properties such as the shape of micropolygons and even the order they are produced are also important to the design and performance of subsequent pipeline stages that perform rasterization and shading.

This chapter describes DiagSplit [Fisher et al. 2009], an algorithm for tessellating parametric surfaces into meshes where nearly all polygons, regardless of surface shape or view, closely approximate an application-specified size. One focus of DiagSplit's design is mesh quality. When configured to produce subpixel-area polygons, DiagSplit produces high-quality meshes that accurately represent complex surfaces and do not contain artifacts like T-junctions or cracks. The other focus is performance. DiagSplit is designed to meet the high-throughput requirements of real-time rendering. It is amenable to fine-granularity parallel execution and it integrates into the GPU pipeline as an extension of existing, highly optimized tessellation mechanisms.

Tessellation Algorithm		Adaptive	Parallel	Crack-Free
NoSplit	(D3D11/OGL4)	no subpatch adaptivity	✓	✓
BinSplit	(Reyes)	overtessellates	✓	✓ (T-junctions)
IsoSplit	(Reyes)	✓		✓
DiagSplit		✓	✓	✓

Table 3.1: Tessellation schemes adopted by the GPU and Reyes pipelines choose different quality and efficiency trade-offs. The DiagSplit algorithm combines the strengths of these existing approaches.

3.1 Tessellation Requirements

Following the design of the GPU pipeline from Section 2.1.1, input primitives to the real-time micropolygon pipeline are defined by parametric patches. For example, the test scenes used in this chapter feature surfaces represented by displaced Bezier patches (see Figure 3.12). Similar to triangle inputs typically provided to GPU pipelines today, surface patches exhibit large variation in size: patches covering only a few pixels and as well as patches covering hundreds or thousands of pixels may be present in the same scene. Given these inputs, tessellation should produce meshes containing polygons (specifically triangles) that are subpixel in area and, ideally, all about the same size.

Undersampling a complex surface (creating polygons that are too large) results in geometric artifacts due to piecewise linear interpolation of surface x-y position (e.g., silhouette errors) and depth (e.g., occlusion errors). The stone house example in Figure 1.1 showed how undersampling a surface also affects lighting and shading, for example, due to incorrect shadowing. In addition, in a pipeline that shades vertices, rather than rasterized fragments (a technique not common in GPUs today but possible in a future micropolygon pipeline: see Chapter 6), undersampling mesh geometry results in low-quality faceted shading.

Oversampling a surface (creating polygons that are too small) hurts rendering performance. Complex surfaces such as displaced, bicubic surfaces are costly to evaluate

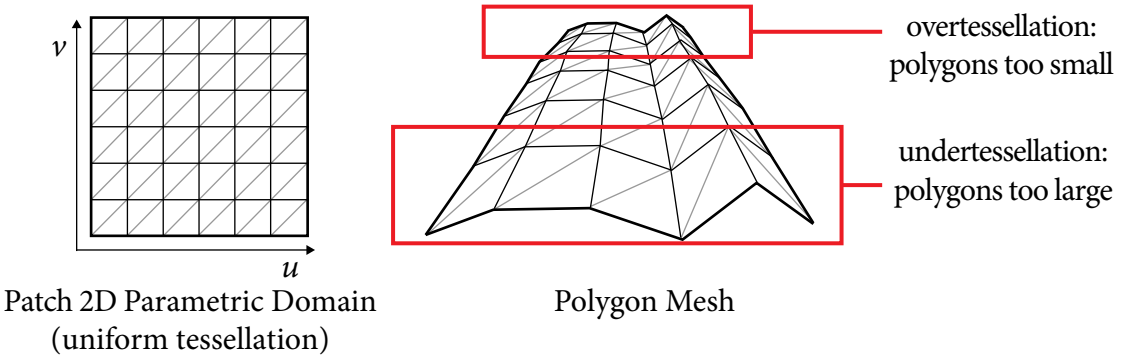


Figure 3.1: Uniform tessellation of a surface results in wide variation in polygon size.

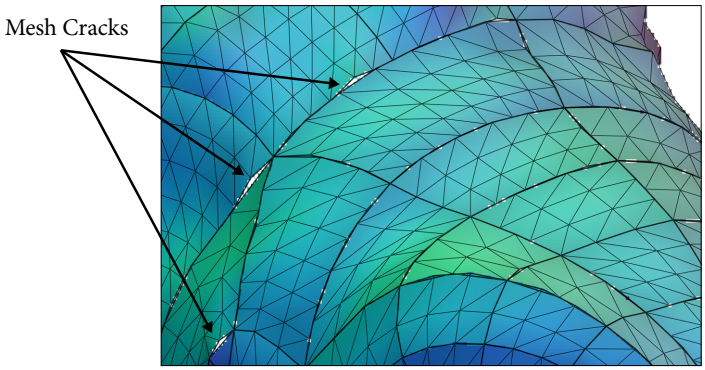


Figure 3.2: A surface tessellation with cracks. Cracks in the surface appear when adjacent tessellations approximate the surface with different numbers of polygons.

so computing surface position and shading attributes at extra mesh vertices introduces significant extra work. A tessellation that oversamples a surface also generates more polygons that must be culled, rasterized, and potentially shaded by the pipeline.

Sampling a surface at any fixed resolution produces a static tessellation that, depending on view, may contain either too few or too many polygons. For example, Figure 3.1 shows a uniform tessellation of a parametric surface; the tessellation partitions the surface equally into five polygons in both parametric directions. The resulting polygons are too large near the camera and too small at a distance. Adaptive tessellation is needed to produce a mesh containing the desired screen density of micropolygons regardless of variation in surface detail or camera location.

In addition to producing polygons that are the right size, adaptive tessellation must avoid frame-to-frame discontinuities ("popping") as tessellation changes in response to object or camera movement, and it must avoid producing mesh cracks. Cracks, as illustrated in Figure 3.2, occur when tessellations of different regions of a surface do not align at region boundaries. Micropolygons are sufficiently small that popping artifacts are rarely visible. Cracks, however, allow surfaces occluded from view to be visible in rendered output, producing objectionable artifacts.

Last, to be viable for real-time use, a tessellation algorithm must achieve high performance. A single input patch may result in thousands of micropolygons, so the process of computing its tessellation must be parallelizable. As shown in Table 3.1, the most widely used tessellation schemes meet some, but not all, of the three tessellation requirements: adaptivity to surface complexity and camera view, crack-free meshes, and parallelizable implementation.

3.2 Background

DiagSplit combines ideas from tessellation schemes used in the Reyes and GPU pipelines. To facilitate comparison, it is useful to introduce all three of these schemes in the context of the Split-Dice algorithm for adaptive surface tessellation.

3.2.1 Split-Dice

Split-Dice is a powerful algorithm, based on the Lane-Carpenter algorithm [Lane et al. 1980], that is used by the Reyes pipeline to tessellate surfaces into micropolygons. The two-phase, divide-and-conquer process of Split-Dice is illustrated in Figure 3.3. The first phase, Split, recursively subdivides patches to create smaller subpatches. In the figure, patch B is split into two subpatches. Splitting terminates when the algorithm estimates that uniform parametric tessellation of all subpatches will produce micropolygons that are approximately the same pre-specified area on screen. Therefore, Split produces subpatches that correspond to local regions of the surface

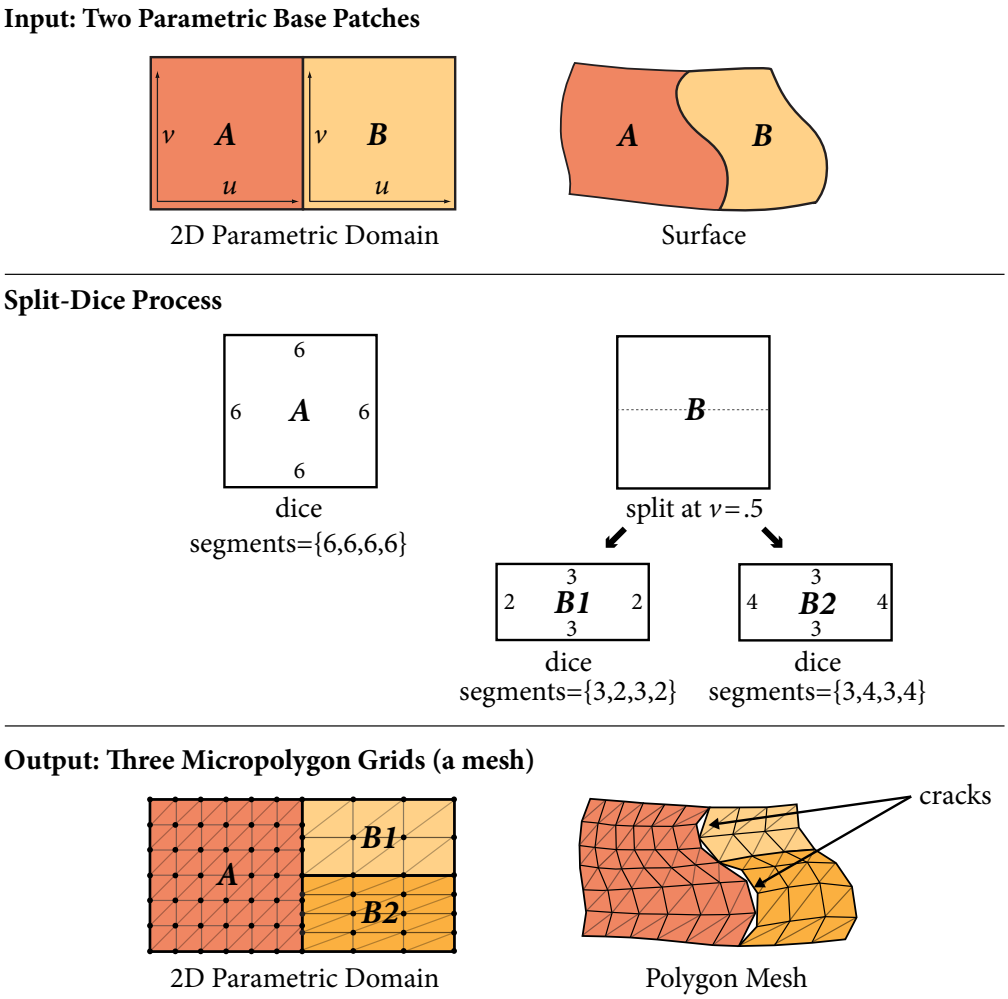


Figure 3.3: Split-Dice tessellation generates three micropolygon grids from two parametric patches. Top: Input surface patches and their corresponding 2D parametric domains. Middle: Patch A is not split, but Patch B is split into subpatches B1 and B2. Bottom-left: Partitioning of A and B's parametric domains into triangles. Tessellations of A and B sample the surface at different domain points (black dots) along their shared boundary. Bottom-right: The resulting triangle mesh; Cracks appear along the A-B boundary.

that are small and flat. Then, the Dice phase uniformly tessellates subpatches generated by Split into micropolygon meshes. In this document, I will borrow terminology from Reyes and refer to each micropolygon mesh produced by dicing a subpatch as a *grid*. Although the term grid originally referred to a regular matrix of quadrilateral micropolygons [Cook et al. 1987], it is used more generally here to refer to any connected micropolygon mesh (tessellation algorithms in this chapter generate triangle micropolygons). The output of Split-Dice is a collection of grids.

Each grid is permitted to have a different density of polygons across the surface, so performing splits allows a tessellation to adapt to variations in the projection of the surface to screen coordinates. The advantage of using both Split and Dice is that Split provides reasonable adaptivity to varying surface complexity, while Dice retains the efficiency of uniform tessellation at subpatch granularity.

The decision to split or dice a patch is made by estimating the variation in the surface's screen-space derivative with respect to each parametric direction. Surfaces whose derivative varies significantly across the patch are not suitable for uniform tessellation and should be split. For example, patches with poorly distributed control points, varying curvature, or which undergo perspective foreshortening (recall Figure 3.1) undergo splits. Surfaces whose derivative is approximately constant across the patch are diced using a number of micropolygons determined by this constant. For many surface types, such as Bezier patches [Catmull 1974; Blinn 1978; Clark 1979], it is possible to compute surface derivatives analytically. However, analytic techniques only approximate surface derivatives when the surface is displaced.

Dice lends itself to data-parallel execution because surface position and attributes at each vertex in the output mesh can be evaluated in parallel. In contrast, Split presents two challenges that make high-performance implementation difficult. First, much like rasterization of large triangles, Split performs unbounded data amplification, potentially generating a large number of subpatches from a single base primitive. Second, it complicates crack avoidance by dynamically introducing new boundaries between subpatches in addition to boundaries between the original base primitives. Split-Dice implementations must ensure the final mesh is crack-free along these new edges.

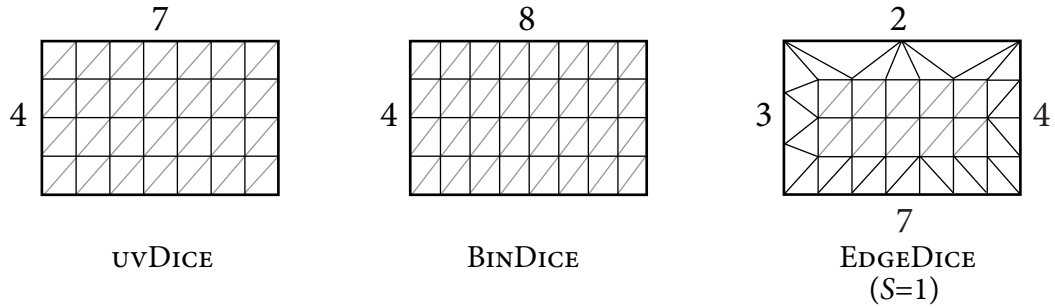


Figure 3.4: Comparison of dicing methods: The Reyes pipeline's implementation of Dice (uvDice) accepts unique tessellation factors for a patch's u and v parametric directions. It produces a uniform mesh tessellation according to these factors. BinDice is similar, but constrains tessellation factors to power-of-two values. Dicing in the GPU pipeline (EdgeDice) accommodates unique per-edge factors and an interior scaling factor S. Intuitively, grids produced by EdgeDice contain a uniform interior mesh stitched to uniform length segments along grid edges.

To understand how cracks occur, consider the boundary between adjacent surface patches A and B in Figure 3.3. Patch A is not split during tessellation. It is diced uniformly using six micropolygons along each edge. Patch B is split at its midpoint in the v parametric direction creating subpatches B1 and B2. B1 and B2's span of the A-B edge is diced using two and four micropolygons respectively. As a result, Patch A's tessellation along this edge does not match that of Patch B (bottom-left). Different samplings of the curved surface cause cracks in the final micropolygon mesh (bottom-right).

3.2.2 Reyes Tessellation

The Reyes pipeline directly implements the Split-Dice algorithm. Reyes divides application-provided surface base patches into subpatches by partitioning along isoparametric directions. Throughout the remainder of this chapter, Reyes-style isoparametric splitting is referred to as IsoSplit (Table 3.1–third row). The dice phase tessellates each subpatch emitted by Split into a regular grid of micropolygons by uniformly partitioning the subpatch in each parametric direction. The output of this implementation of dicing, referred to as uvDice, is illustrated in Figure 3.4-left. The

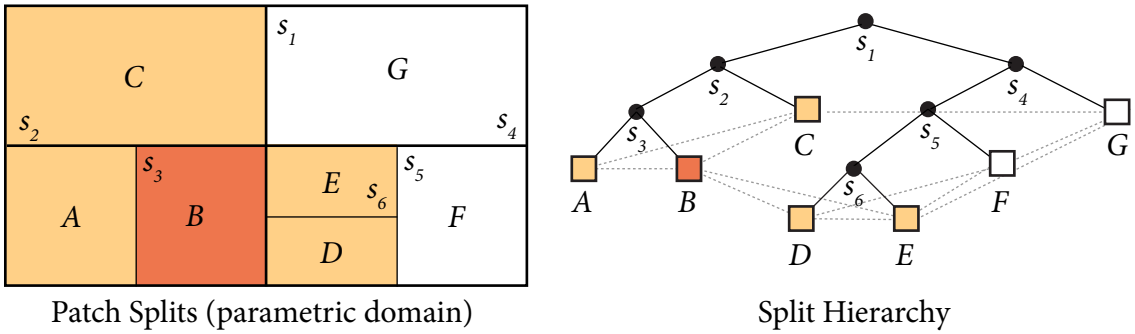


Figure 3.5: Grid stitching eliminates cracks but creates dependencies between subpatches created by IsoSplit. Left: A patch's parametric domain is partitioned into seven subpatches (the six splits are labeled $s1$-$s6$). Right: subpatch adjacency needed for stitching is encoded as a tree. Vertex position information from subpatches A, C, D, and E must be maintained in memory to stitch subpatch B to its neighbors. Dependencies between subpatches are shown as gray dotted lines.

decisions of whether to split or dice, and, if dicing, the number of segments to tessellate in each direction, are determined by computing derivatives across the subpatch interior.

Many approaches have been taken to produce a crack-free mesh using Reyes. Some Reyes implementations [Apodaca and Gritz 2000; Foster 2009], most notably Pixar's RenderMan, fix cracks by leveraging information from adjacent subpatches to "stitch" grids together once final vertex positions are known. Figure 3.6-left shows a strip of triangles used to stitch micropolygons from two adjacent uvDice grids (stitch triangles are colored red). Stitching implementations maintain adjacency information during the splitting process by storing pointers between grids and maintaining structures, like those shown in Figure 3.5, to represent splitting decisions. Interior nodes of this tree correspond to patch splits. Leaf nodes correspond to diced subpatches. In this example, fixing cracks along the edges of subpatch B using stitching requires access to vertex information from four neighboring subpatches (A,C,D,E). Therefore, stitching introduces dependencies between subpatches as indicated by the dotted lines. The positions of vertices along subpatch B's boundaries cannot be determined until the tessellation of subpatches A,C,D,E is known. Conversely, positioning vertices in grids corresponding to subpatches A,C,D,E depends upon the tessellation of B. These

dependencies make it difficult to stream subpatch data through the graphics system (subpatch edge information must be retained in memory until the entire base primitive is complete), bloating working sets and preventing large-scale parallelism. For this reason, crack fixing by maintaining adjacency information is not a viable solution for a real-time graphics pipeline.

Other Reyes implementations avoid cracks without maintaining subpatch adjacency information. Instead, they constrain dicing to use only power-of-two tessellation factors (binary dicing, see BinDice, Figure 3.4) and modify splitting to ensure subpatches agree on the number of segments a surface tessellation requires along shared boundaries. Binary dicing guarantees that subpatches diced at a higher resolution than the boundary requirement still contain some vertices that align perfectly with boundary segment vertices (Figure 3.6-center, black dots). To avoid cracks, tessellation positions extra grid vertices (red dots) to lie on boundary segments by linearly interpolating the position of boundary-aligned vertices ("pasting") [Apodaca and Gritz 2000]. Although pasting is effective, the resulting meshes contain T-junctions and is not watertight. A Split-dice implementation that relies on binary dicing is referred to as BinSplit (Table 3.1–second row). While binary dicing is attractive due to its simplicity, it unfortunately results in poor tessellations. For example, if a patch is optimally diced with 12 segments in one parametric direction, binary dicing requires it to either be diced with eight segments (undertessellation) or 16 segments (overtessellation). In practice, enabling binary dicing and rounding tessellation factors up to the nearest power of two increases tessellation polygon count more than two times. It is desirable to avoid this inflation of polygon count in a real-time graphics pipeline.

3.2.3 GPU Tessellation

Tessellation functionality supported by the Direct3D 11 and OpenGL 4 pipelines does not provide the ability to split base primitives, but provides three pipeline stages [Mic 2010b; Segal and Akeley 2010] that cooperate to provide a flexible implementation of Dice (this scheme is referred to as NoSplit, Table 3.1–first row). The GPU pipeline's Patch Generation stage emits surface patches parametrized on either a quadrilateral

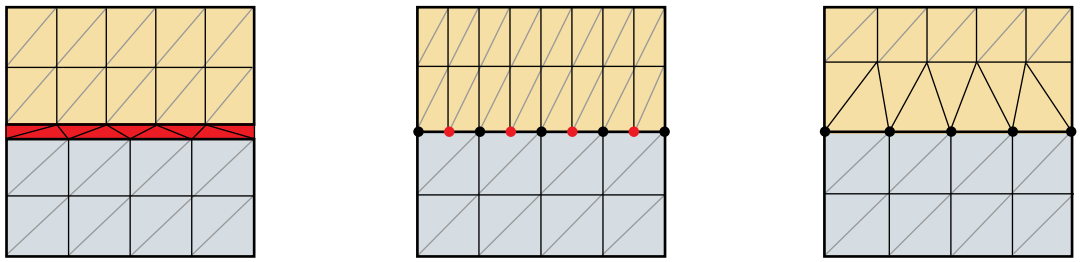


Figure 3.6: Three strategies for eliminating cracks between adjacent grids with different tessellations: Left: Stitching inserts a strip of triangles to connect two uvDice grids. Center: Pasting, used in conjunction with binary dicing, positions extra grid vertices (red dots) on segments that connect vertices shared by both grids. Right: EdgeDice matches different interior grid tessellations to segments defining the surface tessellation along grid boundaries.

or triangular domain and surface tessellation factors along each domain edge. Then, a fixed-function Uniform Tessellation stage generates a mesh with vertices at domain points (u, v) determined by the tessellation factors [Moreton 2001]. Last, the Vertex Processing stage evaluates the surface's position and custom vertex attributes at each point, yielding a renderable mesh.

In contrast to uvDice, which utilizes only two independent tessellation factors (one for each parametric direction), the Uniform Tessellation stage generates a triangle mesh from four independent factors (one for each domain edge). This more flexible dicing strategy, called EdgeDice, is illustrated in Figure 3.4-right. In EdgeDice, both the edges and the interior of the patch are uniformly tessellated in the parametric domain. The number of interior segments in a given parametric direction is taken to be the maximum of the two opposing edge factors, scaled by an interior tessellation scale parameter S between 0 and 1 (although the Uniform Tessellation stage allows different scaling factors in each parametric direction, only isotropic scaling is used in this work). Triangles along the edge of the tessellation stitch the uniform interior to edge segments.

GPU tessellation is designed for real-time performance. Each base primitive is processed independently, enabling parallelism. EdgeDice is implemented efficiently

in fixed-function hardware and surface evaluation at mesh vertices is data-parallel. However, the performance benefits of this design are tempered by three notable constraints.

First, as stated above, the scheme lacks the adaptability of split (DiagSplit will overcome this limitation).

Second, the three-stage GPU tessellation architecture requires surface primitives to support arbitrary parametric evaluation. This enables surface evaluation to be expressed as a Vertex Processing shader program that operates on a single parametric location. However, the convenience of this per-vertex abstraction prevents the use of efficient forward differencing schemes to efficiently evaluate uniformly spaced domain points [Lien et al. 1987]. While many subdivision surfaces can be evaluated directly [Stam 1995], these techniques often require one to two levels of subdivision to meet the conditions for direct evaluation; This diminishes the benefit of tessellation. Furthermore, direct evaluation of patches with extraordinary vertices is more computationally expensive than that for regular bicubic patches. Fortunately, recent work has developed schemes to approximate subdivision surfaces (including support for creases and corners) using Bezier and Gregory patches [Loop and Schaefer 2008; Kovacs et al. 2009; Loop et al. 2009]. These efficient parametric approximations are the surface representation most frequently used with GPU tessellation.

Third, independent patch processing requires special care by application developers to prevent cracks. The parametric location of vertices generated by the Uniform Tessellation stage is determined entirely by a patch's edge and interior scaling factors. Thus, to produce tessellations that align at patch boundaries, Patch Generation shader programs must produce identical tessellation factors for shared edges. Also, Vertex Processing shader programs must always compute the same surface position for a parametric point along an edge, regardless of which edge-adjacent patch the vertex was generated from. These properties are non-trivial for a shader author to guarantee. A particularly tricky, but common, case arises when displacement maps are represented using texture atlases. When the boundary between two patches coincides with a texture atlas seam, the same parametric point on the edge has different texture coordinates in each patch. Filtered displacement map lookups during Vertex

Processing may not produce the same result, leading to inconsistent evaluation of surface position at edge vertices. Displacement maps must be carefully authored and Patch Generation and Vertex Processing shader programs must adhere to strict edge-ordering policies and precisely order floating-point arithmetic to ensure consistent evaluation of tessellation factors and vertex positions [Ni et al. 2009].

3.3 DiagSplit Algorithm

DiagSplit is a variant of the Split-Dice algorithm designed for use in the real-time micropolygon rendering pipeline. It adapts well to surface complexity, does not rely on stitching across subpatch boundaries to eliminate cracks, and does not incur the overtessellation of binary dicing.

Following the NoSplit and BinSplit algorithms described in Section 3.2, DiagSplit determines surface tessellation along subpatch edges using only properties of the edge. Subsequently, DiagSplit determines surface tessellation of the subpatch interior using the Split-Dice algorithm. The interior tessellation is guaranteed to match the previously determined edge behavior, ensuring that there are no T-junctions or cracks. DiagSplit meets this requirement via two significant modifications to the Split-Dice implementation in Reyes:

- DiagSplit is permitted to split subpatches along non-isoparametric directions (hence the name DiagSplit). Non-isoparametric splits occur *only* when necessary to prevent cracks.

- DiagSplit requires EdgeDice dicing to stitch tessellations of subpatch interiors to the tessellation required along edges.

In addition to these changes, DiagSplit also considers the final, displaced position of the surface when computing edge tessellation factors and sets EdgeDice's interior tessellation scale parameter S to produce micropolygons that closely approximate an application-specified target area.

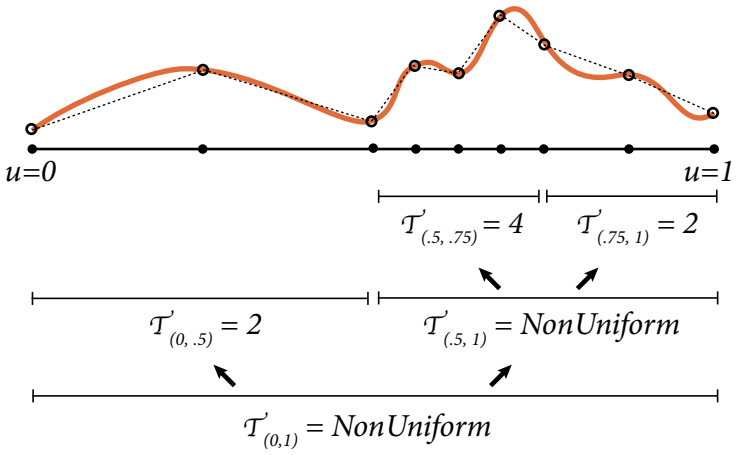


Figure 3.7: DIAGSPLIT produces tessellations that are piecewise-uniform along edges. In this example, tessellation along an edge over the 0–1 parametric domain is determined by $\mathcal{T}$. $\mathcal{T}$ computes an integer tessellator factor when the surface should be tessellated uniformly along an edge. Otherwise, DIAGSPLIT will perform a split dividing the edge at its parametric midpoint and $\mathcal{T}$ is used to set the tessellations along the two partitions.

3.3.1 DiagSplit

Recall that the GPU pipeline's Patch Generation stage computes uniform tessellation factors for all patch edges. DIAGSPLIT defines the tessellation along an edge using the function $\mathcal{T}$. Given an edge, $\mathcal{T}$ either designates that the edge can be uniformly diced using t segments, or that non-uniform tessellation along the edge is necessary. If non-uniform tessellation is required, the edge is partitioned at its parametric midpoint, and $\mathcal{T}$ is used to determine the tessellation along each partition. This process, applied recursively, fully determines a piecewise-uniform adaptive tessellation along the original edge (Figure 3.7).

DIAGSPLIT ensures that the tessellation of the interior of the patch generated by the Split-Dice process conforms to the behavior along edges dictated by $\mathcal{T}$. This guarantee holds for edges of base patches and also holds recursively for subpatch edges introduced by splits.

DIAGSPLIT's behavior is simple when $\mathcal{T}$ dictates that tessellation along all edges of a subpatch is uniform. The subpatch is diced using EDGEDICE according to

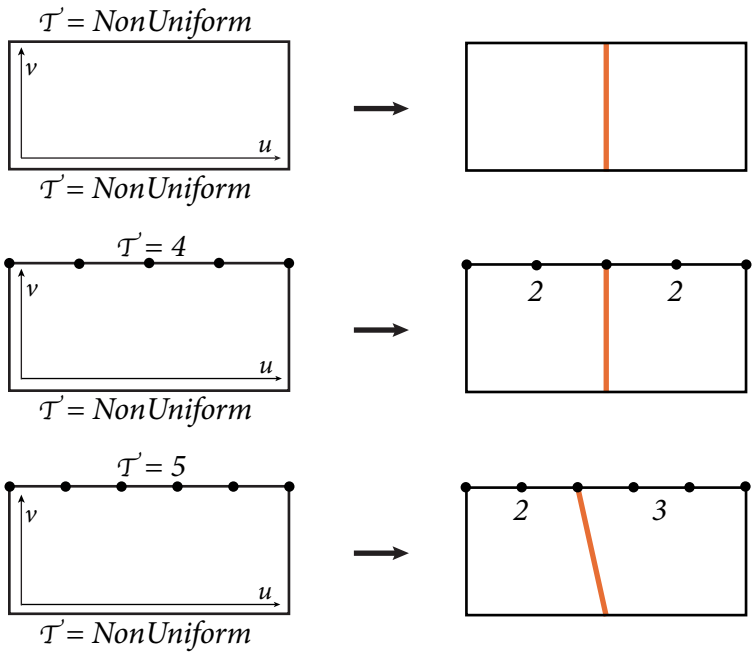


Figure 3.8: DIAGSPLIT produces tessellations that adhere to edge behavior defined by $\mathcal{T}$. Top: When tessellation along both edges in the same parametric direction is non-uniform, DIAGSPLIT splits the subpatch along the line through the midpoint of both edges. Middle, Bottom: When only one of the edges in a parametric direction requires a split, the split runs through the vertex on the opposite edge closest to the edge midpoint. Subedges of the uniform edge are constrained to ensure agreement with the tessellation factor dictated by $\mathcal{T}$. Bottom: Partitioning a uniform edge requiring an odd number of tessellation segments requires a non-isoparametric split.

the four edge tessellation factors. If at least one subpatch edge requires non-uniform tessellation, the subpatch cannot be diced and must be split. Consider the case where the tessellation must be non-uniform along both edges in the u parametric direction (Figure 3.8-top). In accordance with the edge behavior described above, DIAGSPLIT will split the subpatch along the line connecting the parametric midpoint of each edge (orange line). Notice that this behavior is equivalent to that of ISOSPLIT.

When only one edge in the u parametric direction forces non-uniform tessellation (Figure 3.8-middle, bottom), DIAGSPLIT will split the subpatch along the line between the parametric midpoint of the non-uniform bottom edge and *some point* in the uniform tessellation along the top edge. Our implementation chooses the point closest

to the parametric midpoint. If the edge tessellation factor for the top edge is odd, this split occurs along a non-isoparametric line (a diagonal in parametric space). As a result, DiagSplit can generate subpatches with non-isoparametric quadrilateral domains. When splitting is required along both parametric directions, DiagSplit is free to choose which split to perform first or it can implement a split that directly produces four subpatches.

Following the behavior described above, DiagSplit (like GPU tessellation) generates tessellations where adjacent subpatch interior regions are connected using two rows of triangles [Rockwood et al. 1989]. Each subpatch's uniform interior tessellation is independently stitched to segments along its own edges by EdgeDice. Adjacent subpatches stitch to the same segments along their shared edge, so no stitching across subpatches is required to prevent cracks.

Pseudocode for a recursive implementation of DiagSplit operating on quadrilateral domain subpatches is given in Figure 3.9. The function *DiagSplit* carries out the splitting procedure given a subpatch defined by its four parametric corners (*SubPatch*) and tessellation factors (*EdgeFactors*) for all edges (note that edge tessellation factors can take on the special value *NonUniform*). The subroutine *PartitionEdge* is used to compute split points and tessellation factors for new subedges when splits occur. The provided implementation easily extends to triangle domains where each triangle subpatch is split into triangular subpatches. Although not shown in the example, in the rare condition that an edge with a tessellation factor of one is partitioned in a split, DiagSplit produces a triangular child subpatch and proceeds with triangle-domain tessellation.

One aspect of the *DiagSplit* pseudocode was not discussed in the description of the DiagSplit algorithm above. When partitioning an edge that is assigned a uniform tessellation factor (the "else" clause of *PartitionEdge*), *DiagSplit* derives tessellation factors for subedges without additional calls to $\mathcal{T}$. This ensures that subpatch tessellations together contain exactly t uniform segments as required. Calling $\mathcal{T}$ to determine the tessellation along each subedges is incorrect in this case, as there is no guarantee the resulting edge factors sum to the value t. Depending on the values of the edge tessellation factors passed to *DiagSplit*, between one and five calls to $\mathcal{T}$ are

made for each subpatch split.

The GPU pipeline's EdgeDice implementation positions mesh vertices within the $[0, 1]^2$ domain. Although DiagSplit produces subpatches spanning non-isoparametric quadrilateral domains, EdgeDice can be used unchanged if the parametric location of subpatch corners is propagated to the pipeline's Vertex Processing stage. The Vertex Processing shader program uses these coordinates to map EdgeDice's output in the unit square to the base patch's parametric domain (using bilinear interpolation). The surface is then evaluated directly at these parametric locations.

3.3.2 Computing Edge Tessellation Factors

DiagSplit's tessellation quality depends heavily on the implementation of $\mathcal{T}$. Ideally, $\mathcal{T}$ should be inexpensive to compute and accurately estimate the number of segments needed for a tessellation to closely approximate surface position along subpatch edges. In accordance with the tessellation requirements from Section 3.1, the screen spacing of vertices along a subpatch edge should be approximately uniform and not exceed a maximum-specified screen space length (to generate micropolygons, this length is the width of a pixel, or less).

Both analytic and sampling-based methods have been used to compute edge tessellation factors. For example, it is possible to bound surface derivatives of a Bezier patch directly from patch control points and use these bounds to compute conservative tessellation factors for an edge [Clark 1979; Rockwood et al. 1989]. Large variation in surface derivative along an edge indicates a need for non-uniform sampling. In this case, the subpatch containing the edge should be split. One disadvantage of this analytic approach is that it does not account for the displaced position of the surface. Also, it is more complicated to apply to non-isoparametric subpatch edges. Non-isoparametric edges of a Bezier patch are not Bezier curves. Analytic techniques can be applied to non-isoparametric edges, albeit at higher computational cost. For example, one way to bound the tessellation factor for a non-isoparametric edge is to compute derivative bounds for the surface corresponding to an isoparametric region surrounding the edge.

DiagSplit($SubPatch = \{P_{00}, P_{10}, P_{11}, P_{01}\}$,
$EdgeFactors = \{t_{v=0}, t_{u=1}, t_{v=1}, t_{u=0}\}$)

if $t_{v=0}$ or $t_{v=1} = NonUniform$

$\{P_{v=0}, t^a_{v=0}, t^b_{v=0}\} \leftarrow PartitionEdge(P_{00}, P_{10}, t_{v=0})$
$\{P_{v=1}, t^a_{v=1}, t^b_{v=1}\} \leftarrow PartitionEdge(P_{01}, P_{11}, t_{v=1})$
$t_{split} \leftarrow \mathcal{T}(P_{v=0}, P_{v=1})$
$DiagSplit(\{P_{00}, P_{v=0}, P_{v=1}, P_{01}\}, \{t^a_{v=0}, t_{split}, t^a_{v=1}, t_{u=0}\})$
$DiagSplit(\{P_{v=0}, P_{10}, P_{11}, P_{v=1}\}, \{t^b_{v=0}, t_{u=1}, t^b_{v=1}, t_{split}\})$

else if $t_{u=0}$ or $t_{u=1} = NonUniform$

...

else

emit $\{SubPatch, EdgeFactors\}$ to Uniform Tessellation stage

PartitionEdge($P_{\text{start}}, P_{\text{end}}, EdgeFactor = t$)

if $t = NonUniform$

$P \leftarrow (P_{\text{start}} + P_{\text{end}})/2$
$t_0 \leftarrow \mathcal{T}(P_{\text{start}}, P)$
$t_1 \leftarrow \mathcal{T}(P, P_{\text{end}})$

else

Choose vertex index $I = floor(t/2)$ as split vertex
$P \leftarrow$ Parametric coordinates of vertex I
$t_0 \leftarrow I$,
$t_1 \leftarrow t - I$

return $\{P, t_0, t_1\}$

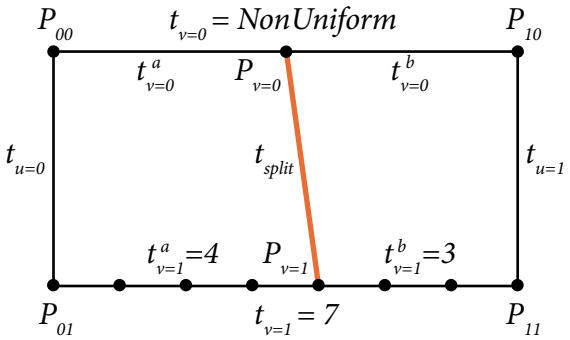


Figure 3.9: The DIAGSPLIT splitting algorithm: *DiagSplit* accepts as input a subpatch (defined by its four parametric corners and four edge tessellation factors) and emits subpatches that can be diced into grids containing near-uniform area polygons.

$\mathcal{T}(P_{\text{start}}, P_{\text{end}})$
  for $i = 0$ to $N - 1$
    $P_i = P_{\text{start}} + (i/(N-1)) * (P_{\text{end}} - P_{\text{start}})$
    $L_i = \mathit{ToScreenCoords}(P_i) - \mathit{ToScreenCoords}(P_{i-1})$
  $t_{\min} = \left\lceil (\sum_{i=1}^{N-1} L_i)/R \right\rceil$
  $t_{\max} = \lceil (N * \max_i(L_i)/R) \rceil$
  if $t_{\max} - t_{\min} \geq \mathit{SplitThreshold}$
    return *NonUniform*
  else
    return $t_{\max}$

Figure 3.10: $\mathcal{T}$ samples surface position at N points along the edge to estimate whether uniform tessellation using t segments will yield vertices spaced on screen by approximately R pixels (for micropolygons, R=1). If $\mathcal{T}$ estimates uniform tessellation cannot meet this goal, it returns the value *NonUniform* to trigger a subpatch split.

The implementation of $\mathcal{T}$ given in Figure 3.10 does not use analytic methods. It coarsely samples the screen-space position of the surface at N uniformly spaced parametric locations along the edge. Then, it approximates the surface along the edge as a piecewise-linear curve consisting of $N - 1$ segments connecting the points (segment lengths are given by L_i's in the pseudocode). The sum of all segment lengths constitutes a lower bound on the edge's actual screen-space length. Using this length estimate, $\mathcal{T}$ computes $t_{\min}$, a lower bound for the edge tessellation factor. It also estimates a tessellation factor upper bound, $t_{\max}$, based upon the longest of the $N-1$ segments. When the difference between $t_{\min}$ and $t_{\max}$ exceeds an application-specified *SplitThreshold*, $\mathcal{T}$ concludes uniform tessellation along the edge is inadequate. Otherwise, $\mathcal{T}$ returns $t_{\max}$ as the tessellation factor for the edge.

To ensure a crack-free mesh, evaluation of $\mathcal{T}$ on an edge shared between two subpatches must always return the same result regardless of which subpatch is currently being processed by *DiagSplit*. Therefore, $\mathcal{T}$ must depend only on surface properties along the edge. It cannot depend on properties associated with a subpatch's interior region (this information is not available to both subpatches). Implementations of $\mathcal{T}$ used with DiagSplit must use techniques established for GPU tessellation [Ni et al. 2009], described previously in Section 3.2.3, to ensure consistent numerical evaluation.

3.3.3 Adjusting Tessellations of Subpatch Interiors

One drawback of making splitting decisions using only surface information along subpatch edges is the inability to make guarantees about tessellation quality for a subpatch's interior. For example, $\mathcal{T}$ bounds segment length, resulting in tessellations that sample a surface at the desired density in the direction of subpatch edges. But it does not prevent oversampling along other screen directions when a subpatch has poor screen-space aspect ratio. This behavior is evident in the DiagSplit tessellation at left in Figure 3.11. Bold lines in the figure indicate subpatch boundaries (grid boundaries). Subpatches near object silhouettes have poor aspect ratio due to perspective. Although vertices along subpatch boundaries are spaced about a pixel apart, the resulting triangles are very small. The size of tessellation triangles, relative to a 0.5-pixel target area, is visualized in the renderings at bottom. Pixels covered by 0.5-pixel-area triangles are green.

To achieve a more uniform distribution of triangle areas across entire objects and scenes, DiagSplit adjusts subpatch interior tessellations by modifying EdgeDice's interior scale parameter $S \in [0, 1]$. S isotropically scales the interior tessellation in each parametric direction. Varying S controls the number of triangles in the final tessellation and, correspondingly, the area of these triangles. The results of interior-area scaling are shown in the top-right of Figure 3.11. Vertex positions on subpatch boundaries are not changed by area scaling (doing so would result in cracks).

DiagSplit computes S using a coarse estimate of the subpatch's screen-space area. Prior to dicing a subpatch, DiagSplit evaluates surface position at a 3×3 grid of points (the grid consists of the subpatch's corners, its edge midpoints, and its center). These points define four quadrilaterals, and the subpatch's area, A_{patch}, is estimated to be four times the area of the largest of these quads. To approximate the target micropolygon area of A_{tri} pixels, DiagSplit sets S so that the subpatch's diced grid contains $A_{\text{patch}}/A_{\text{tri}}$ triangles.

Given subpatch edge tessellation factors a, b, c, d the number of triangles contained in a quadrilateral-domain EdgeDice grid is (M_{u} and M_{v} are the maximum factors in the u and v parametric directions):

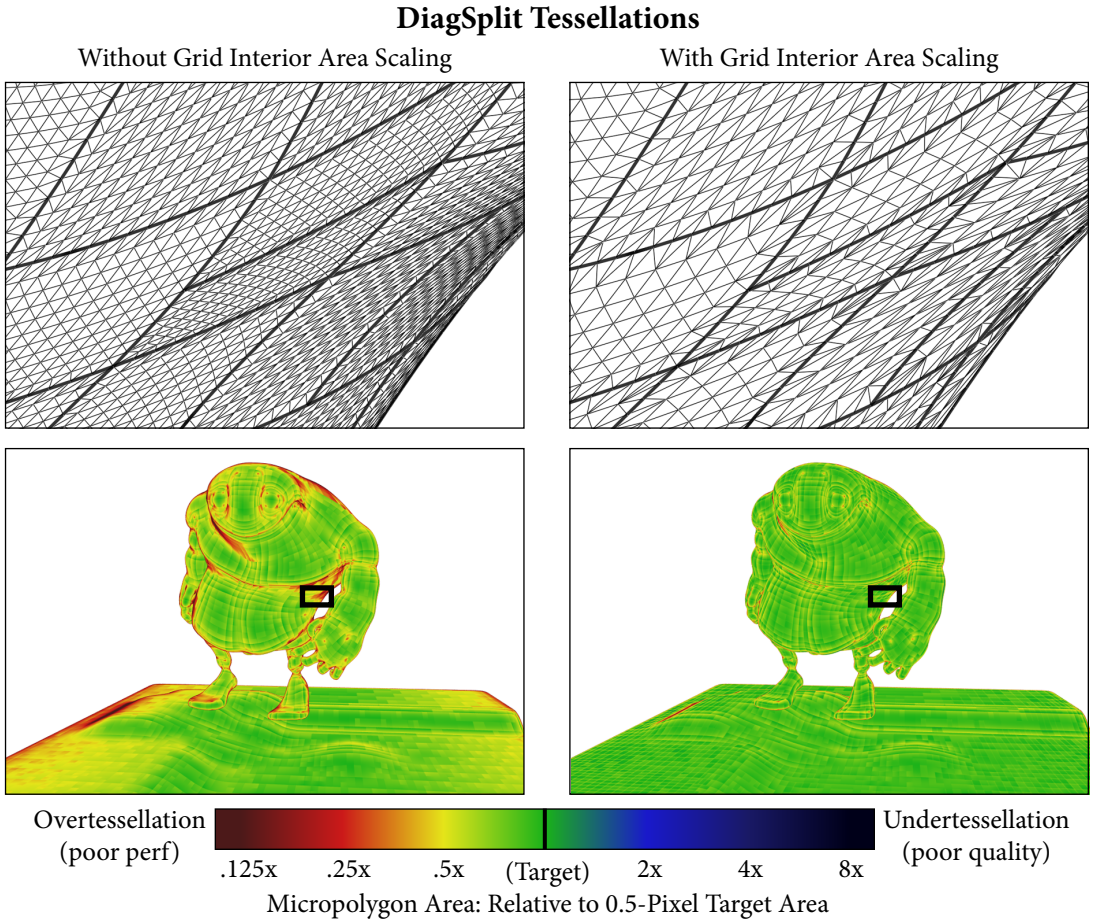


Figure 3.11: Interior-area scaling normalizes triangle areas by adjusting the EdgeDice interior tessellation scale factor S for each subpatch. Top-Left: Micropolygon mesh from a DiagSplit tessellation that does not use interior area scaling. Grid boundaries are shown as dark lines. Bottom-Left: Visualization of triangle size. The tessellation contains triangles that are smaller than the target size when subpatches have poor screen aspect ratio (red regions). Right: DiagSplit with interior-area scaling produces a mesh containing uniform-area triangles. In this example, interior-area scaling also reduces triangle count by 28%.

$$\begin{aligned} N_{\text{tris}} &= 2((SM_{\text{u}} - 2)(SM_{\text{v}} - 2) + \\ &\quad (SM_{\text{u}} - 2) + (SM_{\text{v}} - 2)) + a + b + c + d \end{aligned}$$

Given a desired triangle count, the interior tessellation scaling factor is determined by solving the above equation for S.

Estimating the interior scaling factor S makes three assumptions: a 3×3 uniform grid is a good estimate of the subpatch's actual screen area, triangles in the grid produced by dicing all have the same area (splitting makes this a safe assumption), and integer rounding of interior tessellation factors after scaling by S does not severely impact the tessellation. Nevertheless, as shown by right side of Figure 3.11, interior-area scaling produces very uniform-area micropolygons in practice.

DiagSplit tessellation with interior-area scaling prioritizes generating triangles that closely match a target area over bounding the spacing between vertices in subpatch interiors. The entire splitting process is carried out based on micropolygon edge lengths and interior-area scaling serves as a final "correction" to subpatch interior tessellations based on an estimate of micropolygon area. Interior-area scaling causes micropolygons within subpatch interiors to have longer edges than the lengths determined by $\mathcal{T}$ along subpatch boundaries. As a result, interior-area scaling reduces oversampling due to poor subpatch aspect ratio, but also increases the risk of undersampling surface detail in the interior areas of these patches. Depending on scene characteristics, interior-area scaling reduces overall micropolygon count in tessellations by 10% to 60% (it yields a 28% reduction in triangle count in Figure 3.11).

3.4 Evaluation

We evaluated DiagSplit by comparing its quality and performance against the alternative tessellation algorithms described in this chapter. For high visual quality, we sought to produce triangle micropolygons that are approximately 0.5 pixels in area. At this resolution, the distance between neighboring vertices in diced grids is about

one pixel. For high performance, DiagSplit must avoid overtessellation and the cost of performing splits must be kept low.

Our DiagSplit implementation uses the splitting algorithm described in Section 3.3, EdgeDice with interior-area scaling for dicing, and an implementation of $\mathcal{T}$ that samples an (optionally displaced) surface at four points along subpatch edges (N=4, R=1, *SplitThreshold*=3). To constrain footprint, diced grids contain at most 256 vertices. Splitting continues until subpatches are small enough to meet this dicing constraint. By construction, DiagSplit permits subpatch-parallel execution. We compared this configuration of DiagSplit against the following three alternatives listed in Table 3.1.

- NoSplit: This configuration mimics the behavior of GPU pipeline tessellation. It is subpatch-parallel, does not perform splits, uses EdgeDice dicing with interior-area scaling, and uses a similar edge-based $\mathcal{T}$ as DiagSplit. Unlike DiagSplit, surface evaluations by $\mathcal{T}$ do not account for displacement.

- BinSplit: This configuration performs isoparametric splitting with binary uvDice dicing. Dicing factors are computed using $\mathcal{T}$ (without displacement), but are rounded up to the nearest power of two. Like DiagSplit and NoSplit, BinSplit is amenable to subpatch-parallel execution. When configured to output micropolygons, tessellation algorithms proposed by Eisenacher et al. [2009] and Patney [Patney and Owens 2008; Patney et al. 2009], produce tessellations similar to BinSplit (although [Patney and Owens 2008] contains no mechanism for preventing cracks).

- IsoSplit: This configuration mimics an advanced Reyes implementation and performs isoparametric splitting and uvDice dicing. IsoSplit evaluates the surface at a 4×4 grid of points spanning the subpatch to make splitting decisions and determine dicing factors. Like NoSplit and BinSplit, this evaluation does not account for displacement. Post-tessellation stitching removes cracks, but requires complex data structures that preclude efficient parallel implementation. Stitch geometry is not included in tessellation statistics in this evaluation.

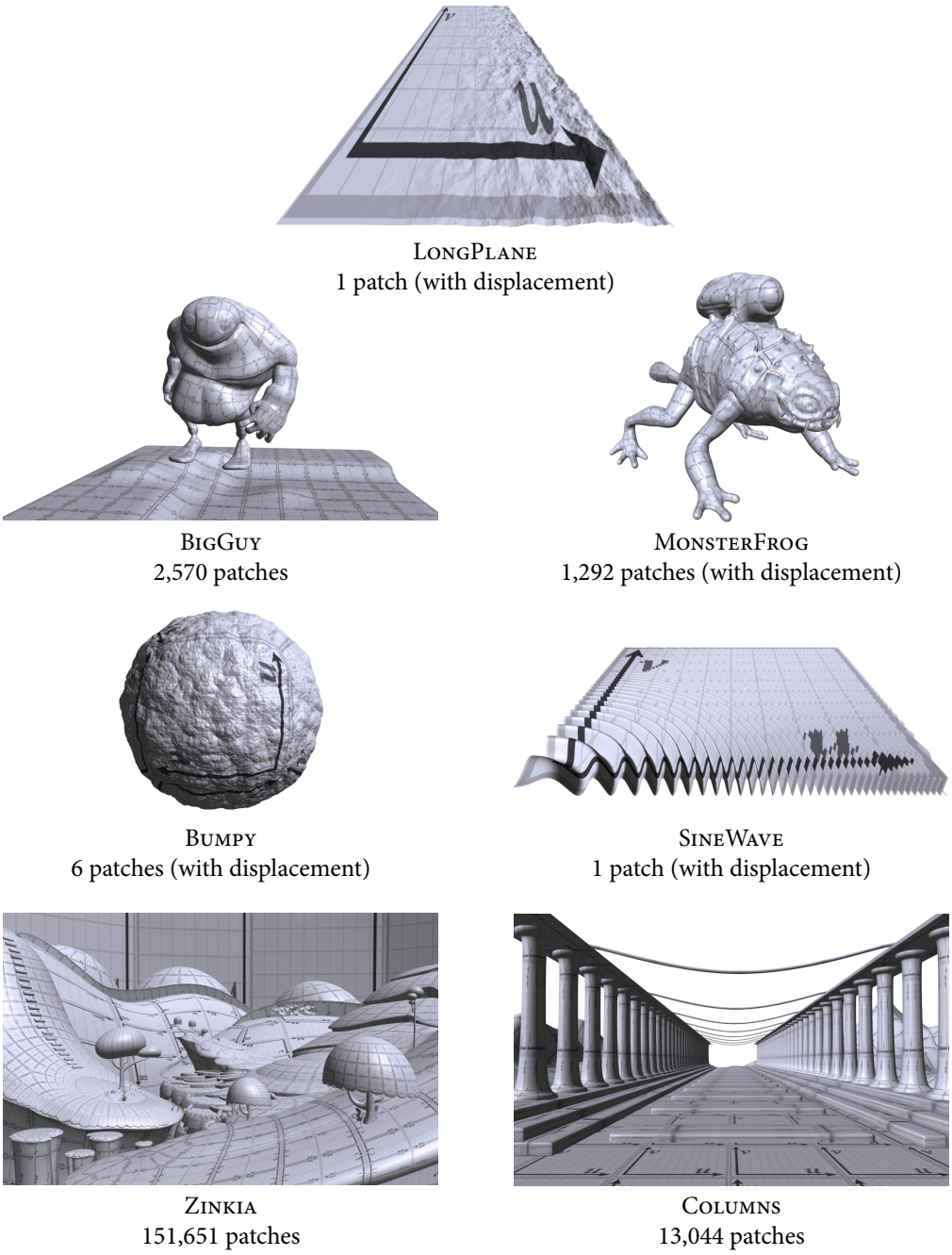


Figure 3.12: Renderings of the seven scenes used to evaluate DiagSplit visualize the parametric coordinate system of base patches. The LongPlane, MonsterFrog, Bumpy, and SineWave scenes contain displaced surfaces. (Zinkia scene © Zinkia Entertainment, S.A.).

We evaluated the four algorithms using the seven example scenes shown in Figure 3.12 rendered at 1728×1080 resolution. To assist understanding of scene properties, the coordinate system of base patches is shown in the renderings. Scene geometry is modeled using Catmull-Clark subdivision surfaces that are directly evaluated using the Loop-Schaefer approximation scheme [Loop and Schaefer 2008]. The seven scenes test different aspects of tessellation:

- LongPlane consists of a single large patch undergoing severe perspective foreshortening. Surface detail is supplied using textured displacement (the magnitude of displacement grows in the u parametric direction). LongPlane is a pathological case for NoSplit and requires adaptive tessellation to avoid substantial undertessellation near the camera and overtessellation at a distance.

- BigGuy and MonsterFrog are simple character tests. BigGuy's surface is smooth but MonsterFrog features textured displacement.

- Bumpy and SineWave use high-frequency procedural displacement to create surface detail (the subdivision base cages are a cube and plane respectively).

- Columns and Zinkia are full scenes featuring significant variation in base-patch size.

3.4.1 Algorithm Comparisons

Figure 3.13 visualizes the quality and performance of all algorithms by coloring images according to the average area of micropolygons overlapping each pixel. Properly tessellated areas containing 0.5-pixel-area micropolygons are green, areas overtessellated by at least four times are red (poor performance) and undertessellated areas are blue (poor quality).

DiagSplit, NoSplit, and BinSplit each provide subpatch-parallelizable, crack-free tessellation solutions. Of these three algorithms, only DiagSplit consistently produces good tessellations. While DiagSplit meets the target area very well (most

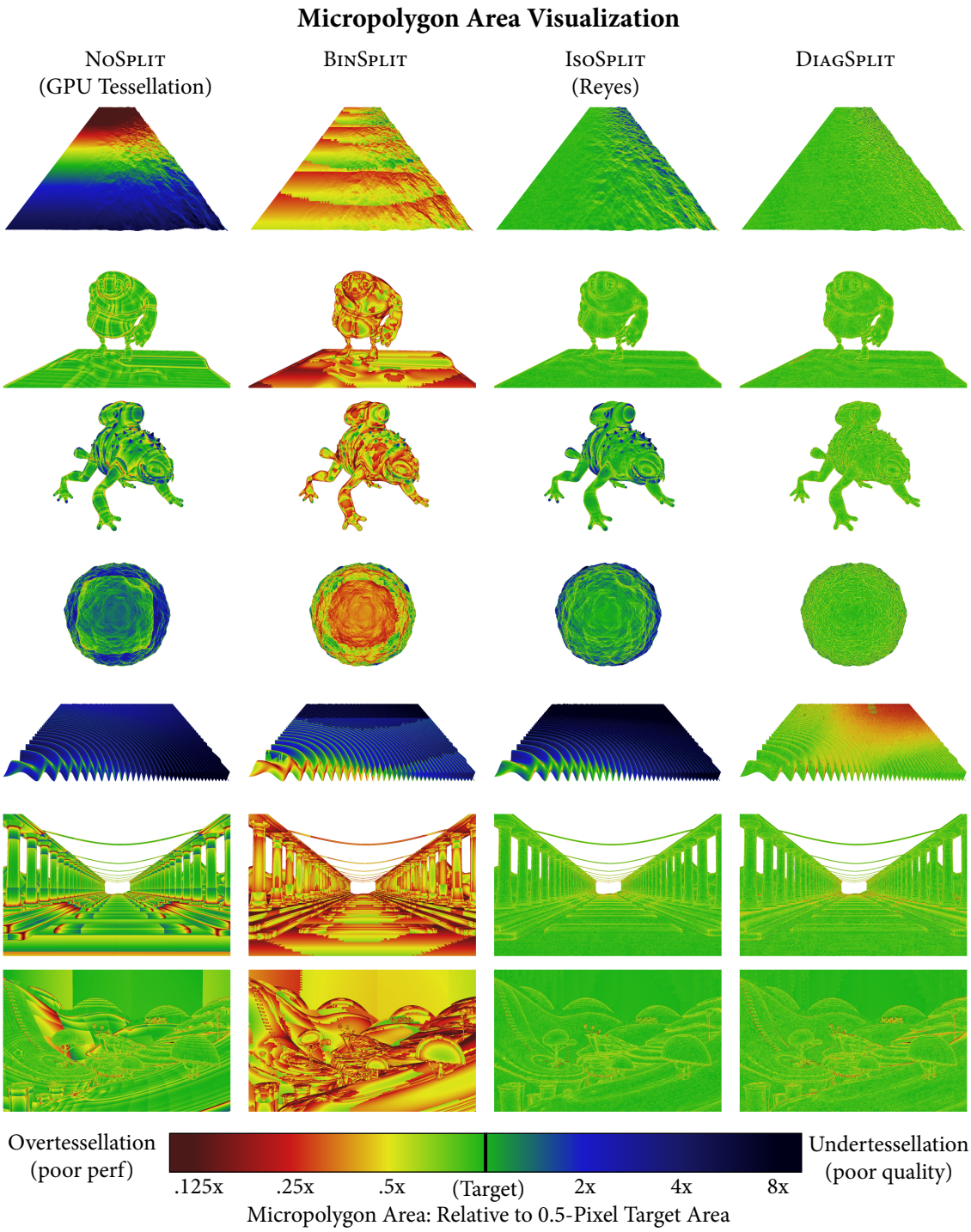


Figure 3.13: Visualization of triangle size: Green pixels are covered by micropolygons close to the 0.5-pixel-area target size. Red and blue pixels indicate overtessellation and undertessellation respectively.

regions of the DiagSplit images are green), NoSplit generates areas of overtessellation and undertessellation. The problem is acute in the pathological case of LongPlane, where NoSplit's uniform tessellation is too coarse near the viewer and too fine at a distance. Removing undertessellation by tessellating LongPlane conservatively (so that the near region of the plane becomes green) increases overtessellation afar and yields a mesh with seven times as many vertices as DiagSplit.

BinSplit is adaptive but it rounds dicing factors up to powers of two and lacks DiagSplit's interior-area scaling capabilities (unlike EdgeDice, uvDice cannot modify its interior tessellation independently from tessellation at subpatch edges). As a result of these two deficiencies, BinSplit overtessellates severely. In the character and full-scene tests, BinSplit generates between 2.2× (MonsterFrog) to 3.1× (Columns) more vertices than DiagSplit. With the exception of LongPlane, BinSplit's tessellations contain more micropolygons than NoSplit's. In addition, because BinSplit tessellation factors jump between powers of two, it produces tessellations that are less locally uniform than those of the other three algorithms. Modifying BinSplit to round edge factors to the nearest power of two (rather than rounding up) reduces overtessellation, but introduces large regions of undertessellation.

DiagSplit is designed for subpatch-parallel execution while IsoSplit is not. However, because IsoSplit does not rely on preserving agreement along edges to prevent cracks, it is able to make splits and set tessellation rates using information from subpatch interiors as well as edges. Further, IsoSplit does not need to determine subpatch tessellation factors until a decision to dice is made (it requires no edge constraints). Despite retaining more flexibility, we found that in tests where displacement is not present, both IsoSplit and DiagSplit tessellations approximate the desired micropolygon area well. In scenes without displacement, there is never more than a 7% difference in the total number and average area of micropolygons produced by DiagSplit and IsoSplit.

When displacement is present, DiagSplit produces better tessellations than IsoSplit because it accounts for the displaced position of the surface (not just the subdivision limit surface) in $\mathcal{T}$. The disparity in tessellation output is notable when displacement magnitudes are large. For example, IsoSplit (as well as NoSplit

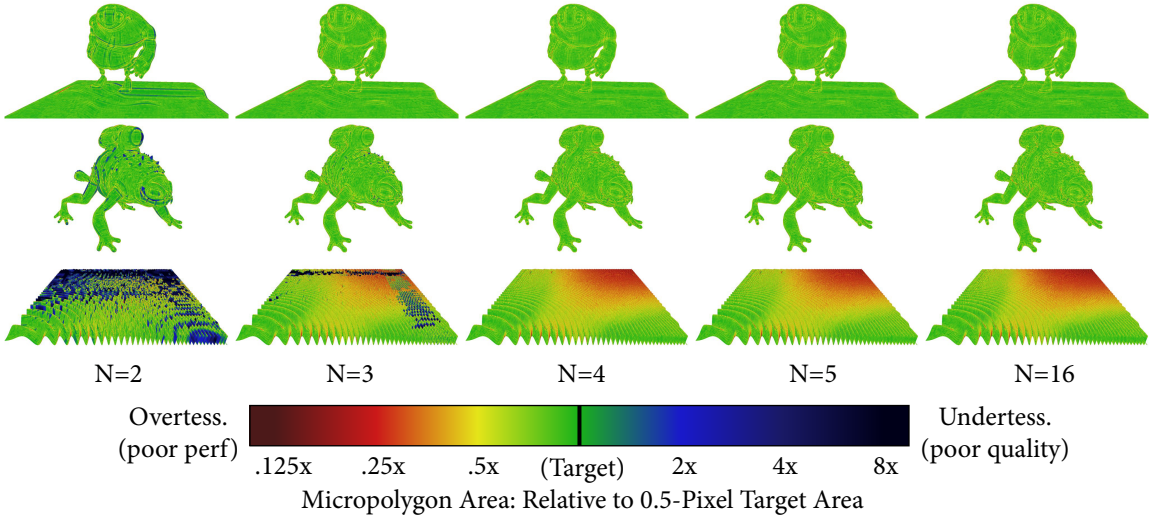


Figure 3.14: DIAGSPLIT tessellation quality improves as the number of surface samples used by $\mathcal{T}$ increases. Errors due to sparse edge sampling are particularly apparent in SINEWAVE, which contains a surface with high frequency detail. The DIAGSPLIT implementation evaluated in this section uses four samples to determine the tessellation factor for an edge (N=4).

and BINSPLIT) consistently undertessellates SINEWAVE because splitting decisions assume the surface is a flat plane. Similar undertessellation is visible on the right side of LONGPLANE, the spikes in MONSTERFROG, and near the silhouette of BUMPY. In offline rendering, manually increasing tessellation amounts for such objects is a reasonable solution to this problem. However, it is labor intensive, view-dependent, and causes overtessellation in areas of a surface without significant displacement. Accounting for displacement during tessellation requires no user input and yields a better approximation to the final surface. Modifying ISOSPLIT to account for displacement yields tessellations of approximately the same quality as DIAGSPLIT's. Accounting for displacement in NOSPLIT has less benefit because the scheme is incapable of adapting to the increased surface detail.

3.4.2 Edge Sampling

DiagSplit uniformly samples surface position along an edge to compute tessellation factors. Experiments show that four samples per edge (N=4) yields a good balance between cost to compute $\mathcal{T}$ and overall tessellation quality. Figure 3.14 visualizes tessellations that result from a range of edge sampling rates.

Even when surfaces contain no high frequency detail, approximating an edge as a line formed by its endpoints (N=2) results in poor quality at silhouettes. When an edge spans a surface silhouette, it is common for the projected position of its endpoints to fall nearly on top of each other. As a result $\mathcal{T}$ estimates edge length to be near zero and incorrectly computes a tessellation factor of one segment. This form of sampling error is troublesome because high-quality silhouettes are a major motivation for micropolygon rendering.

In practice we found that N=3 often suffices for smooth, undisplaced surfaces and that N=4 yields good results for most displacements. Of course, any point sampling of an edge is prone to aliasing. Artifacts due to aliasing are clearly present in the SineWave scene (surface detail becomes very high frequency in the top-right corner of the plane). However, with the exception of pathological cases such as SineWave, tests indicate that using values of N greater than five provide little benefit to tessellation quality and simply introduce higher $\mathcal{T}$ evaluation costs. We have not studied the impact of irregular or adaptive sampling techniques in $\mathcal{T}$; the advantage of adaptivity is largely achieved through splitting and it is important that the cost of $\mathcal{T}$ remain as small as possible to keep the overhead of producing DiagSplit tessellations low.

3.4.3 DiagSplit Characteristics

Table 3.2 quantifies DiagSplit tessellation characteristics for the seven example scenes. The table corroborates the results illustrated by Figure 3.13; on all scenes except SineWave, DiagSplit generates triangles that are, on average, within 26% of the 0.5 pixel target area. Micropolygons of this size do not approximate the high frequency detail of SineWave well, so DiagSplit tessellates more finely to yield a

DiagSplit Execution Summary

	LongPlane	BigGuy	MonsterFrog	Bumpy	SineWave	Columns	Zinkia
Scene Base Patches	1	2,570	1,292	6	1	13,044	151,651
MP Area (Avg)	.43	.42	.37	.39	.24	.38	.44
(Max)	2.56	1.10	3.33	2.63	9.75	1.32	1.17
Split Depth (Avg)	18.0	2.2	2.9	13.0	79.0	1.8	0.1
(Max)	18.0	6.0	11.0	13.0	79.0	10.0	16.0
Grids/Patch (Avg)	8,014.0	6.0	9.2	1,767.2	266,116.0	6.9	1.6
(Max)	8,014.0	32.0	88.0	2,416.0	266,116.0	629.0	4,552.0
Verts/Grid (Avg) (Max = 256)	159.3	141.0	122.7	149.2	52.7	93.5	69.8
Overhead Surf Evals (% of Total)	9%	11%	13%	11%	32%	19%	24%

Table 3.2: Execution statistics from tessellating scenes in Figure 3.13 using DiagSplit. DiagSplit tessellations contain triangles that on average are within 26% of the target micropolygon area (0.5 pixels).

close approximation to the surface (for efficient rendering, it would be preferable to pre-filter SineWave geometry).

Table 3.2 also provides insight into the overhead of adaptivity. The cost of determining where DiagSplit places tessellation vertices is dominated by point sampling surface position. These evaluations, which occur in $\mathcal{T}$ and to compute interior scale parameters prior to dicing, add to the fundamental cost of evaluating surface position and shading interpolants at final micropolygon vertex locations. When subpatches are large on screen, less than 13% of all surface evaluations constitute overhead of computing the split-dice tessellation. This fraction grows to 19% and 24% in Columns and Zinkia, which contain many small base patches (these scenes have the smallest number of vertices per diced subpatch). However, small base patches are rarely split (e.g., despite having several large patches, Zinkia's average split-tree depth is 0.1), so DiagSplit incurs essentially the same overhead to compute edge factors as NoSplit.

3.4.4 Prototype Parallel Implementation

Although DiagSplit is designed for integration into the GPU pipeline (Section 3.5 describes the DiagSplit tessellation pipeline architecture), we have developed a prototype parallel implementation for multi-core CPUs. This implementation leverages locality inherent in split-dice by processing split subpatches in depth-first order. It represents subpatches compactly as 52-byte records (four parametric coordinates, four tessellation factor constraints, and a pointer to the base patch). As a result, a 20-element stack requires less than 1 KB of storage (Table 3.2 indicates this is more than sufficient for most scenes). Depth-first implementation runs almost entirely out of local data caches and does not exhibit the large memory footprint or high-bandwidth limitations of breath-first tessellation [Eisenacher et al. 2009; Patney and Owens 2008].

A single core of a 3 GHz Intel Core i7 processor tessellates Approximate Catmull-Clark (ACC) surfaces [Loop and Schaefer 2008] into micropolygon meshes at a rate of 21 million triangle micropolygons per second (MPs/sec). This timing includes the cost

of converting the base mesh to ACC patches, as well as evaluation of surface normals, tangents, and texture coordinates for shading. About 70% of overall execution time is spent evaluating these surface attributes at final vertex positions (this code has been optimized using vector instructions). Performance drops to 11 million MPs/sec when evaluating displaced surfaces because the cost of filtering texture data in software is expensive. The implementation achieves parallel execution across multiple cores by placing all subpatches generated by splits into a work queue accessed by all processors. This simple design scales moderately well out to eight processors (6.2× for non-displaced surfaces, 6.7× for displaced ones). An eight core Intel Core i7 system tessellates ACC surfaces at a rate of 130 million MPs/sec (73 million MPs/sec when displaced). This high performance is achieved while generating high-quality meshes that do not contain excessive numbers of polygons.

3.5 Pipeline Integration

The subpatch-parallel properties of DiagSplit facilitate its integration into the micropolygon pipeline as an extension of the GPU pipeline's tessellation architecture. A four-stage DiagSplit tessellation pipeline is shown in Figure 3.15. The three-stage GPU tessellation pipeline is provided alongside for comparison. The DiagSplit pipeline adds a new stage to the GPU pipeline (labeled DiagSplit). This stage accepts as input a stream of base patches from the Patch Generation stage and outputs a stream of records representing diceable subpatches. Each subpatch record specifies the parametric coordinates of subpatch corners, edge tessellation factors, and a subpatch interior scaling factor. The remainder of the DiagSplit tessellation pipeline is essentially identical to that of the GPU pipeline. The fixed-function Uniform Tessellation stage generates an EdgeDice triangle grid from subpatch records and data-parallel Vertex Processing computes final vertex positions. In the DiagSplit pipeline, base patch data emitted by the Patch Generation is automatically passed down the pipeline and made available to all Vertex Processing shader program invocations for grids generated from this base patch. In the GPU pipeline, base patch data need only be provided to Domain shader invocations for a single grid.

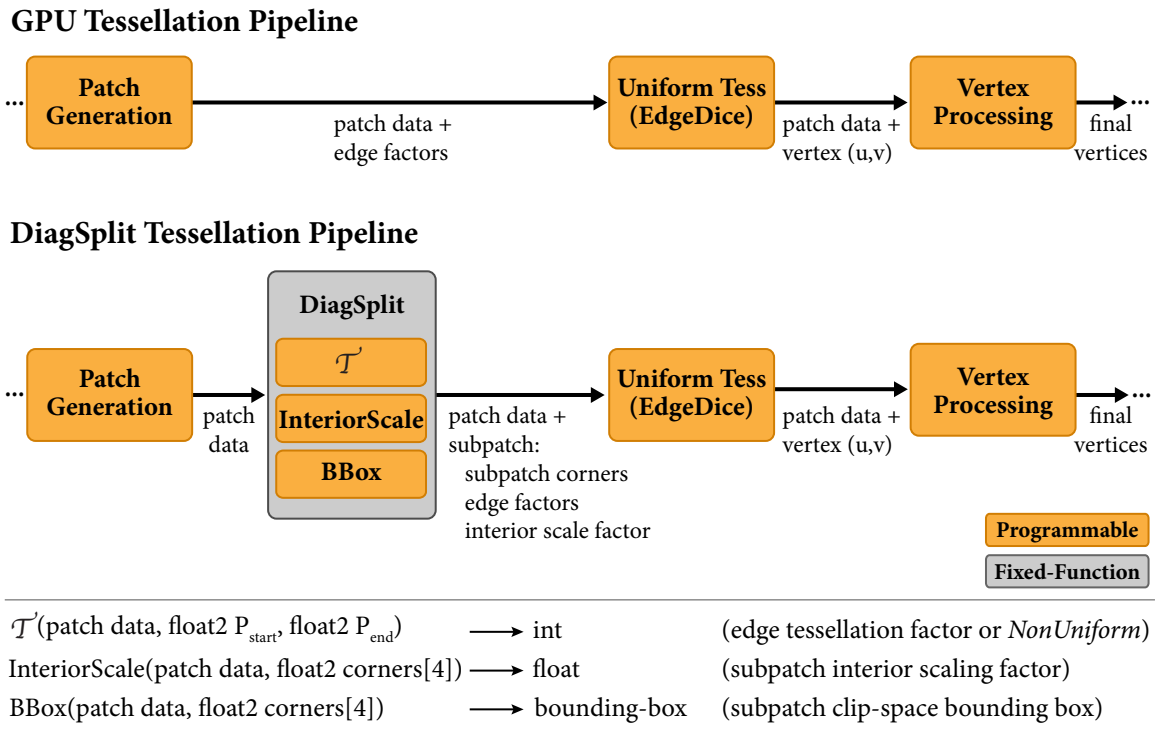


Figure 3.15: A DIAGSPLIT tessellation pipeline architecture: The DIAGSPLIT pipeline extends the GPU pipeline with a new stage (DiagSplit) that performs subpatch splits. This stage is not fully programmable, but operates in conjunction with application-provided, surface-specific functions that compute edge tessellation and interior scaling factors.

Recall the *DiagSplit* pseudocode in Figure 3.9 is not specific to any parametric surface type. Surface-specific components of the implementation involve the edge tessellation factor function $\mathcal{T}$ and evaluation of the surface for area scaling (not shown in pseudocode). Correspondingly, the DiagSplit stage is not fully programmable. That is, the application programmer is not responsible for implementing the pseudocode in Figure 3.9. Instead the application provides the pipeline several surface-specific functions that are used in conjunction with the pipeline-supplied DiagSplit implementation to produce diceable subpatches.

- $\mathcal{T}$**:** Computes the tessellation factor for an edge given patch data and the parametric location of edge endpoints. An example implementation of $\mathcal{T}$ was given in Figure 3.10.

- **InteriorScale:** (optional) Computes a subpatch interior-area scaling factor S from patch data and the parametric location of subpatch corners. If this function is not provided by the application, the interior scale factor for subpatches defaults to 1.

- **BBox:** (optional) Computes the clip-space bounding box of a subpatch from path data and the parametric location of subpatch corners. This function is not required by the DiagSplit algorithm, but provides surface semantics that enable many useful pipeline optimizations.

Constraining the programmability of the DiagSplit stage's architecture provides flexibility for pipeline implementations to be highly optimized. The space of viable implementations is large. In a micropolygon pipeline, the process of splitting base patches into subpatches plays a role similar to that of the rasterizer in a system optimized for large polygons. Split serves as a work generator, producing semi-regular pieces of work (subpatches) for consumption by the remainder of the pipeline. As is the case when rasterizing large triangles featuring significant variation in size, strategies for generating subpatches in parallel and for dynamically redistributing the resulting computations across many execution units (balancing the conflicting goals of

locality and parallelism) benefit from global knowledge of hardware characteristics and pipeline behavior. They are best left under system control.
Also, in addition to making splitting decisions that yield a high-quality mesh, implementations of the DiagSplit stage may include optimizations such as:

- Hierarchical frustum and occlusion culling. Given the spatial bound of a subpatch (computed by **BBox**) the pipeline can discard off-screen or occluded subpatches prior to dicing [Apodaca and Gritz 2000; Boulos et al. 2010]. Splitting provides the opportunity to perform these culling checks on progressively smaller surface regions, not just original base patches or individual micropolygons. This optimization is similar to hierarchical occlusion techniques used on aggregate scene geometry [Greene et al. 1993] or regions of large triangles [Morein 2000]. When rendering the COLUMNS and ZINKIA scenes, using conservative bounds to occlusion-cull subpatches prior to dicing reduces the number of Vertex Processing shader program invocations by 3.5 and 5.1 times.

- Eye-plane splits. Continuing to split a subpatch until its spatial bound lies entirely in front of the eye plane allows the pipeline rasterizer to avoid eye-plane clipping of individual micropolygons [Apodaca and Gritz 2000].

Rather than extending the existing pipeline architecture with a new stage as proposed above, one alternative way to integrate DIAGSPLIT in the GPU pipeline is to extend the existing Patch Generation stage to output an unlimited number of patch records for each input patch received. (Patch Generation currently emits one output patch record for each input.) With this capability, it is possible for the application to implement the entire DIAGSPLIT process as a Patch Generation stage shader program that outputs diceable subpatches. While the simplicity of this interface change is attractive, placing the entire splitting process under application control serializes the execution of splits for each base patch and precludes many of the global pipeline optimizations described above.

3.6 Discussion

DiagSplit adapts the Split-Dice algorithm to split subpatches in non-isoparametric directions and use a dicing scheme that supports different tessellation factors for each subpatch edge. With these modifications, DiagSplit retains the adaptivity and crack-free quality of advanced Reyes implementations but can also process individual subpatches in parallel. Adaptivity allows DiagSplit to generate tessellations containing fewer and more uniform micropolygons than existing parallel methods. This reduces the overall cost of tessellation and also leads to more efficient processing by downstream pipeline stages. Subpatch-parallelism enables implementation via extension of the GPU pipeline's tessellation architecture. As an extension of GPU tessellation, the DiagSplit tessellation pipeline assumes its benefits; it is versatile (programmable vertex evaluation) and also amenable to high-performance implementation (data-parallel vertex evaluation, special-purpose hardware for dicing). However, the DiagSplit pipeline also assumes the GPU pipeline's constraints (limitation to parametric surfaces, tedious coding to ensure a crack-free mesh). Given the rapid adoption of tessellation in current and near future games, these limitations appear to be acceptable for real-time graphics.

When considered in the context of current GPU pipeline implementations, the overhead of processing non-isoparametric subpatches, compared to iso-parametric subpatches, is small. While non-isoparametric subpatches make surface evaluation using efficient forward-differencing schemes more difficult, the GPU tessellation pipeline already precludes use of these methods in favor of a simple per-vertex programming model and brute-force, data-parallel evaluation. Also, the preceding evaluation showed it is beneficial to account for surface displacement when making splitting decisions and setting edge tessellation factors. Although it is more costly to use analytic techniques to make these decisions for non-isoparametric subpatches, direct point sampling methods are unaffected by non-isoparametric properties of subpatches and can easily account for the displaced position of the surface.

Although crack prevention requires DiagSplit to adhere to tessellation properties determined along subpatch edges to prevent cracks, it need not make splitting

decisions based only on edges. For example, the DiagSplit implementation evaluated in Section 3.4 performs additional subpatch splits to ensure that diced grids are limited in vertex count. (These splits, which are omitted from the pseudocode in Figure 3.9, are not required by edges.) DiagSplit's splitting rules and edge constraints ensure these additional splits do not introduce cracks. Further extensions of DiagSplit could split subpatches if greater surface detail is needed within subpatch interior regions, or choose different non-isoparametric partitions to create subpatches with better screen aspect ratio.

Chapter 4

Micropolygon Rasterization

Micropolygons place new demands on a GPU rasterizer. The rasterizer in a micropolygon pipeline must excel at processing large numbers of polygons that rarely overlap more than a few pixels on screen. In contrast, current GPU rasterizers are tuned to process polygons that cover tens of pixels; they are not designed to be efficient for micropolygon workloads. In pursuit of a rasterizer better optimized for micropolygons, this chapter analyzes the costs of computing micropolygon-screen coverage. It shows that for micropolygons (specifically those generated by DiagSplit tessellation), it is preferable to use a simpler rasterization algorithm to reduce computational cost, and that micropolygon-parallel, rather than sample-parallel, execution is a more effective way to scale rasterizer throughput.

The analysis provided in this chapter (initially presented in [Fatahalian et al. 2009]) focuses on the algorithmic efficiency of micropolygon rasterization and the characteristics of its mapping to data-parallel execution. It does not attempt to design and evaluate the performance of a specific, highly optimized hardware or software implementation. The design of a custom ASIC that implements the micropolygon rasterization algorithm described in this chapter is explored by Brunhaver et al. [2010]. For completeness, a summary of Brunhaver et al.'s findings is provided in Section 4.5.

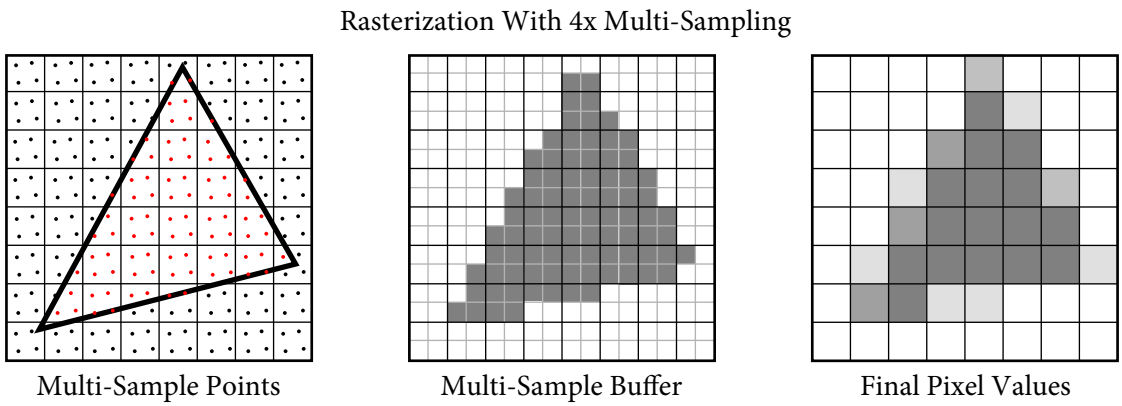


Figure 4.1: Computing a triangle's screen coverage using 4× multi-sampling: Left: Multi-sample points on a pixel grid. Covered multi-sample points are red. Center: The triangle contributes to the multi-sample buffer at all covered multi-sample points. Right: Filtering the multi-sample buffer's contents results in anti-aliased triangle edges in the final image.

4.1 GPU Rasterization

The GPU pipeline's rasterization stage is responsible for computing a polygon's coverage of screen pixels and for generating fragments for subsequent pipeline processing. A region of a pixel is said to be covered by a polygon if it falls within the polygon's 2D screen projection. GPU rasterizers approximate coverage discretely by sampling it multiple times per pixel. Figure 4.1-left shows a pixel grid and the position of four sample points, referred to as *multi-sample points*, in each pixel. Covered multi-sample points are highlighted in red. Sampling coverage four times per pixel (4× multi-sampling) is common in current games, however, to more accurately estimate partial-pixel coverage (that is, to reduce aliasing at polygon edges), high-end GPUs support rendering modes with up to 32 multi-sample points per pixel [NVI 2010b].

For clarity, Figure 4.1 also illustrates how the results of rasterization are used to make an image. The GPU pipeline stores one color value per multi-sample point (the multi-sample buffer: Figure 4.1-center). If a polygon covers a multi-sample point, its color (as determined by shading; see Chapter 6) is potentially blended with the corresponding frame-buffer value (alpha, stencil, or depth tests performed by the pipeline after rasterization can prevent frame-buffer update). When rendering of the

frame is complete, the pipeline filters the multi-sample buffer's contents to produce final pixel values (Figure 4.1-right).

A rasterizer must identify all multi-sample points covered by a polygon. Regardless of whether rasterization is implemented using fixed-function hardware or optimized software [Abrash 2009], finding these points involves three key steps: performing per-polygon preprocessing ("polygon setup"), quickly establishing a set of possibly-covered multi-sample points, and performing individual point-in-polygon tests.

Polygon setup encapsulates computations, such as clipping, back-face culling, and computing edge equations, that are performed once per polygon. Some setup operations (e.g., computing edge equations) decrease the cost of subsequent point-in-polygon tests. When polygons are large, the cost of setup is amortized over many tests, so it need not be widely parallelized. In fact, high-end ATI GPUs [Adv 2010] as well as NVIDIA GPUs prior to the GF100 ("Fermi") architecture serialize polygon setup (high end GF100 generation GPUs can set up four triangles in parallel) [NVI 2010b].

Next, rasterizers compute polygon overlap with coarse screen tiles [Fuchs et al. 1989; McCormack and McNamara 2000] or use hierarchical techniques [Greene 1996; McCool et al. 2001; Seiler et al. 2008] that utilize multiple tile sizes. Coarse overlap tests allow the rasterizer to quickly classify large regions of the screen as completely inside (Figure 4.2-orange 2×2 pixel tiles) or outside (white 2×2 pixel tiles) a polygon without performing individual point-in-polygon tests. Avoiding tests in regions falling outside the polygon is important because the polygon does not contribute to the image at these multi-sample points (performing many point-in-polygon tests to determine this result is wasteful). A rasterization scheme's *sample-test efficiency* (STE), the percentage of point-in-polygon tests that identify covered multi-sample points, is a measure of this waste.

Sample-Test Efficiency (STE): the percentage of point-in-polygon tests that identify covered multi-sample points.

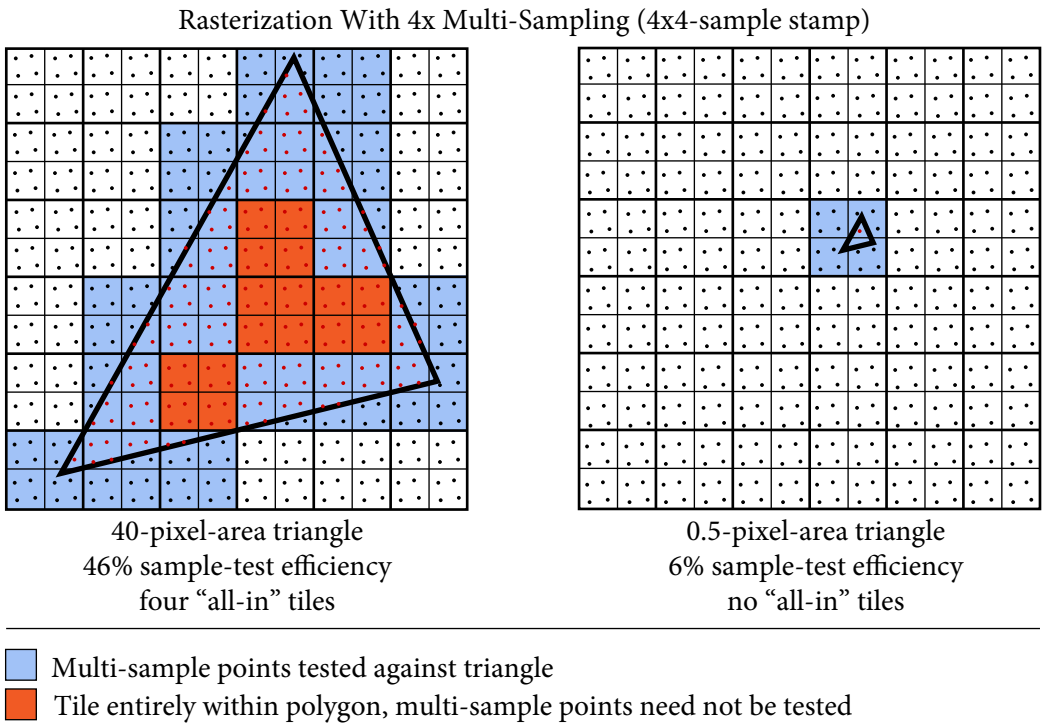


Figure 4.2: Rasterization of two triangles onto a 4× multi-sampled frame buffer using a 4×4 multi-sample stamp. Multi-sample points in regions highlighted blue undergo coverage tests. Orange regions fall entirely within the triangle, so point-in-polygon tests need not be performed in these areas. The 0.5-pixel-area triangle is small compared to the size of the 4×4 multi-sample stamp, resulting in low STE.

Last, rasterizers perform point-in-polygon tests to determine coverage of individual multi-sample points. Point-in-polygon tests are executed efficiently by testing contiguous blocks of multi-sample points ("stamps") against a single polygon in parallel. This approach was introduced in Pixel Planes [Fuchs et al. 1985] which tested all image multi-sample points against a polygon simultaneously. Other rasterizers use tile sizes ranging from 4×4 to 128×128 multi-samples [Pineda 1988; Fuchs et al. 1989; Seiler et al. 2008]. Modern GPU rasterizers simultaneously test 64 or more multi-sample points against a single polygon using data-parallel units [Houston 2008; NVI 2010b].

In summary, modern rasterizers rely on per-polygon preprocessing, bulk acceptance (or rejection) of large tiles of candidate multi-sample points, and wide data-parallel coverage tests to achieve high throughput. Unfortunately, these optimizations are ineffective when polygons shrink to subpixel sizes. First, under micropolygon workloads, the frequency of setup operations increases dramatically and its cost is no longer amortized over many point-in-polygon tests. Setup must be minimized or parallelized when possible. Second, it is unnecessary to use hierarchical schemes to computing candidate sets of multi-sample points. A micropolygon's screen bounding box describes a tight candidate set that contains only a few uncovered multi-sample points. Last, large stamp sizes are inefficient because the screen area covered by a stamp is significantly larger than a micropolygon. The inefficiency of a 4×4-multi-sample stamp (small by modern GPU standards) is illustrated in Figure 4.2. In the figure, screen regions containing multi-sample points tested against the two triangles are blue. STE is 46% for the 40-pixel-area triangle at left, but drops to 6% for the 0.5 pixel-area-micropolygon at right (rasterizing the large triangle also benefits from several completely covered tiles). Large raster stamps trade off extra processing near polygon edges in exchange for efficient data-parallel execution. When rendering micropolygons, all candidate multi-sample points are near a polygon edge.

MPRAST

	`for each MP:`	`// (parallel)`
Setup	`Back-face cull`	`// 24 ops`
Bound	`Compute MP bbox`	`// 27 ops`
Test	`for each sample in bbox:`	
	`  Compute sample XY`	`// 29 ops`
	`  Test MP-sample coverage`	`// 24 ops`
Process Hit	`  If sample covered`	
	`  Update current fragment`	

Figure 4.3: The MPRAST micropolygon rasterization algorithm relies on parallel execution over many micropolygons (parallelism over the outermost loop). Data-parallel operation counts (which include data-movement and mask operations in addition to mathematical operations) correspond to the MPRAST implementation evaluated in Section 4.4.

4.2 The MPRAST Algorithm

Pseudocode for a simple, but efficient, micropolygon rasterization algorithm (MPRAST) is given in Figure 4.3. This algorithm omits many common optimizations found in GPU rasterizers (see Section 4.1). For example, coarse or hierarchical methods are not used to accept or reject blocks of multi-sample points. Instead, MPRAST simply computes a tight axis-aligned bounding box for each micropolygon and tests all multi-sample points within this bound.

The setup phase of MPRAST is minimal. In setup, each micropolygon processed by MPRAST undergoes a sidedness check (to support front or back-face culling). At low multi-sampling rates, it is advantageous to avoid the overhead of explicitly precomputing and storing micropolygon edge equations during setup. Instead, MPRAST conducts each point-in-polygon test by translating micropolygon vertices into a 2D coordinate system with the multi-sample point at the origin. These point-in-polygon tests are more expensive than tests that use explicit edge equations (each requires three additional ops). Therefore, edge precomputation in setup remains beneficial when rasterizing larger micropolygons or at high multi-sampling rates. A micropolygon's bounding box must contain more than seven multi-sample points to overcome the overhead of precomputation.

The bound phase of MPRAST computes an axis-aligned bounding box for the micropolygon. This box is clamped to sub-pixel, not pixel, boundaries to minimize the number of point-in-polygon tests performed. MPRAST multi-sample points are generated using stratified sampling [Cook 1986], so bounding boxes are clamped to strata boundaries (see [Cook et al. 1990] for a clever implementation of generating stratified sample positions from a small lookup table).

The test phase of MPRAST iterates over all multi-sample points in the bounding box, performing point-in-polygon tests. The size of a bounding box and, correspondingly, the STE of MPRAST, is sensitive to the area, aspect ratio, and orientation of micropolygons. To maximize STE, it is important for pipeline tessellation to generate "good" micropolygons that are similar in these properties. In general, quadrilateral micropolygons fill the area of their bounding box better than triangles. The regular interior region of grids produced by EDGEDICE features pairs of adjacent triangles that can easily be treated at quadrilaterals (recall Figure 3.4-right). Instead of processing each triangle independently, one MPRAST optimization is to treat pairs of triangles together, performing two point-in-triangle operations using five edge tests. This optimization halves the number of bounding boxes computed by MPRAST, since only each pair of triangles needs to be bounded.

When a micropolygon covers a multi-sample point, fragments must be generated for subsequent shading and frame-buffer processing (Process Hit). The requirements for generating fragments depend on whether fragment shading is enabled in the pipeline. When fragment shading is not enabled, for example, when generating shadow maps or if shading is computed prior to rasterization at mesh vertices, the rasterizer need only interpolate micropolygon depth and (potentially) color at the covered multi-sample point. When fragment shading is enabled, the rasterizer's task is more complex. For example, it must compute interpolation equations for vertex attributes accessed during shading. The problem of computing micropolygon-multi-sample coverage is challenging regardless of the shading method used by the pipeline. The remainder of this chapter focuses only on the rasterizer's responsibilities for computing coverage. It does not consider the costs of generating inputs for shading. Shading in a micropolygon pipeline is discussed in detail in Chapter 6.

4.3 Parallelizing MPRAST

Consider a rasterizer featuring 64 execution units for performing parallel point-in-polygon tests. For a typical subpixel-area micropolygon, the bounding box computed by MPRAST will not contain enough multi-samples to utilize all execution units. Moreover, even if the bounding box were to contain more than 64 multi-sample points, sample testing work would not divide evenly onto the units, resulting in low utilization. Testing many multi-sample points against a single micropolygon in parallel would be a poor way to scale MPRAST throughput. Instead, scaling MPRAST requires parallelization across many micropolygons.

MPRAST carries this strategy to an extreme. Given N execution units, it tests N unique micropolygons against N unique multi-sample points in parallel. In the context of the MPRAST pseudocode in Figure 4.3, this is parallelism over the outermost loop. There is no shortage of micropolygons reaching the rasterizer, so this approach scales to large numbers of execution units without incurring the overhead of large multi-sample stamps. However, micropolygon-parallel scaling does introduce new implementation costs that stand to mitigate the algorithmic efficiency of MPRAST.

For example, micropolygon-parallel execution is less amenable to sharing control and data across the rasterizer's execution units. In a stamp-based rasterizer, this sharing is trivial. All execution units access the same triangle data, and only a single iterator is needed to advance the rasterizer from stamp to stamp. In MPRAST, state and control of iteration over multi-sample points must be handled separately for each micropolygon. Further, micropolygon-parallel rasterization is susceptible to load imbalance across execution units because the number of multi-sample points contained in each micropolygon's bounding box varies. If parallel rasterization of many micropolygons proceeds in lockstep, execution-unit utilization depends heavily on micropolygons having bounding boxes of similar size (tessellation must produce micropolygons of approximately the same size, orientation, and aspect ratio).

Parallelization across micropolygons also increases footprint (data for many micropolygons must be maintained by the rasterizer), requires flexible access to multi-sample position data (unlike stamp-based rasterization, there is no longer a fixed

mapping between multi-sample points and execution units), and identifies covered multi-sample points out of micropolygon arrival order. Fortunately, although the GPU pipeline defines strict rules for ordering fragments from different input primitives, it does not specify an ordering for fragments from micropolygons generated from the same input base patch. Pipeline implementations are free to rasterize micropolygons from the same base patch in any desired order.

The impact of these costs on the performance and efficiency of MPRAST depends heavily on the details of a particular parallel implementation. For example, managing simultaneous iteration over many bounding boxes introduces instruction overhead to a vectorized software implementation. In contrast, its cost to an ASIC implementation of MPRAST manifests as additional chip area and power.

4.4 Evaluation

This section studies the behavior of a parallel implementation of MPRAST in two key areas. First, it compares the sample-test efficiency of MPRAST against that of stamp-based approaches. Second, it measures the impact of variance in micropolygon bounding-box size on the utilization of data-parallel execution units.

We implemented MPRAST using a library of fixed-width, data-parallel operations. Four implementations utilizing 8-, 16-, 32-, and 64-wide operations were created. In addition to standard floating-point and integer operations, the library supports data gathers/scatters and the ability to manipulate vector masks. Data-parallel operation counts for the Setup, Bound, and Test phases of MPRAST are given in Figure 4.3. These counts correspond to an implementation that does not precompute edge equations in Setup and processes triangle micropolygon individually (no triangle-pairs optimization).

Parallel execution occurs exactly as described in Section 4.3: all rasterizer execution units operate in lockstep, rasterizing different micropolygons in parallel. Processing for all active micropolygons must complete before a subsequent batch of micropolygons is accepted by the rasterizer. Our implementations make no attempt

to dynamically correct for load imbalance caused by variance in the amount of per-micropolygon work. Thus, utilization reported here should be considered a low watermark for future optimized implementations.

We conducted experiments using micropolygons generated by DiagSplit for a large collection of test scenes. Subpatch-granularity occlusion-culling is enabled in the rendering pipeline prior to rasterization (as described in Section 3.5), so a large fraction of micropolygons that reach the rasterizer are visible to the camera and front facing (very few micropolygons fail the sidedness check in the rasterizer).

4.4.1 Sample-Test Efficiency

Figure 4.4 quantifies the STE benefit of avoiding large multi-sample stamps when rasterizing micropolygons. Each graph plots rasterizer STE for various multi-sampling rates (1, 4, and 16 multi-sample points per pixel) given a stream of micropolygon inputs of the specified size (the four graphs correspond to 0.25, 0.5, 1.0, and 2.0-pixel-area triangle micropolygons). Recall from Chapter 3 that a DiagSplit tessellation containing 0.5-pixel-area triangles features about one mesh vertex per pixel.

MPRAST employs a raster stamp containing one multi-sample point (1×1 stamp, red bars). When multi-sampling is disabled, the screen area of this "stamp" is already larger than a 0.5-pixel-area micropolygon. Testing larger multi-sample stamps against a single micropolygon at once quickly results in low STE, as illustrated by the gold bars in the figure. For example, consider the case of using an 8×4 multi-sample stamp to rasterize 0.5-pixel-area micropolygons to a 4× multi-sampled frame buffer (this data point is interesting because 4× multi-sampling is commonly used in games and each rasterizer unit in an NVIDIA GTX 480 GPU tests an eight-pixel stamp of samples against a polygon per clock [NVI 2010b]). The 8×4 multi-sample stamp results in 3% STE. In contrast, MPRAST's single multi-sample stamp yields 19% STE under this configuration (on average, a micropolygon bounding box overlaps approximately 11 multi-sample strata). MPRAST performs over six times fewer point-in-polygon tests. Across all conditions, the STE difference between MPRAST and

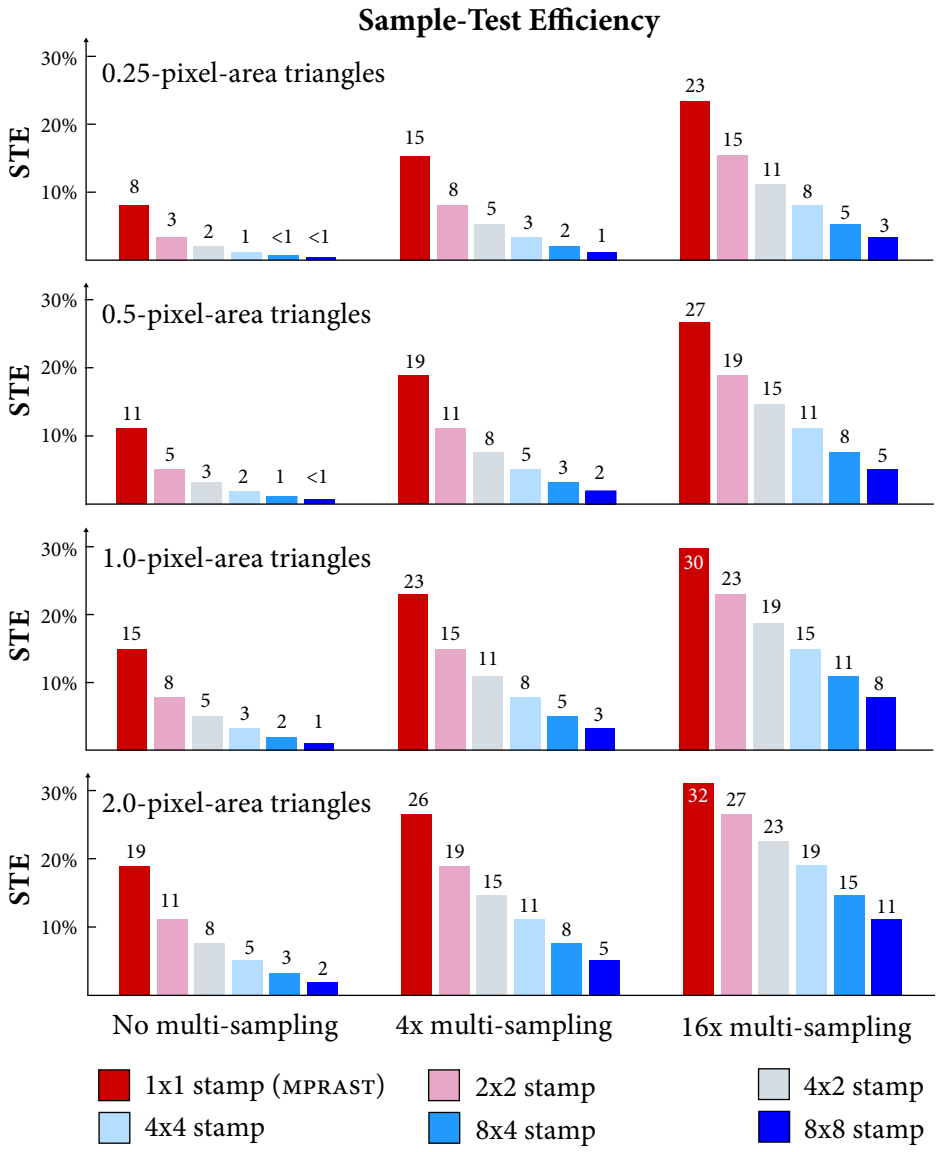


Figure 4.4: Under the load of micropolygons MPRAST (red bars) realizes higher STE than stamp-based rasterization. The benefit of MPRAST is greatest when micropolygons are small relative to the density of multi-sample points (small micropolygons, low multi-sampling).

the 8×4 multi-sample stamp rasterizer ranges from 2.2 times (for 2-pixel-area triangles at 16× multi-sampling) to 12.6 times (0.25-pixel-area triangles with no multi-sampling). When compared to the 8×8 multi-sample stamp under conditions of no multi-sampling, MPRAST performs up to 23 times fewer point-in-polygon tests.

Even though MPRAST achieves higher STE than stamp-based approaches, STE when rasterizing micropolygons is fundamentally lower than when polygons are large. At multi-sampling rates feasible for real-time rendering, the size of multi-sample strata is large relative to micropolygon area. Even during 16× multi-sampling, MPRAST's STE for 0.5-pixel-area micropolygons is only 27%. Shrinking micropolygons size to 0.25 pixels further reduces STE to 23%.

4.4.2 Utilization and Cost

Table 4.1 breaks down the cost of the *Setup*, *Bound*, and *Test* phases of MPRAST during processing of 0.5-pixel-area micropolygons. Results for implementations using 8, 16, 32, and 64 data-parallel execution units are given. Average execution-unit utilization by each phase of MPRAST appears in parenthesis. *Setup* utilization falls short of 100% because our implementation processes in parallel only micropolygons from the same DiagSplit grid. The number of micropolygons in a grid is rarely an exact multiple of the number of rasterizer execution units. *Bound* achieves nearly the same utilization as *Setup* because most grids reaching the rasterizer contain all front-facing micropolygons.

Even at low sampling rates, over 84% of all MPRAST operations constitute part of *Test*. Fortunately, DiagSplit produces high-quality tessellations, so variance in the number of multi-sample points tested per micropolygon is low and *Test* maintains high utilization of many data-parallel execution units. In the common case of 4× multi-sampling, *Test* sustains approximately 81% utilization of eight data-parallel execution units. Scaling the implementation eight-fold to 64 units drops utilization only to 63%. MPRAST implementations that dynamically rebalance or pack sample testing work for data-parallel execution can only achieve, at most, a modest 1.6× improvement in utilization. This result suggests micropolygon-parallel rasterization

	Data-Parallel Execution Units			
	8	16	32	64
NO MULTI-SAMPLING				
Setup	.07 (.99)	.07 (.98)	.06 (.96)	.06 (.92)
Bound	.18 (.98)	.08 (.97)	.08 (.94)	.07 (.90)
Test	.84 (.78)	.85 (.72)	.86 (.66)	.87 (.60)
Overall Util	.81	.76	.70	.64
Par Ops/MP	42.40	22.78	12.27	6.69
4× MULTI-SAMPLING				
Setup	.03 (.99)	.03 (.98)	.03 (.96)	.03 (.92)
Bound	.04 (.98)	.04 (.97)	.04 (.94)	.04 (.90)
Test	.92 (.81)	.93 (.75)	.93 (.70)	.93 (.63)
Overall Util	.82	.77	.71	.65
Par Ops/MP	87.02	46.50	25.10	13.78
16× MULTI-SAMPLING				
Setup	.01 (.99)	.01 (.98)	.01 (.96)	.01 (.92)
Bound	.02 (.98)	.02 (.97)	.02 (.94)	.01 (.90)
Test	.97 (.85)	.97 (.80)	.97 (.74)	.97 (.68)
Overall Util	.85	.80	.75	.68
Par Ops/MP	225.33	119.61	64.39	35.23

Table 4.1: MPRAST execution breakdown for 0.5-pixel-area triangle micropolygons. Table cells report the fraction of MPRAST execution time spent in the *Test*, *Bound*, and *Setup* phases of the algorithm. Execution-unit utilization for these stages is given in parenthesis. MPRAST execution time is dominated by performing point-in-micropolygon tests (*Test*). Even though micropolygon bounding boxes vary in size, execution-unit utilization in *Test* remains above 60%.

is not limited by variance in the amount of work per micropolygon. Maintaining state for many micropolygons, or the added complexity of generating fragments from many micropolygons at once, are more likely to limit the scalability of optimized MPRAST implementations.

The average cost of rasterizing a micropolygon using MPRAST is also given in Table 4.1. Operation counts refer to data-parallel operations, so wide data-parallel processing yields lower per-micropolygon operation counts. Although this implementation of MPRAST can benefit from further optimization (for example, Roca et al. [2010] suggest MPRAST optimizations for increasing STE and reducing the cost of point-in-micropolygon tests), micropolygon rasterization simply has high computational cost. There are no trivial "all-in" cases that accelerate large blocks of tests and, when tests are performed, STE is lower than that for large triangles. As a result, MPRAST performs a lot of multi-sample testing work. For example, during 4× multi-sampled rendering, a 32-wide version of MPRAST performs 25 data-parallel operations per micropolygon. Sustaining a rasterization rate of a billion micropolygons per second (approximately 16 million micropolygons at 60 Hz) requires processing throughput of 25 billion 32-wide operations per second, roughly equivalent to the entire programmable compute capability of a high-end NVIDIA GTX 480 GPU.

Recently, other researchers have attempted to rasterize micropolygons in software using compute-mode programming on GPUs. Consistent with our results, these implementations either do not achieve real-time performance [Zhou et al. 2009] or consume significant fractions of GPU compute resources to rasterize very simple scenes [Eisenacher and Loop 2010]. The high cost and the brute-force nature of MPRAST strongly suggest that an efficient implementation of the micropolygon rendering pipeline should feature fixed-function hardware for rasterization.

4.5 Fixed-Function Implementation

Motivated by the analysis of MPRAST in the previous section, Brunhaver et al. [2010] have conducted an initial exploration of the design of custom hardware for micropolygon rasterization. The results of this exploration are summarized briefly here. The

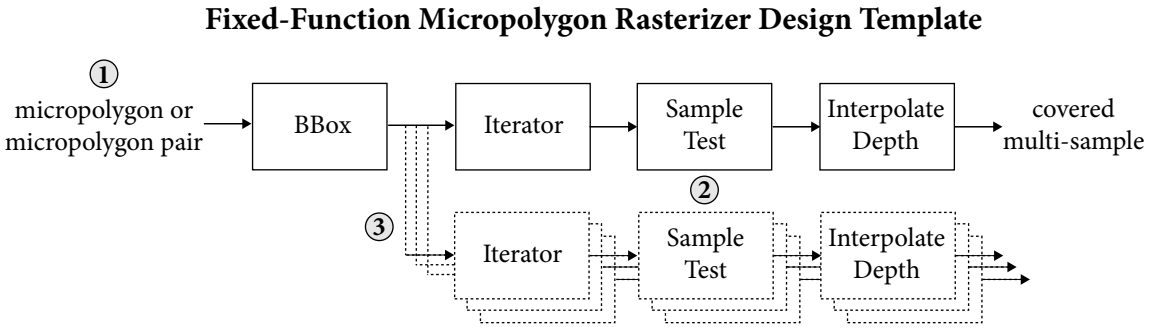


Figure 4.5: Basic structure and key design parameters for a fixed-function micropolygon rasterization unit that implements a variant of MPRAST. The unit shown above achieves limited parallelism through pipelined execution and optional use of small multi-sample stamps. To achieve high throughput, the unit is replicated to allow for processing multiple micropolygons in parallel (not shown).

reader is referred to Brunhaver et al. [2010] for complete details of this work.

Figure 4.5 illustrates the basic structure of Brunhaver et al.'s pipelined micropolygon rasterization unit, which closely follows the MPRAST algorithm. The unit takes as input a single micropolygon (or a pair of edge-adjacent micropolygons) with vertex positions represented as fixed-point values. It computes an axis-aligned bounding box for the micropolygon (BBox), and then tests the micropolygon's coverage of all multi-sample points within this bound. In the figure, the Iterator block generates micropolygon-sample pairs and the Sample Test block performs point-in-micropolygon tests.

Brunhaver et al. use the design in Figure 4.5 as a template for exploring optimizations that increase the area and power efficiency of their rasterizer. First, they implement the triangle-pairs optimization described in Section 4.2. This optimization increases hardware complexity: BBox must compute a bound over four vertices (rather than three) and Sample Test must perform five edge tests (rather than three).

Even so, experiments show synthesized designs that employ the triangle-pairs optimization achieve higher area and power efficiency that design that do not.

Second, they exploit the small size of micropolygons to reduce the precision of operations in Sample Test. Although not guaranteed by DiagSplit, the bounding box of nearly all micropolygons reaching the rasterizer spans fewer than eight pixels in width or height. By representing vertex positions using 12-bit fixed-point values, the cost (in terms of both energy and area per operation) of performing point-in-polygon tests is reduced by nearly a factor of two. (4.8 fixed-point format is used to provide eight bits of subpixel precision as required by the Direct3D architecture.) This design requires a separate, low-performance data path for rare cases where micropolygons do not meet this assumption.

Third, Brunhaver et al. depart from the MPRAST algorithm described in this chapter and explore the use of small multi-sample stamps (hardware blocks for processing multiple multi-sample points in parallel are indicated by the dotted boxes in Figure 4.5). During 4×-multi-sampled rendering, their results indicate the most efficient designs employ a stamp size of one multi-sample. However when rendering uses 16× multi-sampling, Brunhaver et al.'s results show that the low cost of a reduced-precision Sample Test unit makes a 2×2-multi-sample stamp a more efficient hardware solution.

Overall, Brunhaver et al. estimate that a hardware rasterizer formed by aggregating many of the units shown in Figure 4.5 (leveraging wide micropolygon-parallel execution to increase throughput) could process billions of micropolygon per second and consume only a small fraction of the area and power budget of a current GPU.

4.6 Discussion

This chapter advocates micropolygon-parallel execution to achieve high rasterizer throughput. By eschewing the large multi-sample stamps used by current GPU rasterizers, MPRAST can reduce the number of multi-sample tests needed to rasterize a scene substantially—up to 23 times in our experiments. The nearly uniform size

of micropolygons and the relaxation of GPU pipeline ordering semantics for micropolygons generated from the same surface patch facilitates efficient implementation of micropolygon-parallel rasterization.

Although MPRAST, as presented here, leverages only parallelism across micropolygons to achieve performance scaling, micropolygon-parallel and multi-sample-parallel execution are not mutually exclusive. Implementations striving for additional throughput or better performance on small, but not necessarily sub-pixel, polygons may choose to parallelize widely across polygons and then exploit small multi-sample stamps for further scaling. In Section 4.5 I described how this strategy was used by Brunhaver et al. [2010] to increase rasterizer efficiency. It is also adopted by NVIDIA's GF100 architecture [NVI 2010b], which anticipates smaller polygons produced by GPU tessellation by increasing rasterizer throughput by a factor of four over previous NVIDIA architectures. GF100 does so not by increasing raster stamp size, but by providing four rasterizers (each using large, eight-pixel stamps) that work on different polygons in parallel.

Motivated by the relatively low cost of rasterization for large-triangle workloads, Seiler et al. [2008] have recently questioned the merits of dedicated GPU hardware support for rasterization. However, despite increases in efficiency, MPRAST still entails extremely high cost. Its implementation, as shown by Brunhaver et al. [2010], benefits greatly from fixed-function acceleration. In light of these results, although micropolygons motivate reoptimization of GPU rasterizer designs, it seems likely that future designs will continue to utilize efficient, fixed-function processing.

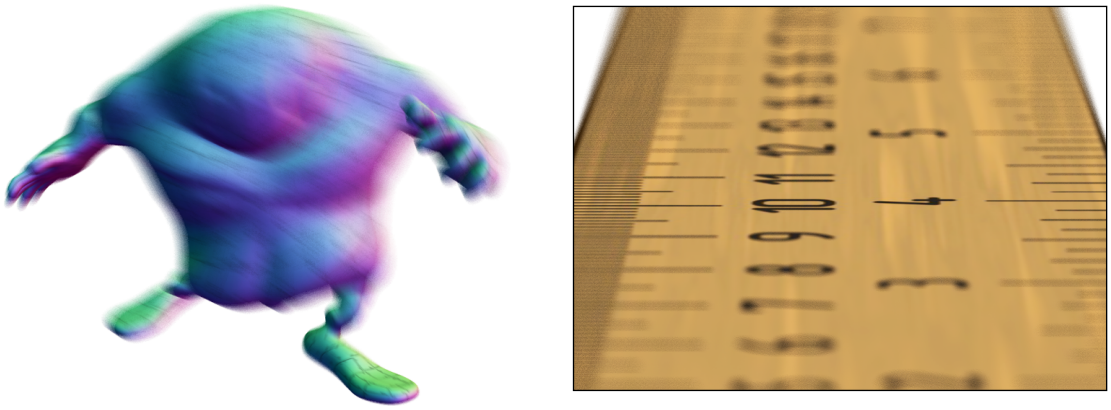
Chapter 5

Rasterization With Motion and Defocus Blur

The MPRAST algorithm from the previous chapter computes a micropolygon's screen coverage at a single instant in time. It generates images, like an ideal pinhole camera, where all scene objects are perfectly focused. However, real cameras do not always produce perfectly sharp pictures. Scene objects move over the duration of a frame's exposure, causing blur in the final image (motion blur, Figure 5.1-left). Objects positioned far from the camera's focal plane are also blurred in photographs, due to defocus (defocus blur, Figure 5.1-right).

Simulating motion blur and defocus blur accurately is a key feature of high-quality offline rendering systems. Camera focus (or a lack thereof) is manipulated by artists to achieve visual style and guide viewer attention to critical parts of a scene [Arijon 1991]. Motion blur is important for producing realistic-looking animations, especially when scene objects undergo rapid motion. When rendered without motion blur, object movement appears choppy (the result of temporal aliasing). Motion-blurred animation appears realistic and smooth.

This chapter considers the problem of augmenting a micropolygon rasterizer to simulate motion blur and defocus blur. GPU rasterization must already undergo change to improve micropolygon rendering performance (the topic of Chapter 4). The additional rasterizer changes explored here improve rendering quality. To generate



Motion Blur Defocus Blur

Figure 5.1: Images rendered with motion blur (left) and defocus blur (right).

motion blur and defocus blur effects, this work follows previous approaches [Cook et al. 1984; Cook 1986; Cook et al. 1987; Akenine-Möller et al. 2007; Egan et al. 2009] and integrates micropolygon coverage over time (for motion blur) and camera lens aperture (for defocus) using stochastic point sampling. Specifically, it analyzes the cost and algorithmic efficiency of two algorithms for stochastic rasterization. The first is a data-parallel implementation of a previously published method by Pixar. The second algorithm leverages interleaved sampling to decouple rasterization cost from the amount of scene motion or defocus. At the cost of some loss in image quality, this algorithm outperforms the Pixar approach when rendering objects undergoing moderate defocus or high motion. It has the added benefit of predictable performance and is attractive for real-time use.

As in Chapter 4, the analysis provided here (initially presented in [Fatahalian et al. 2009]) focuses specifically on the problem of sampling micropolygon-screen coverage. Motion and defocus blur also require change to how shading is performed in the GPU pipeline. The problem of shading blurred micropolygons is discussed in Chapter 6.

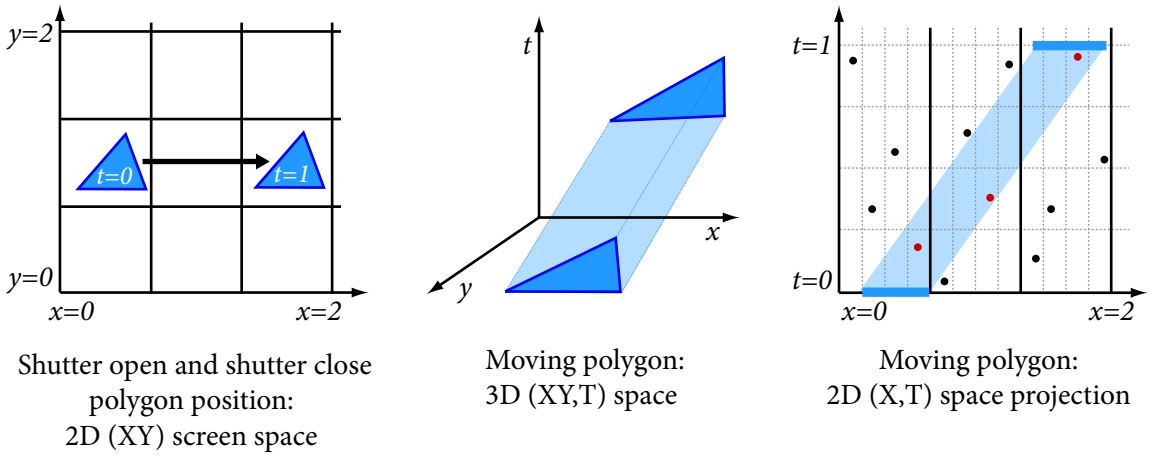


Figure 5.2: Left and center: A polygon moving through (XY,T) space with linear motion. On screen, its motion is in the positive X direction. Right: A simplified illustration showing only one spatial dimension (X,T plane). Sample points are stratified in space and time. Only points lying inside the shaded region result in hits.

5.1 5D Rasterization

A camera's sensor (or film) measures the amount of light entering the camera over the duration of an exposure. The analog of this process when rasterizing micropolygons with motion blur is to integrate micropolygon-screen coverage over the same period of time.

Figure 5.2 shows a blue micropolygon that moves two pixels horizontally during a frame. Its position at the start (t=0) and the end (t=1) of the frame's exposure is shown on the 3×3 pixel grid at left. As the micropolygon moves, its 2D projection sweeps out a volume in a three-dimensional space of screen coordinates and time. This (XY,T)-space volume is illustrated in the center of the figure. For clarity, a simplified view of (XY,T) space displaying only the X spatial dimension is shown at right. In this projected (X,T) view, the volume corresponding to the moving micropolygon is now an area, and is shaded blue. This view spans three pixels in the X direction and shows the location of four multi-sample points per pixel. Unlike tradition rasterization, multi-sample points are now distributed in both space and time.

Estimating the integral of micropolygon-screen coverage from t=0 to t=1 involves finding all multi-sample points in the 3D (XY,T) space that lie within the volume swept out by the moving micropolygon. In the (X,T)-space diagram at right, multi-sample points within the shaded area are covered by the micropolygon (they are colored red). That is, at the time associated with the multi-sample point, the micropolygon covers the 2D screen position of the multi-sample point. In this case, since the micropolygon covers exactly one multi-sample point in each of the pixels, in the final rendered image, each pixel will be a quarter blue.

Figure 5.2 illustrates only motion-blurred rasterization, but a similar process is used to simulate defocus (readers unfamiliar with the process of image formation by a lens are referred to Kolb et al. [1995]). For defocus, multi-sample points are distributed in a 4D (XY,UV) space of screen coordinates and lens aperture positions. A micropolygon covers a multi-sample point in (XY,UV) space if its screen projection, as seen from the point UV on the lens aperture, covers the 2D position of the multi-sample point.

In general, rasterizing a defocused and motion-blurred micropolygon involves finding all multi-sample points in a 5D (XY,UV,T) space that fall within the volume representing the blurred micropolygon. 5D rasterization presents challenges for high-performance implementation. First, point-in-polygon tests in higher dimensions are expensive. Second, and more importantly, it is difficult to localize a moving, defocused polygon in 5D, so generating a tight set of candidate multi-sample points (maintaining high sample-test efficiency) is challenging. To understand why this is the case, again consider the moving micropolygon in Figure 5.2. If the micropolygon was moving faster, the shaded region in the (X,T)-space diagram would be more slanted, but the area of the region would remain the same. The number of multi-sample points covered by an object is proportional its area, so the expected number of samples covered by a moving micropolygon is the same as that when the micropolygon is stationary (assuming micropolygon size does not change greatly during motion). Thus, for motion blurred rasterization to achieve STE (sample-test efficiency: see original definition in Section 4.1 and the example in Figure 4.2) similar to the stationary case, it must perform approximately the same number of point-in-micropolygon tests.

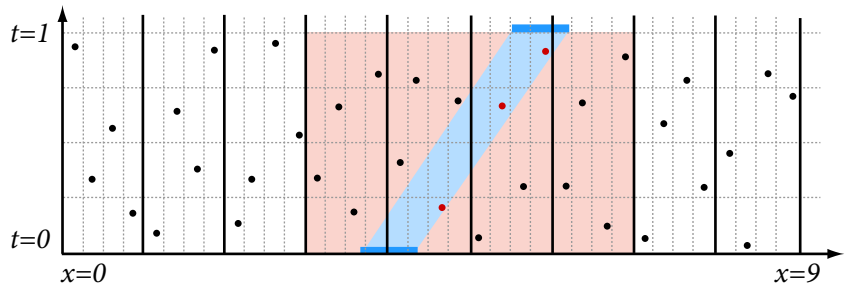


Figure 5.3: Testing all multi-sample points within the spatial bounding box of a moving polygon results in low STE. Point-in-polygon tests are performed against all multi-sample points in the orange region. Only three of these multi-sample points (shown in red) are covered by the micropolygon.

Recall from Chapter 4, that obtaining high STE when rasterizing micropolygons without blur requires tight spatial bounds. A simple way to extend the MPRAST algorithm to the (XY,T) domain is to spatially bound a micropolygon for an entire interval of time, and test all multi-sample points within this bound. Figure 5.3 highlights the multi-sample points tested by this scheme in orange. Because a rapidly moving micropolygon crosses a large region of space during the shutter interval, this approach can result in a significant increase in total point-in-polygon tests. In many cases, the micropolygon is far from a tested multi-sample point at the time associated with the sample. The problem is acute for micropolygons as even small object motion is significant in relation to micropolygon area. It is common for sub-pixel-area micropolygons to move tens of pixels between frames. Inflation of bounds due to defocus is even worse; a modest defocus radius spreads a micropolygon's spatial bound over hundreds of pixels.

In 2D, an axis-aligned bounding box provides tight bounds because micropolygons are compact on screen. However, blurred micropolygons form volumes shaped like long slivers in higher dimensional space. Extending the technique described above to use oriented bounding boxes [Akenine-Möller et al. 2007] or bounding polygons [Wexler et al. 2005; McGuire et al. 2010] yields tighter spatial bounds under conditions of non-screen-axis-aligned motion, but these tricks provide little benefit

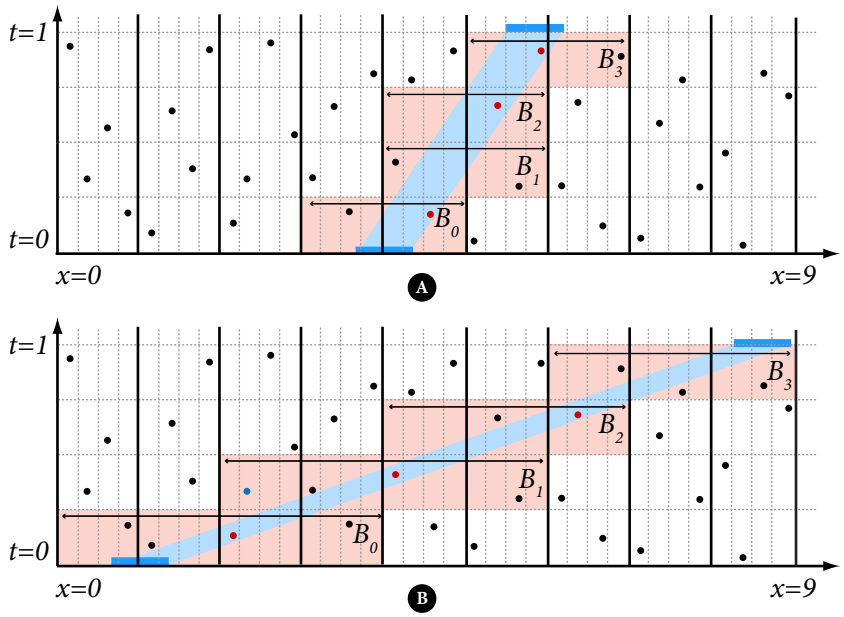


Figure 5.4: INTERVAL uniformly partitions time into intervals, then bounds the XY extent of the micropolygon in each interval. For each of the four intervals shown above, multi-sample points lying within the time interval and within the micropolygon's spatial extent are tested against the micropolygon. Spatial bounds for each interval, B_i, are tight when a polygon is moving slowly (A) but loosen as the amount of motion increases (B). INTERVAL is most efficient for slow moving micropolygons.

for defocus blur and still fail to account for what part of a bounding region a micropolygon occupies at various moments during an exposure (or when viewed through a specific part of the lens). The two algorithms described in the remainder of this chapter increase the STE of 5D rasterization by bounding a micropolygon's position over compact intervals of time and aperture.

5.2 Interval Algorithm

A 5D rasterization approach by Pixar [Cook et al. 1990] leverages stratified sampling to quickly compute a tighter candidate sample set for blurred micropolygons. For simplicity, it is described here under conditions of only motion blur (the 3D (XY,T) case) and then generalized to full 5D rasterization.

Pixar's approach generates a unique set of S stratified time values for each region of space (the published embodiment generates S stratified samples per pixel). Stratification partitions the time domain into S intervals. For each micropolygon, the algorithm iterates over all intervals, computing the spatial bounding box of the micropolygon for a given interval of time. Given this bound for each interval, it tests only the multi-sample points that fall within the interval's spatial extent. There is exactly one such multi-sample point per pixel due to the stratified nature of the samples. Because this algorithm iterates over intervals of time (intervals of time and lens aperture in the 5D case), it will be referred to as INTERVAL. Pseudocode for INTERVAL is given below:

```
for each MP:
  for each STRATUM:                                    // S iterations
    BBOX = compute axis-aligned pixel bbox for MP given STRATUM
    for each pixel P in BBOX:
       test MP against sample from STRATUM in P
```

The behavior of INTERVAL in the (X,T) plane is illustrated in Figure 5.4. Pixel boundaries are indicated by vertical black lines. (X,T) strata boundaries appear as dotted gray lines. Multi-sample points in the orange shaded region are tested against the micropolygon. INTERVAL's STE depends on the spatial bound, B_i, of the micropolygon over the time range associated with each stratum. Therefore, STE depends on object velocity. When the micropolygon is moving slowly (Figure 5.4-A), INTERVAL yields high STE because the polygon can be tightly localized in space (the B_i's are small). For a stationary object, INTERVAL behaves similarly to MPRAST, except spatial bounding boxes are clamped to pixel, rather than sub-pixel boundaries. STE decreases as object motion becomes large (Figure 5.4-B). For example, an object streaking across the screen can decrease the STE of INTERVAL sharply. Notice that the STE of INTERVAL not only depends on the magnitude of motion, but also on its direction (horizontal or vertical motion produce tighter bounds than diagonal motion). Although not implemented in Cook et al. [1990], bounding the micropolygon using

an oriented box decreases this directional dependence.

INTERVAL extends gracefully to full 5D (XY,UV,T) rasterization by pairing UV and T strata. When constructing the 5D position of multi-sample points, INTERVAL always pairs values from the same UV stratum with values from the same T stratum. The association of ranges in T and UV can be constructed in any manner (our implementation uses a random pairing), but must be the same for all samples in the image (it is XY invariant). Because of this property, the range of sample UV and T values for the current stratum is immediately computable given a stratum index. INTERVAL uses these bounds on sample UV and T values to bound the micropolygon spatially. It places no other constraints on the properties of UV and T values used or on the XY location of multi-sample points.

5.3 Interleave Algorithm

5.3.1 Algorithm

The INTERVAL algorithm exhibits two performance characteristics that are not desirable in a real-time system. First, its efficiency decreases under conditions of high motion and defocus blur, especially at real-time multi-sampling rates where only a few multi-sample points per pixel and, correspondingly, a few intervals are used. Second, real-time systems benefit from a predictable frame rate, so it is desirable to decrease the sensitivity of rasterization performance to object velocity or defocus blur (object velocity, in particular, is difficult for a game designer to constrain).

5D rasterization using interleaved sampling [Keller and Heidrich 2001; Mitchell 1991] in the UV and T dimensions avoids both of these problems. The key idea of this approach is that every image multi-sample point is assigned one of N unique UVT tuples $uvt_i = (u_i, v_i, t_i)$. Consider an image as a grid of tiles where each tile contains N multi-sample points and covers a K_x by K_y region of the screen. Within a tile, each multi-sample point is assigned a unique tuple uvt_i. Thus, each of the N tuples is used exactly once per tile and all multi-sample points located at t_i in 5D space are also located at u_i and v_i. This property of the interleaved sampling pattern is exploited

by the following algorithm for 5D rasterization (referred to as INTERLEAVE):

```
for each MP:
  for each unique UVT tuple (ui,vi,ti):              // N iterations
    MP_POSITION = compute MP position at ui,vi,ti
    BBOX = compute tile-grid bbox from MP_POSITION
    for each TILE in BBOX:
      test MP against sample with tuple (ui,vi,ti)
```

The behavior of INTERLEAVE in the simplified (X,T) case is illustrated in Figure 5.5. Notice that all multi-sample points in the image are assigned one of only eight unique times (N=8). This assignment is repeated every two pixels in space (K_x=2). The tile boundaries are illustrated by bold vertical black lines in the figure. For each unique time, INTERLEAVE computes the *exact* position of the micropolygon at the time (this is not a bound over an interval of time) and determines micropolygon overlap with screen tiles. Our implementation uses the micropolygon's axis-aligned bounding box to quickly compute tile overlap. The micropolygon is then tested against the one multi-sample point within each overlapped tile that is associated with the current time. Recall that by construction, there is only one such multi-sample point per tile.

The STE of INTERLEAVE is independent of the amount of motion or defocus blur of rasterized geometry. Any micropolygon, regardless of whether it undergoes slow or fast motion, is tested against at least N multi-sample points (INTERLEAVE's loop over UVT tuples involves N iterations). In practice, a micropolygon will be tested against more than N multi-sample points due to overlap of multiple tiles for each tuple. On average, this overlap depends only on the size of the micropolygon and tiles.

The parameter N serves as a performance-quality knob for INTERLEAVE. Large N yields potentially higher sampling quality (more unique time and lens values are used) at the expense of increased rasterization cost. Note that the spatial extent ($K_x \times K_y$)

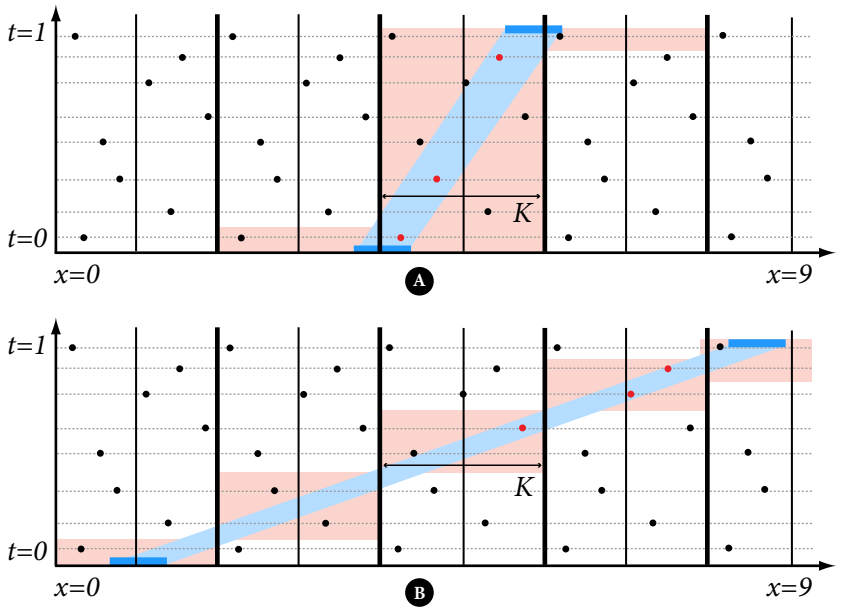


Figure 5.5: INTERLEAVE performs a separate rasterization step for each unique multi-sample time value (indicated by horizontal dotted lines). The micropolygon's position is determined exactly at these times, so STE is independent of object velocity. Approximately the same number of tests are performed against the slow (A) and fast (B) moving micropolygons. Bold vertical lines indicate tile boundaries for the interleaved sample pattern (two-pixel tiles).

of tiles is fixed once a cost budget of N tuples and a desired multi-sampling rate is selected. Said differently, for a given multi-sampling rate, the STE of INTERLEAVE is directly proportional to the size of the interleaving tile.

It is helpful to consider the relationship between INTERVAL's per-strata polygon spatial bounds (B_i in Figure 5.4) and INTERLEAVE's tile size (K_x in Figure 5.5). Intuitively, the STE of INTERVAL and INTERLEAVE can be compared by comparing the average B_i to K_x. For example, INTERLEAVE is more efficient when the tile size is small in comparison to the amount of micropolygon translation over an interval of time. In practice, comparing B_i to K_x provides only a coarse estimate of STE (and overall cost) because a polygon may overlap multiple tiles. A detailed comparison of the STE achieved by both algorithms is provided in Section 5.5.

5.3.2 Permuting Sample Positions

Intuitively, INTERLEAVE rasterizes a stationary micropolygon a total of N times, each time to a multi-sample buffer containing $1/N$ of the samples in the full multi-sample frame buffer. This approach is similar to Haeberli et al.'s [1990] procedure for rendering motion and defocus blur using the accumulation buffer. In fact, Haeberli et al.'s method is an implementation of interleaved sampling with a tile size of one pixel (the number of rendering passes is given by N). Both Keller et al. and Haeberli et al. arrive at the use of interleaved sampling techniques in rasterization as extensions of uniform sampling. Keller et al. emphasize the performance benefits of sampling on regular grids then suggests interleaving regular sampling grids to reduce aliasing. Similarly, Haeberli et al. introduce the accumulation buffer as a mechanism for integrating many uniformly sampled images. In contrast, INTERLEAVE leverages interleaved sampling to bound blurred micropolygons tightly on screen, not to exploit a uniform 2D sampling structure. Unlike these previous uses of interleaved sampling, INTERLEAVE does not position multi-sample points with the same UVT position on a uniform spatial grid. It is important to distribute these multi-sample points non-uniformly in the frame buffer to reduce aliasing.

Figure 5.6-left shows an interleaved sampling pattern constructed by repeating a 2×2 pixel tile across an image (K_x=2, K_y=2, N=16). All samples colored red share the same UVT value. These samples form a regular grid on screen. This regularity yields sampling artifacts even when N is made large enough to eliminate strobing in blurred images. The middle column of Figure 5.7 (a defocused ruler) and Figure 5.8 (a motion-blurred resolution chart) show a zoomed view of artifacts resulting from repeating 1×1-, 2×2-, 4×4-, and 8×8-pixel tiles across a 16×-multi-sampled image. The size of the interleaving tile is particularly noticeable in the renderings of the defocused ruler. Even though interleaved patterns with large tile sizes (lower rows) employ more unique UVT values to sample coverage, artifacts from these sampling patterns are arguably the most objectionable.

Without increasing the multi-sampling rate or N, image quality can be improved by varying the XY position of a UVT tuple in each tile. INTERLEAVE permutes the mapping of UVT values to tile-relative XY positions on a per-tile basis (each tile is

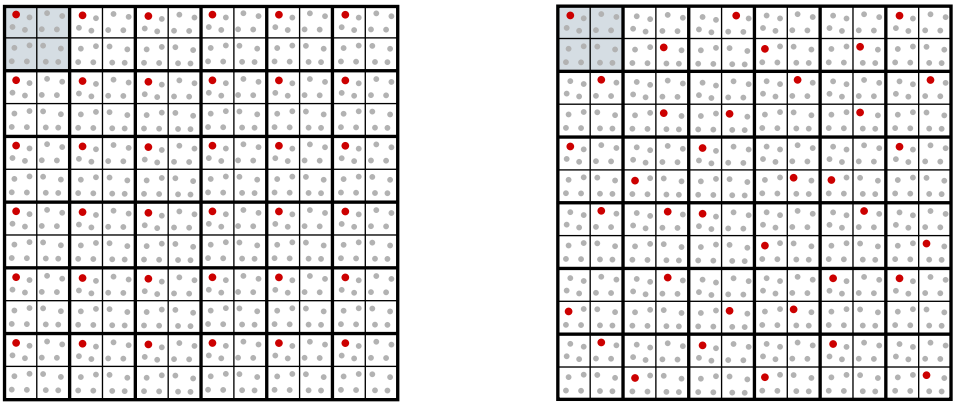


Figure 5.6: Left: Interleaved sampling pattern constructed by repeating a 2x2 pixel tile of UVT values across an image. Multi-sample points with the same UVT value form a uniform grid (red dots). Right: The tile-relative XY position of multi-sample points with the same UVT value is permuted on a per-tile basis, resulting in non-uniform spacing across the image.

associated with one of 64 precomputed permutations). The permutations preserve stratification of UVT values within each pixel. There remains exactly one sample per image tile located at UVT_i, but these samples no longer form a regular grid across the image. An example of an interleaved sampling pattern (K_x=2, K_y=2, N=16) resulting from these permutations is shown at right in Figure 5.6.

The zoomed images in the right column of Figures 5.7 and 5.8 show the effect of interleaved sampling with permutations. Although the same tile size and same UVT values are used to render images in each row of the figures, the aliasing artifacts visible in images in the center column are replaced by less objectionable high-frequency noise in their counterparts at right.

Mitchell [1990] notes the connection between interleaved sampling to break up banding caused by aliasing and the use of dithering to randomize quantization error. In terms of this analogy, permutations improve the quality of dithering that occurs when using interleaved sampling by assigning each pixel a different set of S of the N total UVT values (there are N choose S such sets). When permutations are not used, each pixel is assigned only one of only $K_x \times K_y$ unique sets of UVT samples.

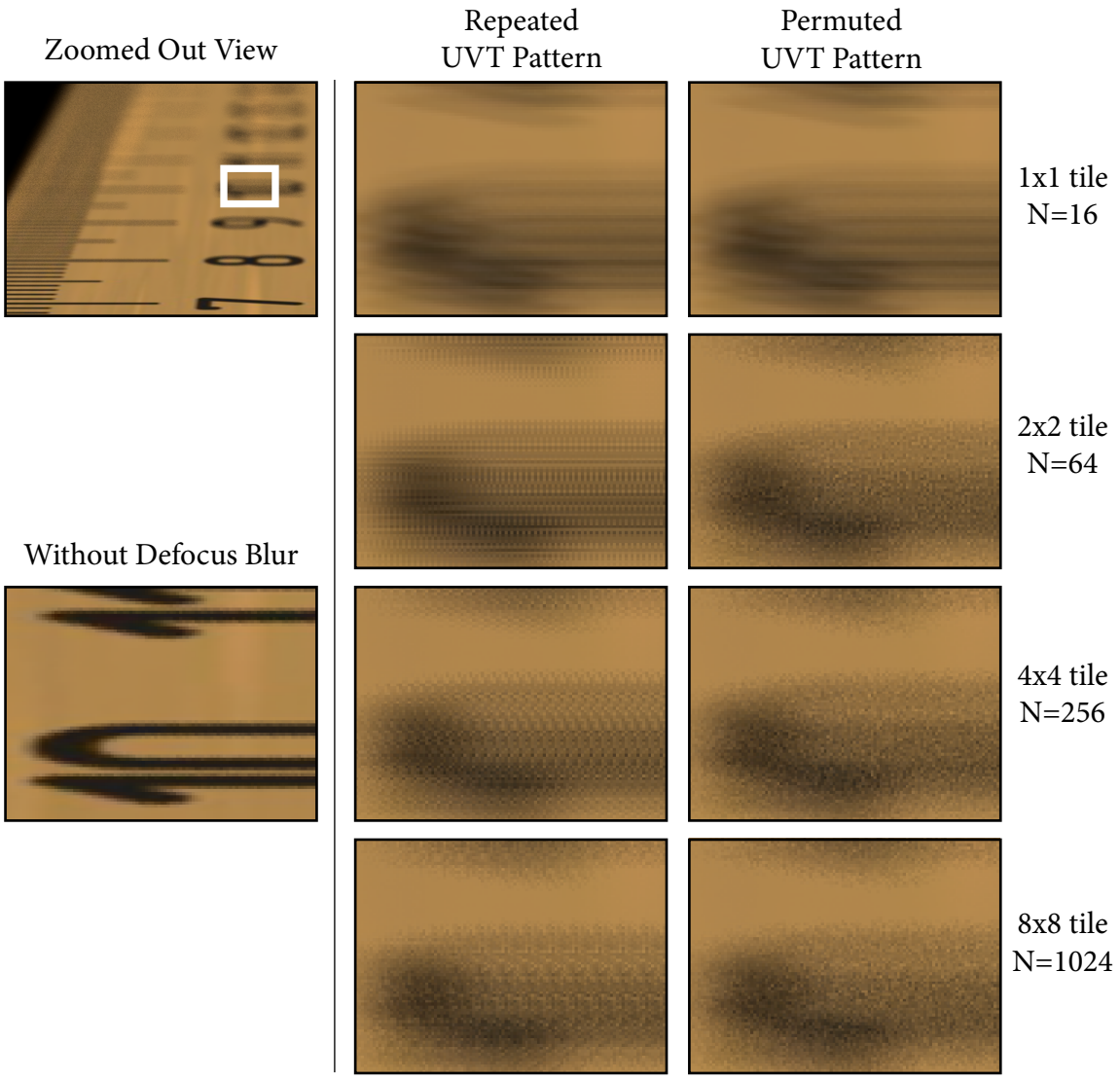


Figure 5.7: Repeating a tile of UV values over the image results in sampling artifacts (center column). The artifacts, which have an appearance similar to the results of poor image dithering, become more objectionable with increasing tile size—they are most noticeable in the center-bottom image, which uses an interleaved sampling pattern containing 1024 unique lens samples. Permuting the position of UV values in each screen tile improves image quality (right column). For each value of N, the same lens positions are used to render both images.

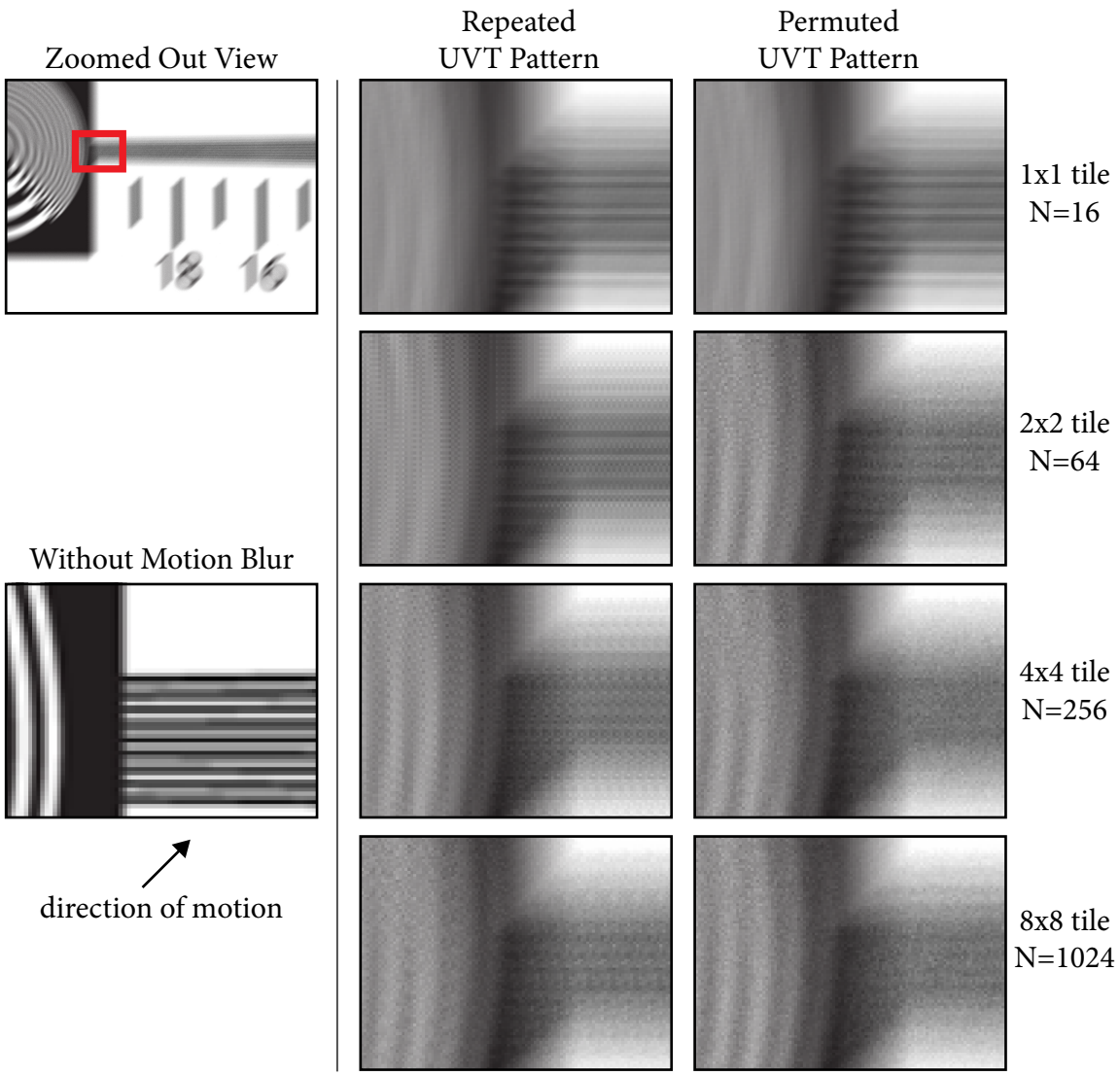


Figure 5.8: Repeating a tile of T values over the image results in sampling artifacts (center column) that become more noticeable with increasing tile size. As was the case in Figure 5.7, images at the bottom of the center column exhibit the most objectionable artifacts. Permuting the position of T values in each screen tile improves image quality (right column). For each value of N, the same time values are used to render both images. In this example, object motion is up and to the right.

The implementation of INTERLEAVE used to render the images in Figures 5.7 and 5.8 permutes only the pixel offset of a UVT tuple within a tile. Each UVT_i is always paired with the same subpixel XY strata. This simplification assigns far fewer unique sets of UVT values to pixels, but it produces satisfactory results, simplifies rasterizer implementation, and permits compact representation of precomputed permutations. Pseudocode that defines the assignment of UVT tuples to XY screen positions is given in Appendix A.

5.4 Data-Parallel Implementation

The implementations of INTERVAL and INTERLEAVE evaluated in this chapter use the same library of fixed-width data-parallel operations used to implement and evaluate MPRAST in Chapter 4. Figure 5.9 lists pseudocode for both data-parallel implementations along with operation counts for the case of full 5D sampling. Details of how these implementations perform a 5D point-in-micropolygon test are provided in Appendix B.

Both INTERVAL and INTERLEAVE test micropolygons against a large set of candidate multi-sample points. As a result, and in contrast to MPRAST, testing many multi-sample points against a single blurred micropolygon in parallel is a viable strategy for achieving high-throughput execution. Conveniently, INTERVAL's loop over intervals and INTERLEAVE's loop over UVT tuples (highlighted in blue in Figure 5.9) do not involve dynamic loop bounds. The number of intervals or UVT tuples is a property of the sampling scheme, so the number of iterations through these loops is known prior to rendering and is micropolygon invariant. Implementations of the two algorithms are parallelized across iterations of these loops. In a data-parallel implementation it is important to select multi-sampling rates so loop bounds (S or N) are multiples of the data-parallel operation width.

Because the number of unique UVT tuples used by INTERLEAVE is large ($N \geq 64$ in the subsequent evaluation), INTERLEAVE can make use of many execution units while processing only a single micropolygon at a time. However, there are fewer INTERVAL sampling intervals than INTERLEAVE UVT tuples ($S < N$). When INTERVAL

INTERVAL (5D)

	`for each MP:`	`// (optionally parallel)`
Setup	`Back-face cull`	`// 17 ops`
	`for each interval:`	`// S iters (parallel)`
Bound	`  Compute MP bbox over interval`	`// 70 ops`
Test	`  for each pixel in bbox:`	
	`    Compute sample XYUVT`	`// 33 ops`
	`    Position MP given UVT`	`// 36 ops`
	`    Test MP-sample coverage`	`// 24 ops`
Process Hit	`    If sample covered`	
	`      Generate fragment`	

INTERLEAVE (5D)

	`for each MP:`	
Setup	`Back-face cull`	`// 17 ops`
	`for each UVT tuple:`	`// N iters (parallel)`
Bound	`  Position MP given UVT`	`// 36 ops`
	`  Compute MP tile bbox`	`// 27 ops`
Test	`  for each tile in bbox:`	
	`    Compute sample XY`	`// 23 ops`
	`    Test MP-sample coverage`	`// 24 ops`
Process Hit	`    If sample covered`	
	`      Generate fragment`	

Figure 5.9: INTERVAL and INTERLEAVE rasterization algorithms formatted to show similarities in structure. Parallelization occurs over iterations of loops highlighted in blue. Additional parallelization across micropolygons in INTERVAL occurs when there are fewer intervals than data-parallel execution units. Operation counts correspond to the case of full 5D sampling.

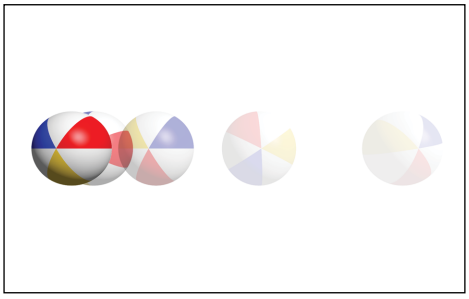
utilizes fewer sampling intervals than rasterizer execution units, it processes a small number of micropolygons simultaneously. For example, INTERVAL processes four micropolygons at once when executing with 16 intervals (S=16) on a platform with 64 data-parallel execution units.

In the data-parallel implementations of both algorithms, utilization of *Test* phase operations depends on the variance of the dynamic loop bounds. For example, INTERVAL's *Test* phase will run at full utilization only if a micropolygon's bounding boxes for all intervals contain the same number of multi-sample points. Rasterizer utilization decreases with increasing *Test* loop bound variance because units done with testing work wait idle until the interval (or tuple) requiring the most point-in-polygon tests completes.

5.5 Evaluation

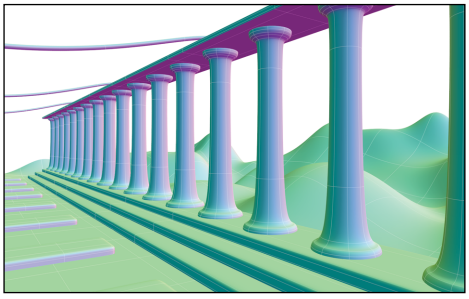
Using the data-parallel implementations of INTERVAL and INTERLEAVE from Section 5.4, we measured the cost and efficiency of motion and defocus-blurred rasterization. In addition to comparing the performance of INTERVAL and INTERLEAVE against each other, we also measured their extra cost relative to the MPRAST algorithm from Chapter 4.

This section follows the same methodology as the previous chapter's evaluation of MPRAST (the reader is expected to be familiar with Section 4.4). We produced data-parallel versions of the INTERVAL and INTERLEAVE algorithms using 8, 16, 32, and 64-wide operations. Implementations execute the pseudocode from Figure 5.9 in lockstep and include no logic to dynamically correct load imbalance to improve execution-unit utilization. We used these implementations to rasterize micropolygons generated by DiagSplit tessellation of the four animated scenes shown in Figure 5.10 (DiagSplit tessellations target 0.5-pixel-area micropolygons). All animations contain motion blur and are rendered at 1728×1080. TALKING also incorporates a defocus effect to draw viewer attention to the character facing the camera. Rasterization of TALKING requires full 5D sampling. All other scenes only require only (XY,T) sampling.



BallRoll

SoccerJump



ColumnPivot

Talking

Figure 5.10: Animation sequences used in algorithm evaluation from left to right: BallRoll, ColumnPivot, SoccerJump, and Talking. All scenes feature motion blurred geometry. Talking also features camera defocus and requires full 5D (XY,UV,T) rasterization.

Estimating time and lens integrals via stochastic sampling requires high sampling rates to eliminate noise. Although the high multi-sampling rates used in offline rendering for film (often more than 64 samples per pixel) will remain out of reach for real-time systems for some time, we found that in many cases 16× multi-sampling is sufficient to reduce noise to visually acceptable amounts. (This assessment is based on personal opinion when inspecting image output, not a scientific study.) Our experiences also indicate only a small number of UVT tuples is necessary for INTERLEAVE to generate acceptable image quality. In general, and especially under conditions of only motion blur, we were satisfied with INTERLEAVE's output using 16× multi-sampling, and 64 unique UVT tuples (N=64, 2×2 pixel tile size), although image quality would not be satisfactory without the interleaving tile permutations described in Section 5.3.2. Noise due to sparse sampling does appear in rendered frames, but it is difficult to perceive under animation. Aliasing and noise in defocused images is more objectionable because the scene is unchanging and a viewer can interrogate image details more closely. Ideally, higher pixel sampling rates should be used for large defocus. In this section, references to INTERLEAVE imply N=64 unless otherwise stated.

5.5.1 The Extra Cost of Blur

Switching rasterizer implementations from MPRAST to INTERVAL or INTERLEAVE has inherent cost, even when all scene geometry is stationary and in sharp focus. We temporarily disabled object movement and defocus blur in the TALKING scene, making the visual output of all algorithms the same. Under these conditions, we found that the STE of INTERVAL (10%) is twice as high as that of INTERLEAVE (5%), but less than half that of MPRAST (26%). Simply enabling support for motion and defocus blur in the rasterizer decreases STE by 2.6×. INTERVAL's STE is approximately equal to that of rasterization without blur using a 4×4 multi-sample stamp. A 4×4 multi-sample stamp spans one pixel of a 16×-multi-sampled frame buffer, therefore it clamps micropolygon bounds to a similar screen granularity as INTERVAL.

The overall cost of rasterizing a micropolygon using INTERVAL is 4.6× greater

than MPRAST. This difference is nearly two times greater than the relative difference in STE because sample tests in the full 5D (XY,UV,T) sampling case cost more. Configuring INTERVAL to disregard defocus computations and use only 3D (XY,T) sampling narrows the performance gap with MPRAST to 3×. INTERLEAVE performs 6.7× (5D sampling) and 5.8× (3D sampling) more processing than MPRAST.

5.5.2 Animated Scene Costs

As stated in Section 5.2, as scene motion or defocus increases, the relative performance of INTERLEAVE improves with respect to that of INTERVAL. Figure 5.11, plots the average cost of rasterizing a micropolygon using each algorithm, for all frames of the four animations. In these sequences, the rasterizers simulate blur from a 1/48 of a second exposure (a common exposure setting for film cameras). Operation counts refer to 32-wide data-parallel operations.

As expected, INTERVAL's performance fluctuates more widely over the sequences than INTERLEAVE's. Even within a short sequence such as SOCCERJUMP (1.5 sec), the performance of INTERVAL varies as much as 4×. The performance of INTERLEAVE is not constant, but varies much less dramatically. We found that INTERLEAVE's STE is nearly uniform over the sequences, and attribute its performance variation to two causes. First, INTERLEAVE's utilization of parallel execution units is greater at low amounts of blur (this effect is discussed further in the next section). Second, under conditions of fast motion, the pipeline back-face culls fewer micropolygons prior to sampling (polygons that begin the shutter interval back facing but flip as a result of motion cannot be culled) resulting in increased average per-micropolygon cost.

The animations show that object motion must be very fast (quickly moving hands in conversation, a camera jerk, an athletic jump) to equate algorithm performance. Although INTERVAL may introduce variation in rasterization costs, these costs are often lower than those of INTERLEAVE. However, when defocus is present, INTERLEAVE outperforms INTERVAL. The difference in cost can be extreme. For example, INTERLEAVE performs seven times fewer operations per micropolygon than INTERVAL when rendering TALKING.

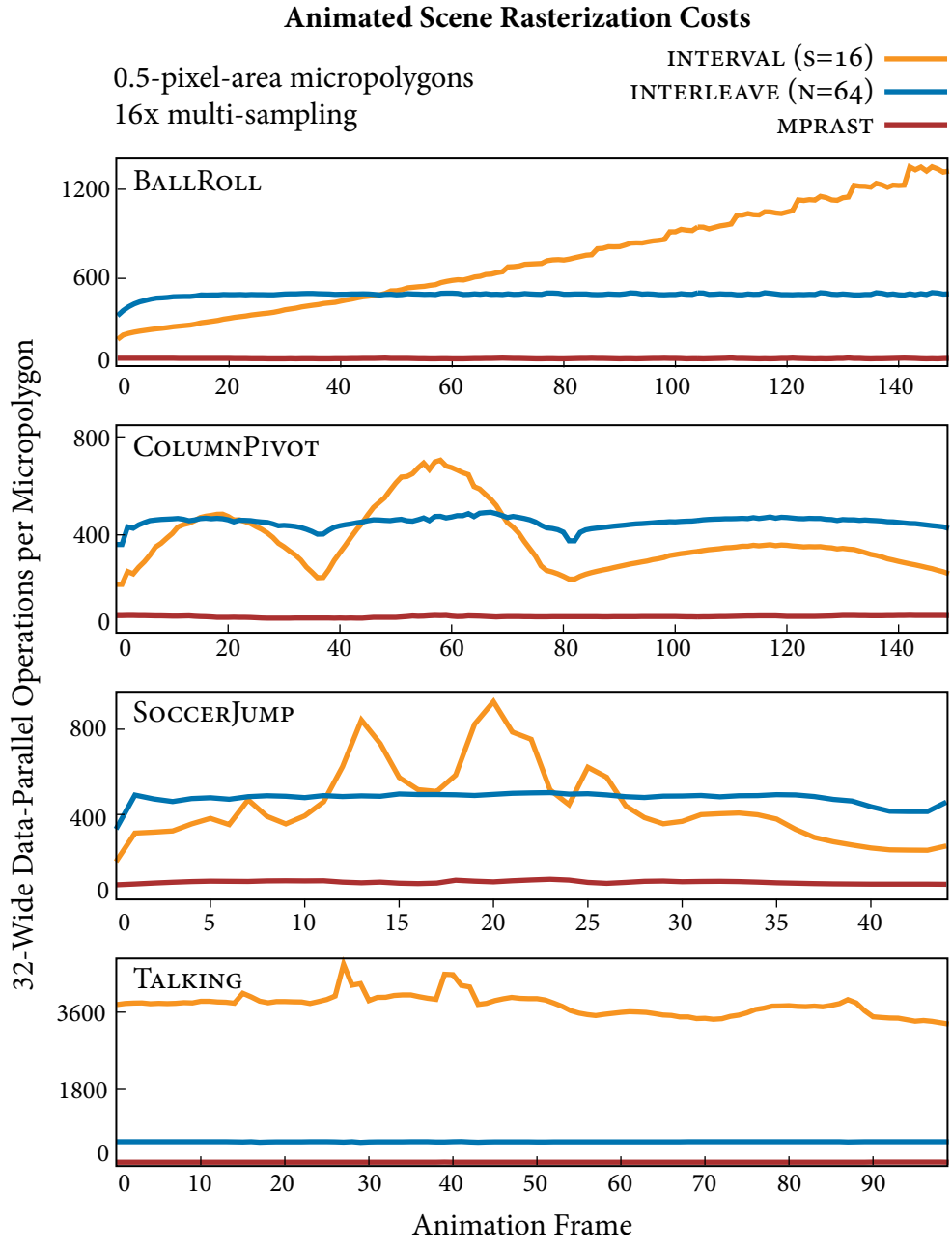


Figure 5.11: INTERVAL exhibits significant variation in rasterization cost (measured in data-parallel operations). In the SOCCERJUMP animation sequence, INTERVAL's per-micropolygon rasterization cost varies by nearly 4× over just 45 frames of animation. INTERLEAVE's cost is nearly invariant of the amount of scene motion or defocus.

5.5.3 Controlled Study

To gain further insight into the amount of blur required to equate INTERVAL and INTERLEAVE performance, we constructed a scene containing randomly positioned and randomly oriented triangle micropolygons of *exactly* 0.5 pixels in area. We measured the STE and per-micropolygon operation-count of both algorithms as the magnitude of micropolygon motion (Figure 5.12-top) or defocus blur (Figure 5.12-bottom) is steadily increased.

The results in Figure 5.12 are consistent with those from the animation scenes. The y-intercept of the STE curves is similar to the 5% and 10% STE measured when rendering animation scenes without scene movement and in sharp focus. In this controlled setup, between 21 and 40 pixels of motion blur are required to equate INTERVAL's STE with that of INTERLEAVE. This corresponds to fast object motion (an object motion crossing 40 pixels in 1/48 of a second will cross a 1728×1080 screen in 0.9 seconds).

INTERLEAVE obtains the same STE as INTERVAL when a micropolygon's defocus blur radius is only two pixels. At high screen resolutions, modest camera defocus yields a blur radius significantly larger than this amount. Under these conditions, the cost of INTERVAL increases greatly. With only ten pixels of defocus blur, INTERLEAVE can utilize 256 unique lens samples and still provide greater performance. This explains the extreme difference in algorithm cost in TALKING.

Figure 5.12's STE and operation-count curves differ in two notable ways. First, the INTERVAL-INTERLEAVE crossover points shift upward when operation-count is considered (between 38 and 60 pixels of motion blur is necessary to equate algorithm cost). This is primarily due to higher utilization of data-parallel execution by INTERVAL. Second, notice that for smaller amounts of blur, INTERLEAVE's STE curves are flat, but its operation-count curves are not. This effect is also due to execution-unit utilization. When a micropolygon is not blurred, it has the same tile bounds for all UVT tuples; iteration in *Test* exhibits perfect utilization. As the polygon is blurred, variance in tile bounds increases, dropping utilization. Utilization stabilizes once blur becomes large with respect to the INTERLEAVE tile size. This effect is partially responsible for the dips in the INTERLEAVE curves in Figure 5.11.

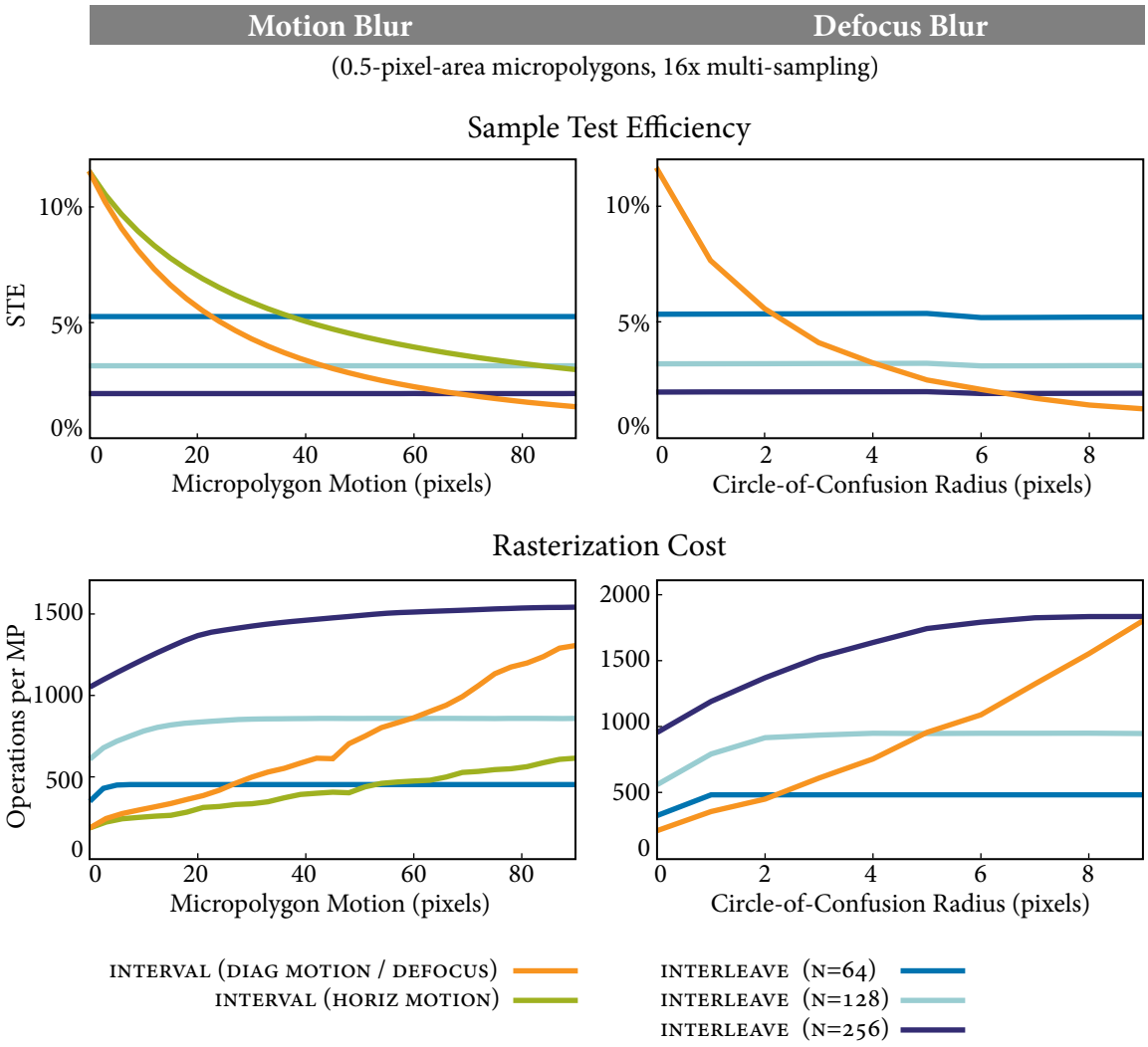


Figure 5.12: INTERVAL's STE (and performance) drops as motion blur (top) and defocus blur (bottom) increase. At high motion, but only small defocus, INTERVAL's performance drops below that of INTERLEAVE. INTERLEAVE's parameter N determines where the performance crossover point lies.

	Data Parallel Execution Units			
	8	16	32	64
INTERVAL				
Setup	.01 (.12)	.02 (.06)	.02 (.06	.02 (.06)
Bound	.06 (1.0)	.05 (1.0)	.05 (1.0)	.05 (.99)
Test	.93 (.92)	.93 (.89)	.94 (.86)	.94 (.83)
Overall Util	.92	.88	.86	.83
INTERLEAVE N=64				
Setup	.01 (.12)	.02 (.06)	.04 (.03)	.07 (.02)
Bound	.23 (1.0)	.23 (1.0)	.21 (1.0)	.20 (1.0)
Test	.76 (.67)	.76 (.63)	.75 (.61)	.73 (.60)
Overall Util	.74	.70	.68	.64
INTERLEAVE N=256				
Setup	.01 (.12)	.01 (.06)	.01 (.03)	.02 (.02)
Bound	.31 (1.0)	.30 (1.0)	.27 (1.0)	.25 (1.0)
Test	.69 (.67)	.70 (.63)	.72 (.55)	.73 (.51)
Overall Util	.77	.74	.67	.62

Table 5.1: Fraction of total execution time and data-parallel unit utilization (in parentheses) of each stage of INTERVAL and INTERLEAVE (N=64 and N=256 configurations shown).

5.5.4 Utilization

Table 5.1 provides detailed execution statistics for each phase of INTERVAL and INTERLEAVE. Statistics are averaged over a collection of frames from the four animation scenes.

Both INTERVAL and INTERLEAVE realize lower STE than MPRAST and correspondingly spend an even higher fraction of time performing sample tests (positioning the micropolygon is hoisted out of the inner loop of INTERLEAVE, so much of the time spent in *Bound* can be considered "testing" work). INTERVAL's *Test* phase constitutes over 93% of execution time and maintains over 83% utilization when scaling to 64 execution units. Dividing UVT-space into equal intervals yields similarly sized spatial bounds for each interval, resulting in low variance in the number of iterations through the inner *Test* loop. Recall that scaling INTERVAL to many execution units requires

several micropolygons to be processed at once. Micropolygons reaching the rasterizer in succession typically originate from the same region of a tessellated surface. They undergo similar motion and have similarly-sized interval bounds.

INTERLEAVE achieves lower utilization in these critical regions because there is higher variance in micropolygon-tile overlap than in the size of INTERVAL's bounding boxes. Still, INTERLEAVE remains amenable to wide data-parallel scale out. Although INTERLEAVE does not fully utilize eight execution units (67%), its utilization drops only to 60% when scaling to 64 execution units. INTERLEAVE spends a larger fraction of time in *Bound*, which always runs at full utilization. As a result, further optimization to increase utilization of *Test*, such as dynamically repacking sample testing work into batches, will yield a performance benefit of at most 41%.

Although Table 5.1 only displays execution statistics for rasterizing 0.5-pixel-area micropolygons at 16× multi-sampling, analysis of INTERVAL's and INTERLEAVE's execution under the load of different micropolygon sizes and different multi-sampling rates exhibits very similar data-parallel behavior.

5.6 Discussion

This chapter analyzed the sampling cost of simulating motion and defocus blur in a rasterizer. Depending on scene characteristics, the cost of rasterization with blur exceeds that of rasterization without blur between three and seven times. Although blurred rasterization does entail more expensive point-in-micropolygon tests, the key to reducing sampling costs is limiting the number of sample tests performed. INTERVAL and INTERLEAVE take different approaches toward this goal. While the STE of INTERVAL depends heavily on scene characteristics, INTERLEAVE leverages interleaved sampling to provide blur-independent performance. As a result, INTERVAL's performance varies from more than two times greater than INTERLEAVE (in cases of minimal blur) to up to seven times less (under moderate defocus). The predictable performance of INTERLEAVE is attractive for real-time rendering, however in the (XY,T) sampling case, object motion must be large before INTERLEAVE's cost drops below that of INTERVAL.

Even though INTERVAL and INTERLEAVE strive to bound micropolygons tightly, in terms of STE, both algorithms are inefficient. At 16× multi-sampling, only one in twenty tests performed by INTERLEAVE identifies a covered multi-sample point; the situation is even worse for INTERVAL under high blur. Although the compute-intensive nature of 5D rasterization does lend itself well to fixed-function acceleration [Brunhaver et al. 2010], further work should attempt to improve these ratios. For example, extending the INTERVAL algorithm to operate on hierarchies of micropolygons, rather than individual micropolygons, was considered during this work, but never pursued.

Optimal design of interleaved sampling patterns remains an interesting problem. At low multi-sampling rates and small N, interleaving uniform grids of samples does not produce satisfactory image quality. INTERLEAVE became a viable algorithm only when coupled with a sampling pattern that distributed UVT values non-uniformly on screen. The method of generating these patterns described in this chapter is admittedly ad hoc. A more principled approach to designing interleaved sample patterns (and corresponding image reconstruction filters) will likely achieve even better INTERLEAVE image quality.

Last, in contrast to the 5D sampling approach pursued here, there are many techniques for approximating motion and defocus blur effects cheaply by post-processing rendered output. Sung et al. [2002] and Demers [2004] provide surveys of motion blur and defocus blur techniques respectively. Listings of more recent techniques appear in [Akenine-Möller et al. 2007; Lee et al. 2010; Ritchie et al. 2010]. These approaches work well in some regions of a frame but often produce artifacts such as color bleeding across depth or object discontinuities. Even so, many current games use image post-processing to produce blur effects and force artists to carefully create content or limit camera configurations to obtain good results. While there will always be a use for fast approximations in real-time applications, the robustness and accuracy of directly integrating micropolygon-screen coverage over time and lens aperture should motivate its inclusion in future GPU rasterizers. Since motion is pervasive in games, and motion-blurred (XY,T) rasterization requires less computational cost than defocus, it is likely that GPU rasterizers will adopt 3D (XY,T) rasterization before supporting full 5D sampling.

Chapter 6

Quad-Fragment Merging

Previous chapters have presented solutions that increase the efficiency of generating and rasterizing micropolygons in the GPU pipeline. However, it is shading that often constitutes the most expensive part of a rendering workload. Unfortunately, the GPU pipeline performs many redundant shading computations when processing surfaces tessellated into micropolygons. This extra shading can significantly decrease rendering performance. This chapter describes a GPU pipeline extension, called quad-fragment merging [Fatahalian et al. 2010], that overcomes this problem. Without requiring action from the application developer, quad-fragment merging identifies and eliminates many redundant shading computations from the GPU pipeline. On average, this optimization reduces the number of shaded fragments by more than a factor of eight while preserving high image quality.

This chapter begins with an extended description and evaluation of the quad-fragment merging technique. It closes with a brief discussion of quad-fragment merging's merits in comparison to two popular shading techniques that also avoid redundant shading when processing micropolygons: deferred shading and Reyes-style shading at each micropolygon vertex.

6.1 GPU Shading

Shading a surface has high computational cost. It involves computing the illumination incident on the surface due to a scene's many light sources and it requires simulating how the surface scatters light towards the virtual camera. Current Fragment Processing shader programs execute hundreds of instructions to perform these computations. Shader program instruction counts will undoubtedly increase in future games as the light and material models used by artists grow in sophistication (recall that a large percentage of a GPU's processing resources already are responsible for executing shading computations). Given the high cost of shading, current GPUs employ three important strategies to limit the number of shading computations performed when generating images.

- **Occlusion culling prior to shading**. GPUs discard surface regions from the rendering pipeline prior to shading if they are occluded by other surfaces. These surface regions are not visible from the camera and do not impact the final image.

- **Independent visibility and shading sampling densities**. Although GPU rasterizers sample triangle-screen coverage many times per pixel, GPUs shade each triangle uniformly in screen-space at a density of one shading sample per pixel. This technique is called *multi-sample anti-aliasing* (MSAA) [Akeley 1993]. It is acceptable to sample shading more sparsely than coverage because high-frequency texture data is pre-filtered prior to sampling to avoid aliasing.

- **Derivatives via finite differencing**. To determine filter extents for texturing, GPUs estimate surface texture-coordinate derivatives by taking differences between texture coordinates for shading samples in neighboring pixels. Sharing data between neighbors avoids re-computation of nearby shading results when finite-difference derivatives are needed during shading.

The process of rasterizing and shading a triangle in a GPU pipeline is illustrated in Figure 6.1 (the figure expands on Figure 4.1's depiction of multi-sampled rasterization). In this example, the rasterizer samples triangle coverage at four multi-sample

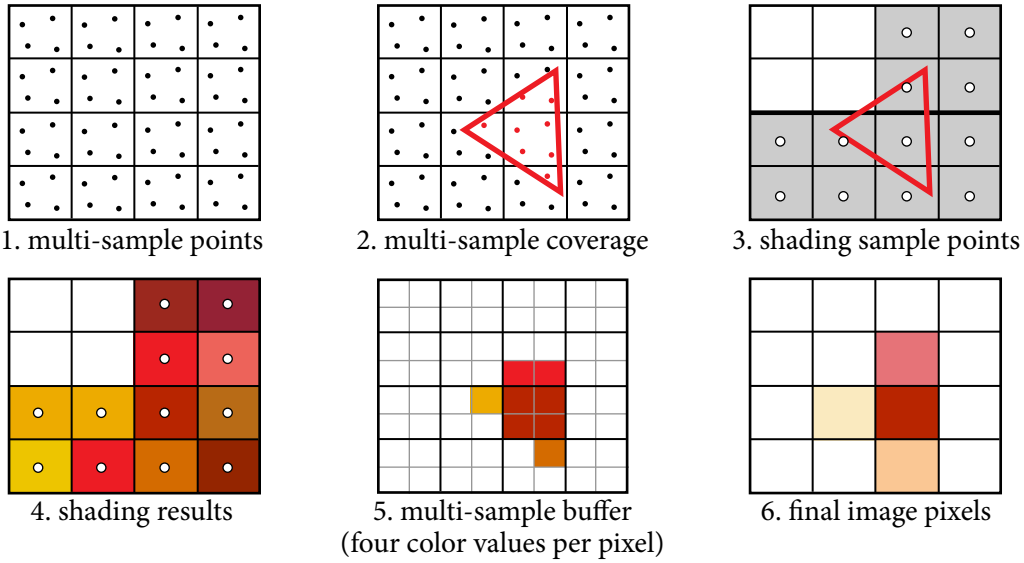


Figure 6.1: Rendering a triangle to a 4×4-pixel screen region using 4× multi-sample anti-aliasing: The triangle's screen coverage is sampled four times per pixel (panels 1–2). Shading is sampled once per pixel, but at the granularity of 2×2 pixel blocks (3–4). The results of coverage and shading computations are stored in the multi-sample buffer (5) and subsequently filtered to produce final image pixels (6).

points per pixel (4× MSAA). Panel 1 of this figure shows a 4×4-pixel region of the screen and, as in previous chapters, it represents multi-sample points as black dots. Panel 2 highlights the multi-sample points covered by the triangle in red.

If any multi-sample point in a pixel is covered by the triangle, the pipeline will execute a shading computation for that pixel. Inputs to the shading computation, such as texture coordinates, are sampled by interpolating values stored at triangle vertices. Typically the pixel center is used as the interpolation/sampling point [Kessenich 2009; Mic 2010b]. Panel 3 shows these *shading sample points* as white dots. If the pixel center lies outside the triangle, the shading inputs are extrapolated from values at vertices. Alternatively, GPUs permit shading inputs to be sampled at the covered multi-sample point which is closest to the pixel center (centroid sampling [Kessenich 2009; Mic 2010b]). Centroid sampling avoids errors due to extrapolation of shading inputs, but results in a non-uniform screen-space sampling of shading near triangle edges.

Fragments emitted by rasterization encapsulate information needed to compute a triangle's contribution to the rendered image at a pixel. This information consists of shading inputs, along with triangle coverage and depth information for each of the pixel's multi-sample points. (For convenience, a fragment is said to "cover" a multi-sample point if the triangle it was created from did.) To support derivative estimates using finite differencing, rasterization generates fragments in 2×2-pixel blocks [Akeley 1993; Kessenich 2009; Mic 2010b]. Blocks of four fragments, called *quad fragments*, are the minimal granularity of shading work in the GPU pipeline. Panel 3 highlights in gray the three quad fragments generated by rasterizing the triangle. Notice that if the triangle covers any multi-sample point in a 2×2-pixel region, a quad fragment is generated at these pixels and shading is computed at all four corresponding pixel centers. The results of shading each fragment are given by the pixel colors in panel 4.

After a fragment is shaded, its color is blended with multi-sample buffer values for all covered multi-sample points (panel 5). Finally, after all rendering for a frame is complete, the multi-sample buffer's contents are used to reconstruct final pixel values (panel 6).

Ironically, quad-fragment shading and multi-sample anti-aliasing (two techniques intended to limit the amount of shading-related work performed by the GPU pipeline) cause GPUs to generate multiple quad-fragments at screen regions where two surface triangles meet. This behavior is illustrated in Figure 6.2, which visualizes the number of quad fragments shaded at each 2×2-pixel region of the screen. When triangles are large (e.g., 50-pixel-area triangles, top row), most 2×2-pixel regions are covered entirely by a single triangle, so the overhead of shading extra fragments near triangle edges is low. However, as average triangle size is reduced, the amount of shading performed by the pipeline grows rapidly. When the BigGuy character is tessellated into one-pixel-area triangles (bottom row), it is shaded more than eight times at each covered pixel. Most of this shading is redundant. Tessellating BigGuy into subpixel micropolygons can cause the GPU pipeline to sample shading more frequently than coverage (the opposite effect of what MSAA intends to achieve). In the worst case, if each micropolygon covers only one multi-sample point, the GPU will sample shading at four times the multi-sample rate.

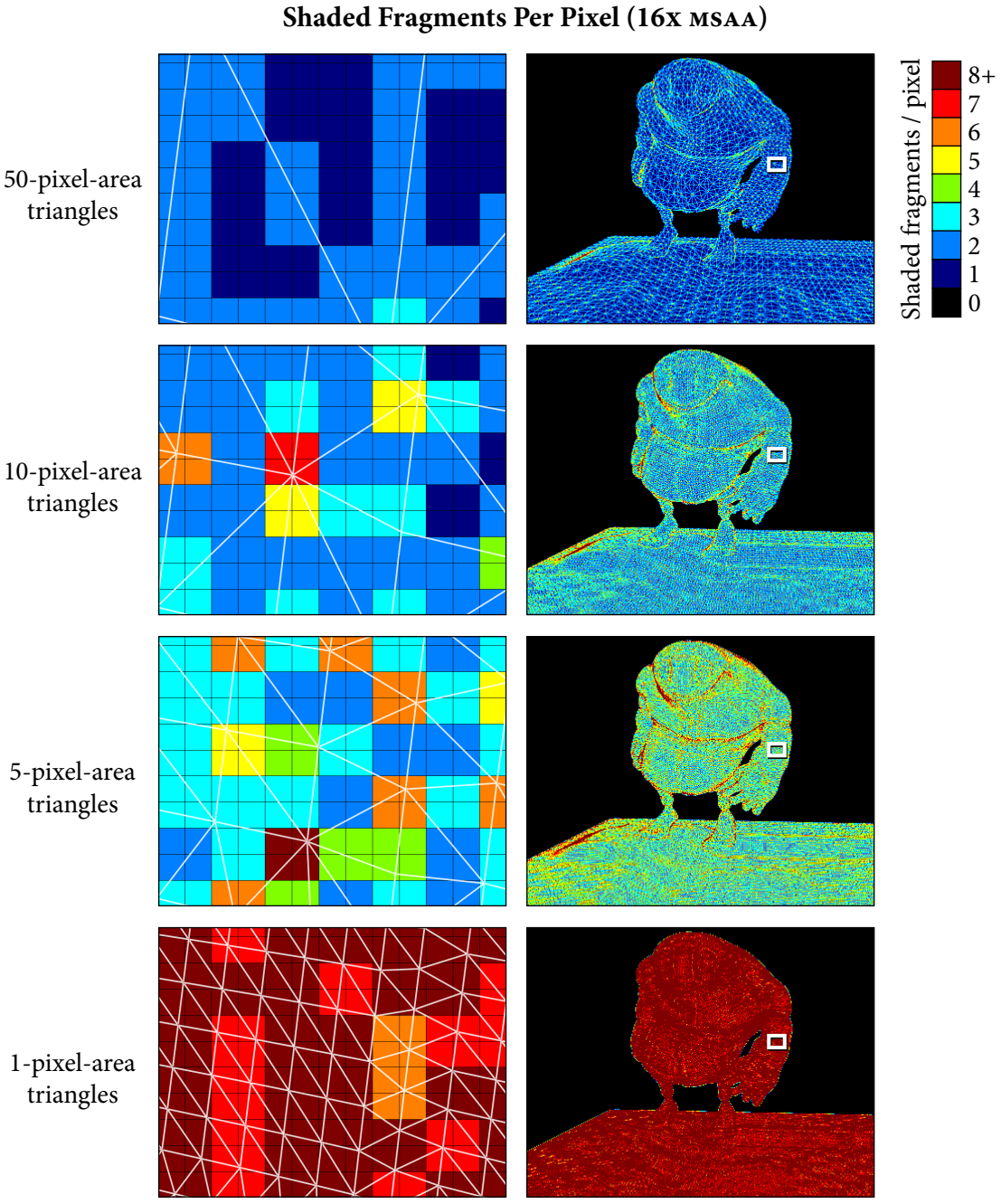


Figure 6.2: Visualization of the number of quad fragments shaded per pixel. The amount of shading performed by the GPU pipeline increases as scene triangle size shrinks. When this scene is tessellated into one-pixel area triangles, more than eight shading computations occur at each covered pixel (zoomed view at left).

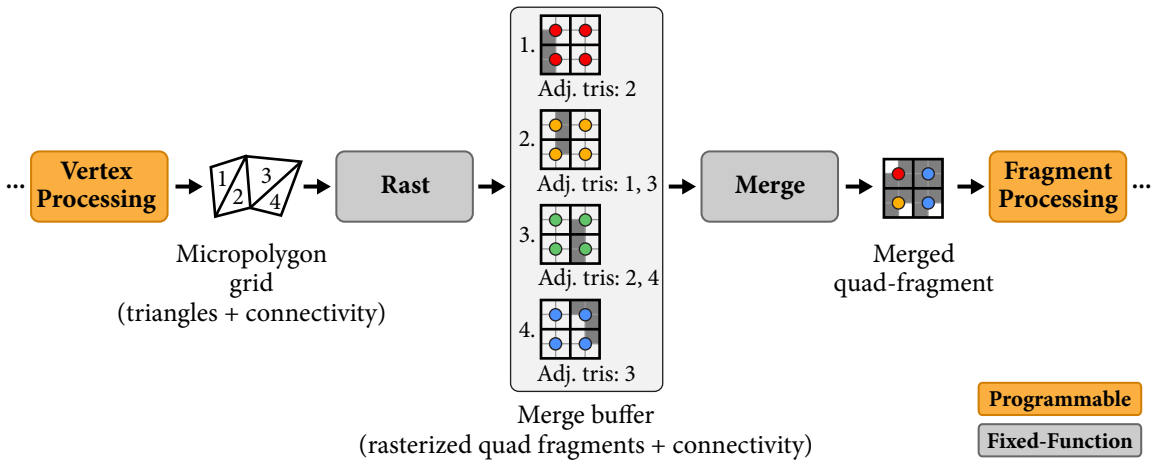


Figure 6.3: A quad-fragment merging pipeline: tessellation produces grids of triangles with adjacency. Triangles are rasterized to produce quad fragments. A new stage, Merge, buffers rasterized quad fragments from nearby surface triangles and combines them into a smaller number of quad fragments prior to shading.

6.2 Quad-Fragment Merging

To limit shading costs, it is desirable to shade entire surfaces, not just individual micropolygons, at a density of about one sample per pixel. Quad-fragment merging achieves this sampling density by merging rasterized quad fragments from micropolygons *belonging to the same surface* into a smaller number of quad fragments prior to shading. An overview of the quad-fragment merging technique and its integration into the GPU pipeline is provided in Figure 6.3.

In a micropolygon rendering pipeline, input to the rasterizer is not a stream of arbitrary triangles. Micropolygons are generated by the pipeline during tessellation (Chapter 3) so the connectivity of triangles in each micropolygon grid is known. In Figure 6.3, this connectivity information is propagated through the pipeline. It is included with the triangles emitted by tessellation and passed along with quad fragments produced by rasterization. A new pipeline stage, called Merge, buffers rasterized quad fragments from a grid. This stage identifies quad fragments (with sparse multi-sample coverage) at the same screen location and from adjacent grid triangles and merges them to reduce the total amount of pipeline shading work. The

output of the Merge stage is a stream of quad fragments (with dense multi-sample coverage). Like rasterized quad fragments in a current GPU pipeline, merged quad fragments are shaded independently using data-parallel processing by the pipeline's Fragment Processing stage.

6.2.1 Merge-Buffer Structures

The primary component of Merge stage is a fixed-size buffer that stores quad fragments for merging (the merge buffer). To facilitate description of the Merge stage's behavior in the ensuing sections, Figure 6.4 provides C-style definitions of a quad-fragment record (`quad_fragment`) and a merge-buffer entry (`buffer_entry`). Quad-fragment records contain surface coverage and depth information at each multi-sample point within a 2×2-pixel region, as well as surface attributes sampled at each of the four pixel centers (or, if centroid sampling, a multi-sample point in the pixel) that are used as inputs for shading (`shade_input_data`). The content of shading inputs is defined by the signature of fragment shaders bound to the pipeline at runtime and includes all interpolated attributes (e.g., texture coordinates, position, normal). The source triangle's sidedness (`facing`) and its coverage of each fragment's shading sample point (`shade_coverage`) are also stored in the quad-fragment record.

Each merge-buffer entry contains a quad-fragment record as well as two bitmasks. The source triangle mask (`tri_mask`) enumerates the triangles that have contributed to the entry's quad fragment. Bit i in the mask is set if the entry's quad fragment was generated by rasterizing grid triangle i, or if quad fragments from triangle i have been merged into the entry. Bits in the adjacent triangle mask (`adj_mask`) are set for all triangles that share an edge with source triangles. In the implementation described here, these masks are sized to support grids of up to 512 triangles (512-bit masks). They could be sized differently to permit merging quad fragments from larger or smaller groups of triangles.

After receiving a quad fragment from the rasterizer, the merging stage first determines if it can be merged with an existing entry in the merge buffer. This logic

```
quad_fragment {
  int           x, y;
  bool          facing;

  BITMASK       coverage;                // 4*MULTISAMPLES bits
  float         z[4][MULTISAMPLES];

  BITMASK       shade_coverage;          // 4 bits
  SHADER_INPUT  shade_input_data[4];
};


buffer_entry {
  quad_fragment frag;
  BITMASK       tri_mask;                // 512 bits
  BITMASK       adj_mask;                // 512 bits
};


bool can_merge(e1, e2) {
  return
      e1.frag.x == e2.frag.x &&
      e1.frag.y == e2.frag.y &&
      e1.frag.facing == e2.frag.facing &&
     (e1.tri_mask & e2.adj_mask) &&
     (e1.frag.coverage & e2.frag.coverage) == 0;
}

// merge quad-fragment in entry e2 into e1
void merge(e1, e2) {
  e1.tri_mask |= e2.tri_mask;
  e1.adj_mask |= e2.adj_mask;
  e1.frag.coverage |= e2.frag.coverage;
  copy_z(e1.frag, e2.frag);
  select_shading_inputs(e1.frag, e2.frag);
}
```

Figure 6.4: Each merge-buffer entry (`buffer_entry`) contains a quad fragment and bitmasks enumerating the quad fragment's source triangles and the triangles that are adjacent to source triangles. Determining whether two buffer entries can be merged (`can_merge`) involves only a few bitwise operations.

is given by the `can_merge` function in Figure 6.4 and is described in detail in Section 6.2.3. If `can_merge` returns true, the new quad fragment is merged with the current contents of the entry. Otherwise, the quad fragment is placed in an available buffer entry. If no entries are available, an occupied buffer entry is chosen for eviction (our implementation evicts the oldest entry) and its quad fragment is emitted by the Merge stage for shading.

The next two sections describe the two principal operations of quad-fragment merging: constructing a merged quad fragment from mergeable inputs and determining when two quad fragments can be merged.

6.2.2 Performing Merges

Figure 6.5 shows two adjacent grid triangles that lie within the same 2×2-pixel region. Rasterizing the triangles using 4× MSAA produces two quad fragments. Multi-sample coverage for each quad fragment is depicted as a gray mask (cells shaded gray in the mask correspond to covered multi-sample points). Notice that since the triangles are small, their multi-sample coverage in the 2×2-pixel region is sparse. Shading inputs for each fragment are represented by the large circles in the figure (four circles per quad fragment). The circles are colored according to the triangle they were sampled from. This visual representation of a quad fragment is used throughout the rest of this chapter.

The function `merge` in Figure 6.4 describes the process of merging quad fragments contained in merge-buffer entries $e1$ and $e2$. Precisely, the pseudocode merges the total contents of the entries, not just their contained quad fragments. The reader should consider a quad fragment arriving at the Merge stage to be represented by an entry record with a single bit set in its `tri_mask` and up to three bits set in its `adj_mask`.

The first step in merging two merge-buffer entries is to combine their coverage and topology bitmasks as well as their depth information. Bitwise operations for combining coverage and topology are given in the pseudocode for `merge`. Merging depth values (`copy_z`) is a data selection controlled by the two quad fragments' coverage

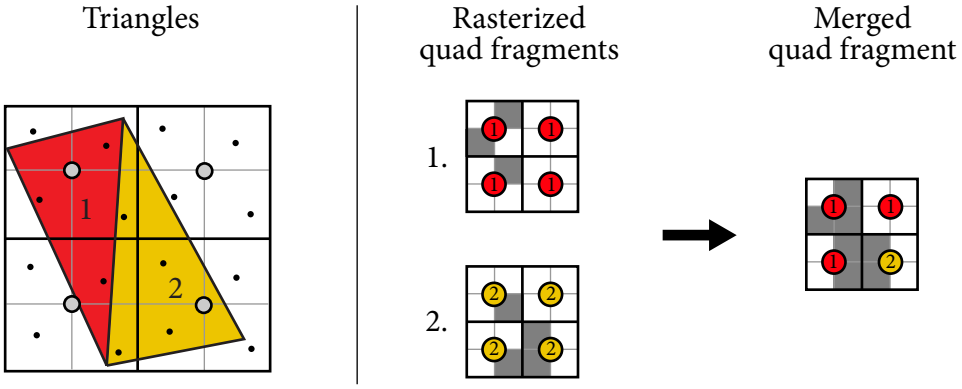


Figure 6.5: Merging two quad fragments involves combining multi-sample coverage and depth information and selecting shading inputs for all four fragments. The example above illustrated all three cases for selecting shading inputs for a merged quad fragment. Case 1: the top-left and bottom-right fragments receive shading inputs from the quad fragments that cover the pixel center. Case 2: the bottom-left fragment is assigned shading inputs from quad fragment 1, because triangle 1 covers the closest covered multi-sample point to the pixel center. Case 3: the top-right fragment is not covered. It receives inputs from quad fragment 1 because its horizontal neighbor (the top-left fragment) is covered by quad fragment 1.

masks. Merging multi-sample coverage and depth information is simple because quad fragments with overlapping multi-sample coverage are never merged (conditions for merging are given in Section 6.2.3).

Last, the Merge stage must associate shading inputs with fragments in the merged result (`select_shading_inputs`). For each fragment in the merged quad fragment, shading inputs are selected from one of the two input quad fragments according to the following rules.

1. If the pixel center is covered by one of the input quad fragments (the top-left and bottom-right fragments in Figure 6.5), shading inputs are selected from the quad fragment that covers the pixel center. In the unlikely case that both quad fragments overlap the pixel center but have non-overlapping multi-sample coverage masks, the fragment is assigned shading inputs from the quad fragment that arrived first at the Merge stage.

2. If the pixel center is not covered, but a multi-sample point within the pixel is

(Figure 6.5: bottom-left fragment), shading inputs are selected from the quad fragment featuring the closest (relative to the pixel center) covered multi-sample point.

3. If neither the pixel center nor any multi-sample point within a pixel is covered (Figure 6.5: top-right fragment), shading inputs are selected from the same input quad fragment as a neighboring fragment that does feature a covered multi-sample point. Our implementation assigns priority to the fragment's horizontal, vertical, and then diagonal neighbor. In the figure, since the top-left fragment is assigned inputs from quad fragment 1, the top-right fragment is also assigned shading inputs from quad fragment 1.

The first selection rule seeks to achieve a uniform, screen-space sampling of surface shading. The second attempts to minimize artifacts when shading inputs are extrapolated outside of triangle boundaries. The third aims to minimize errors in derivative estimates when an entire pixel in a quad fragment is not covered by the surface.

Unlike the quad fragments generated by rasterization, the merged quad fragment, shown at right in Figure 6.5, represents surface coverage and shading inputs drawn from two triangles. This quad fragment is emitted by the Merge stage for subsequent shading. In general, when many triangles from a surface fall within the same 2×2-pixel region, successive merges produce a quad fragment that represents the surface's total multi-sample coverage at these pixels (potentially from many triangles) and contains four uniformly spaced surface shading samples from up to four unique surface triangles. Since only one set of shading inputs is used per fragment, a pipeline with quad-fragment merging will not necessarily sample shading inputs from all triangles in a mesh.

6.2.3 Conditions for Merging

Ideally, to minimize shading costs, the Merge stage will merge all quad fragments at the same screen location into a single quad fragment for shading. However, to

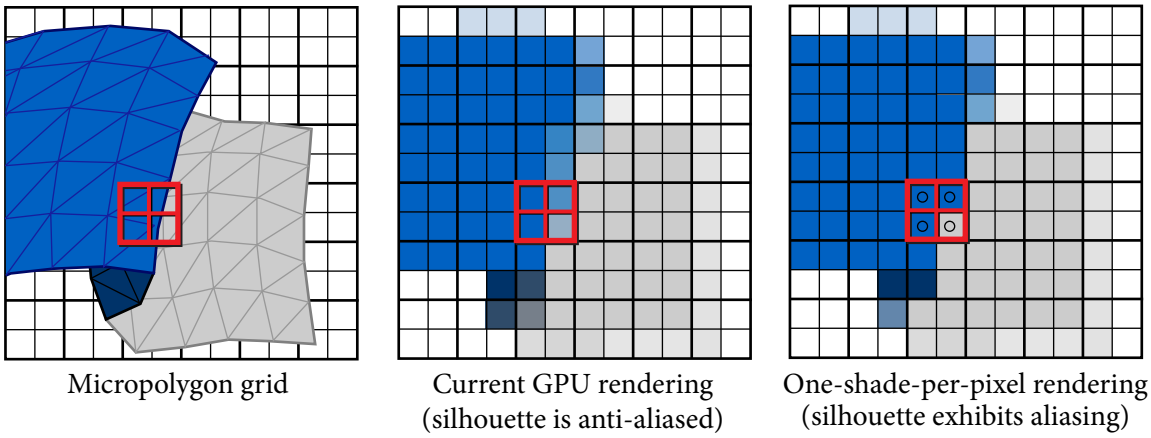


Figure 6.6: A fold in a surface causes blue and gray triangles to project into the same 2×2-pixel region. Center: the current GPU pipeline shades one quad fragment for each triangle overlapping these pixels, then blends the results to produce an anti-aliased silhouette. Right: shading only one quad fragment for the entire surface results in aliasing along the silhouette since the results of a single shading computation are applied to all covered multi-sample points in a pixel.

preserve high rendering quality, the Merge stage must avoid merges that introduce visual artifacts.

Figure 6.6, which shows a micropolygon grid resulting from tessellation of a highly curved surface, provides one example of how merging could introduce rendering artifacts. The surface contains a fold so blue and gray triangles that are far apart on the surface project into the highlighted 2×2 pixel region. Because quad fragments from both blue and gray triangles contribute coverage to these pixels, rendering this grid using the current GPU pipeline with MSAA will produce light-blue pixels along this silhouette edge (center image). However, if only a single merged quad fragment is shaded for all triangles covering these pixels (right image), final pixel values will either be blue or white (the results of a single shading computation are applied to all multi-sample buffer values in each pixel). As a result, the rendered image exhibits aliasing along the silhouette edge. In addition to aliasing, shading results may be erroneous because differences between shading quantities in adjacent pixels are not representative of the derivatives of these quantities along the surface. Quad-fragment shading is based on the assumption that each quad fragment corresponds to a locally

contiguous region of the surface. In this example, since there is a discontinuity between visible parts of the surface along the silhouette, this assumption does not hold so it is inadequate to shade only a single quad fragment at the highlighted 2×2-pixel region.

To prevent artifacts due to merges across surface discontinuities such as the silhouette in Figure 6.6 the merge stage will merge quad fragments only if they meet the following four conditions:

1. They have the same screen pixel location.
2. They have the same sidedness (either front or back-facing).
3. They have source triangles that are adjacent in the mesh.
4. They do not cover the same multi-sample point (they do not occlude each other).

For a pair of merge-buffer entries, these four conditions can be checked using simple bitwise operations (see function `can_merge`, Figure 6.4). Preventing merges between quad fragments with overlapping multi-sample coverage (rule four) obviates the need to resolve occlusions in the Merge stage and simplifies coverage, depth, and topology merging operations (Section 6.2.2). Rule four also ensures that depth and stencil buffer contents are unaffected by the addition of quad-fragment merging to the GPU pipeline (assuming the Fragment Processing shader program does not modify a fragment's depth values). Stencil-buffer algorithms and rendering of transparent surfaces are also unaffected by the presence of quad-fragment merging.

Figure 6.7 illustrates the Merge stage's behavior for a sequence of three triangles. Rasterizing these triangles produces three quad fragments that are labeled according to their order of arrival at the Merge stage. At the start of the sequence, the merge buffer is empty. When the quad fragment from triangle 1 arrives at the Merge stage, it is inserted into the empty merge buffer. Next, the quad fragment from triangle 2 is merged with the quad fragment in the buffer because triangle 2 is in the entry's adjacent-triangle mask (triangle 2 shares an edge with triangle 1). Finally, the quad fragment from triangle 3 is merged with the buffered quad fragment (triangle 3 shares

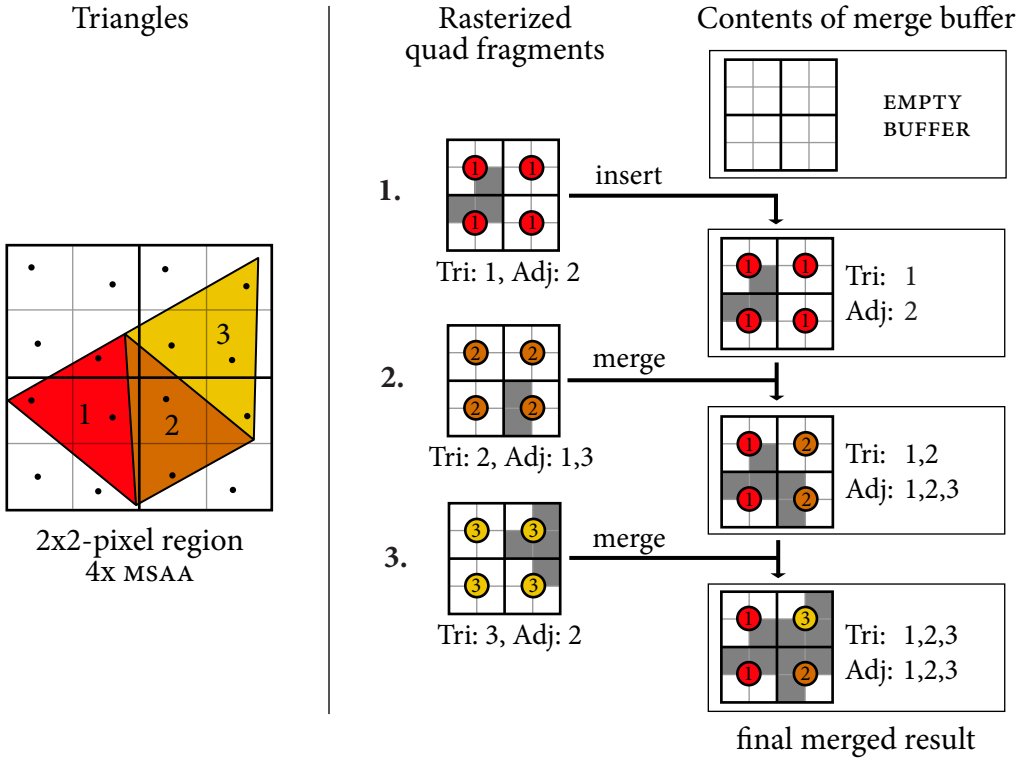


Figure 6.7: Behavior of the Merge stage on a stream of three quad fragments from adjacent grid triangles. The first arriving quad fragment (from triangle 1) is inserted into the merge buffer. The quad fragment from triangle 2 is merged with the buffered quad fragment from triangle 1 (these triangles share an edge). The quad fragment from triangle 3 is also merged with the buffered quad fragment because triangles 2 and 3 share an edge.

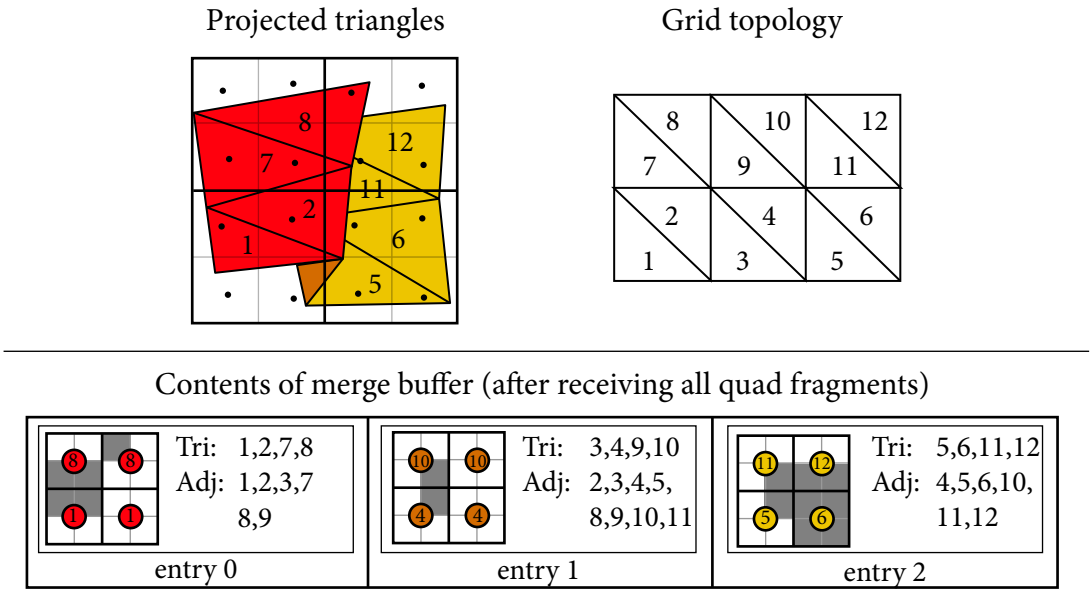


Figure 6.8: Merging does not occur across the grid's silhouette edge because triangles in the red region of the surface (1,2,7,8) do not share an edge with triangles colored in yellow (5,6,11,12). Brown-colored triangles (3,4,9,10) are back-facing so their quad fragments cannot be merged with those of the red or yellow groups.

an edge with triangle 2). In this example, shading only the merged quad fragment, rather than each rasterized quad fragment, reduces shading work by three times.

Together, the conditions above prevent merging across many types of discontinuities, such as silhouettes or folds. In Figure 6.8, quad fragments from the front-facing, red portion of the grid originate from triangles that do not share an edge with triangles in the yellow region. The quad fragments from these two groups of triangles are not merged (rule three). Nor are these two quad fragments, which originate from front-facing triangles, merged with the quad fragment from the grid's back-facing triangles (rule two). Three quad fragments are shaded for this 2×2-pixel region. For the top-right pixel, the pipeline's multi-sample buffer will contain red and yellow shading results from triangles 8 and 12. Filtering the multi-sample buffer's contents after rendering is complete produces an anti-aliased edge. Additionally, by shading the three quad fragments separately, shader derivatives are representative of actual screen-space derivatives for each group of triangles.

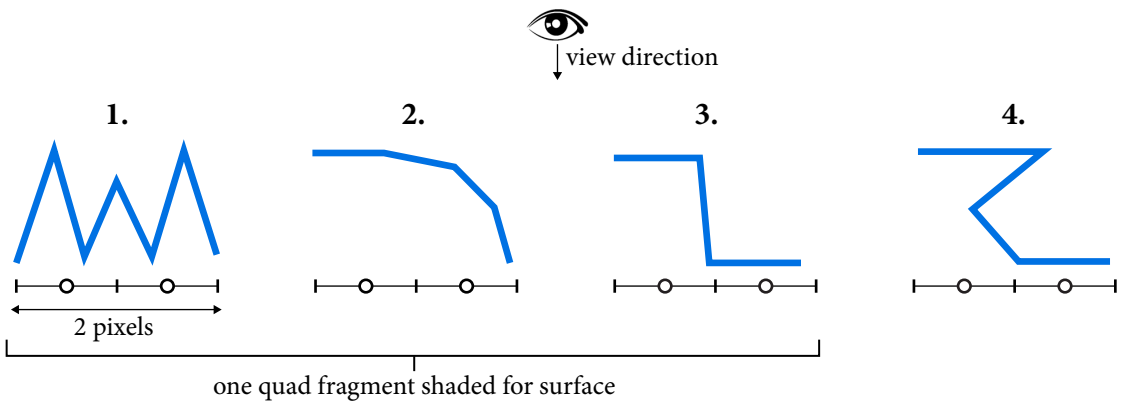


Figure 6.9: Four examples of surfaces with high curvature (only one screen dimension is shown, so the surfaces appear as lines). Quad-fragment merging will produce one fragment for all triangles from surfaces 1-3 because the triangles are connected, non-occluding, front-facing, and fall within the same 2×2-pixel region. Sampling shading only once per pixel for these surfaces may result in aliasing.

In many cases, application of the four merging rules produces a single quad fragment for each grid that overlaps a 2×2-pixel region of the screen. This was the case in Figure 6.7 and is true for the first three surfaces shown in Figure 6.9. (The illustration shows only one screen spatial dimension, so surfaces are represented by lines; each surface spans two pixels.) The fourth surface in Figure 6.9 folds over on itself. Like the surface in Figure 6.8, it results in multiple shaded quad fragments.

Surfaces 1 through 3 have high curvature, therefore shading once per pixel will undersample the surface and may cause aliasing. In contrast to current GPUs, which perform a significant amount of extra shading in this case, quad-fragment merging limits shading costs and requires geometry to be properly authored for the one-shade-per-pixel requirement to avoid aliasing. Pre-filtering high-frequency geometry (surface 1) or aligning base patch boundaries at large surface discontinuities (surface 3) are possible ways to limit aliasing.

6.2.4 Optimizations

Three key optimizations to the quad-fragment merging implementation described in the previous sections are required for the Merge stage to successfully identify a large fraction of possible merges but operate using only a small amount of buffering.

1. Generate quad fragments that do not contribute coverage: In Figure 6.10-left, three adjacent, front-facing triangles fall within the same 2×2-pixel region. However, the logic described in Section 6.2.3 will not merge the two rasterized quad fragments from these triangles. Triangle 2 does not cover a multi-sample point, so its rasterization generates no quad fragments. As a result, quad fragments from triangles 1 and 3 are not merged because these triangles do not share an edge. This problem is particularly acute at low multi-sample rates, since it is more likely for small triangles to not cover any multi-sample points.

This problem is overcome by modifying the rasterizer to generate quad fragments with empty coverage masks whenever a triangle overlaps a 2×2-pixel region, but does not cover a multi-sample point. In practice, determining if any triangle vertex lies within the pixel is an inexpensive and sufficient overlap check. In Figure 6.10-left, the Merge stage merges the empty quad fragment from triangle 2 with the quad fragment from triangle 1 (this merge only updates the topology masks for the entry). Later, the Merge stage merges the quad fragment from triangle 3 with this result. Quad fragments with empty coverage masks are never emitted by the Merge stage so they do not introduce extra shading. Averaged over test scenes presented in Section 6.3, the empty quad-fragment optimization increases the number of merges performed by the Merge stage 1.2 (16× MSAA) to 1.8 (4× MSAA) times.

2. Attempt to merge evicted quad fragments: Figure 6.10-right presents another example where it is desirable to merge all rasterized quad fragments from a grid, but merging behavior described in Section 6.2.3 does not. In this example, the order triangles arrive at the Merge stage (triangle number indicates arrival order) results in missed merges. Triangle 5 does not share an edge with triangles 1 through 4, so upon arrival at the Merge stage, its rasterized quad fragment is not merged with existing buffer entries (the quad fragment from triangle 6 has not yet been produced).

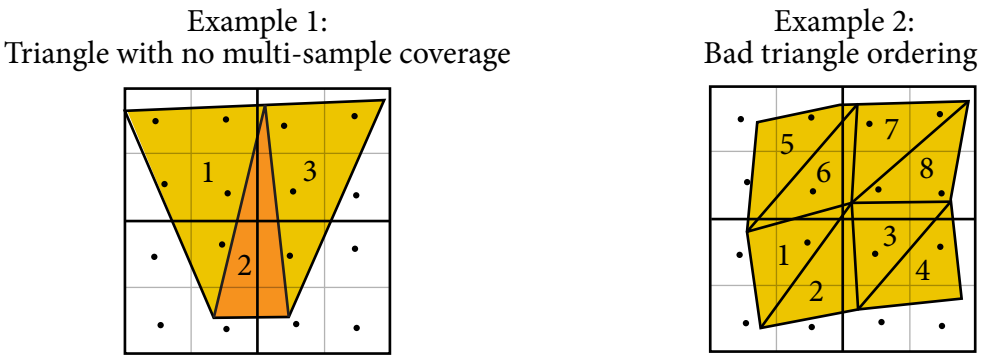


Figure 6.10: Left: The pipeline rasterizer emits a quad fragment with an empty coverage mask when a triangle overlaps a 2×2-pixel region but does not cover a multi-sample point (triangle 2). The "empty" quad fragment causes the Merge stage to update its adjacent triangle masks, leading to an eventual merge of quad fragments from triangles 1 and 3. Right: The quad fragment from triangle 5 cannot merge with the quad fragment from 1,2,3, and 4 until after the arrival of the quad fragment from triangle 6. Attempting merges prior to evicting quad fragments from the merge buffer increases Merge stage robustness to sub-optimal triangle ordering.

To avoid missing merges due to triangle ordering, when a quad fragment is chosen for eviction from the merge buffer, the Merge stage attempts to merge the eviction candidate with existing merge-buffer entries. If the quad fragment cannot be merged, it is evicted and submitted for shading. In the case of Figure 6.10-right, after processing all eight triangles, the merge buffer will contain separate entries for the merged quad fragment from triangles 1 through 4 and that of triangles 5 through 8. Attempting a final merge prior to evicting either of these quad fragments will combine them into a single quad fragment. The effect of the merge-on-evict optimization is substantial. On average it increases the number of merges by 1.8 (16× MSAA) to 2.5 (4× MSAA) times.

3. Immediately evict fully covered quad fragments: When a merge operation yields a fully covered quad fragment (all multi-sample points in the resulting quad fragment are covered), no further merges are possible with this buffer entry. Fully covered quad fragments are immediately evicted from the merge buffer.

6.3 Evaluation

This section evaluates the performance and image quality of quad-fragment merging using two software rendering pipelines. The first pipeline, NOMERGE, mimics the behavior of a current GPU by shading quad fragments from each triangle independently. The second, MERGE, also shades quad fragments, but implements quad-fragment merging as described in Section 6.2. In both pipelines DiagSplit tessellation produces grids containing at most 512 triangles (recall the merge buffer entry's topology bitmasks are 512 bits). Both pipelines also occlusion cull quad fragments prior to shading.

The two pipelines were used to render the eight scenes shown in Figure 6.11. PLANE is a basic test of merging behavior. SINEWAVE's camera position is chosen to create many grazing triangles and serves as a quality test for MERGE. BUMPYFROG (which features high-frequency displacement) and BIGGUY are standalone characters. ZINKIA and ARMY provide full scenes. POINTCLOUD and FURBALL exhibit fine-scale geometry and complex occlusion. The occlusion properties of these scenes will be important when comparing quad-fragment merging to Reyes-style vertex shading in Section 6.4.2. All scenes were rendered at 1728×1080 resolution.

6.3.1 Performance

Shaded Quad Fragment Counts

The red line in Figure 6.12 plots the average number of fragments shaded by NOMERGE at each screen pixel when rendering BIGGUY using 16× multi-sampling (pixels not covered by geometry do not factor into this average). When BIGGUY is tessellated into 0.5-pixel-area triangle micropolygons, NOMERGE shades covered pixels nearly 14 times. Recall, only one shaded fragment per pixel is the desired sampling rate. A notable amount of extra shading occurs even when triangles are large enough to cover a few pixels. Ten-pixel-area triangles still result in nearly four shaded fragments per pixel.

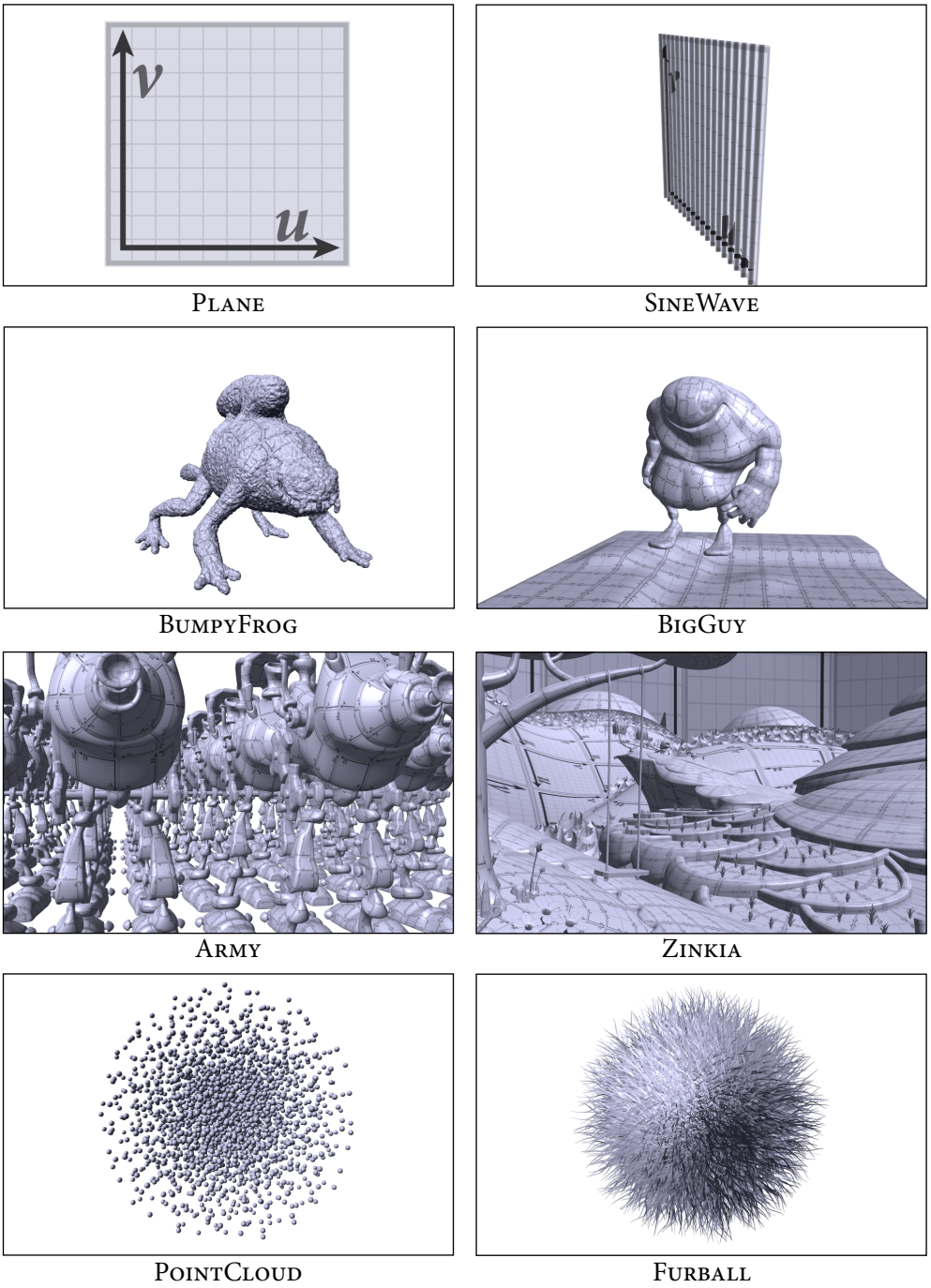


Figure 6.11: Test scenes featuring high-frequency geometry (SineWave, BumpyFrog), complex occlusion (PointCloud, Furball), grazing triangles (SineWave), characters (BigGuy, Army), and environments (Zinkia).

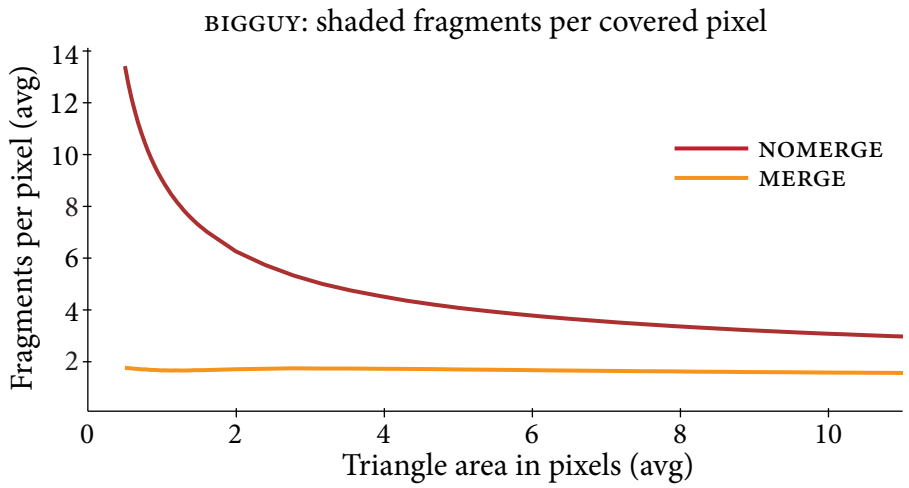


Figure 6.12: When BIGGUY is tessellated into 0.5-pixel-area triangles, NOMERGE generates nearly 14 fragments per covered screen pixel. The MERGE pipeline shades only 1.8 fragments per pixel. Even when triangles are ten pixels in area, MERGE provides approximately a 2× reduction in shading work.

The MERGE pipeline (orange line) reduces the number of shaded fragments substantially. In many cases, all quad fragments at the same screen location are merged, so the amount of shading is independent of the size of scene triangles. Although quad-fragment merging was designed to enable efficient shading for micropolygons, it provides a significant reduction in shading work even when rendering small (but not necessarily sub-pixel) triangles.

On average, MERGE shades covered pixels approximately 1.8 times. This number falls short of the ideal one-fragment-per-pixel rate for three reasons: merging does not occur across grid boundaries, early occlusion culling in the pipeline is not perfect (regions of objects are shaded but ultimately occluded), and multiple fragments must be shaded in pixels containing object silhouettes. The images in Figure 6.13, which visualize the number of fragments shaded at each pixel, show that MERGE indeed shades many image pixels exactly once (DIAGSPLIT is configured to target 0.5-pixel-area micropolygons in these renderings; actual average triangle area varies from 0.37 to 0.50 pixels across the scenes). In these images dark blue pixels are shaded once, bright green pixels four times, and dark red pixels at least eight. A majority of pixels

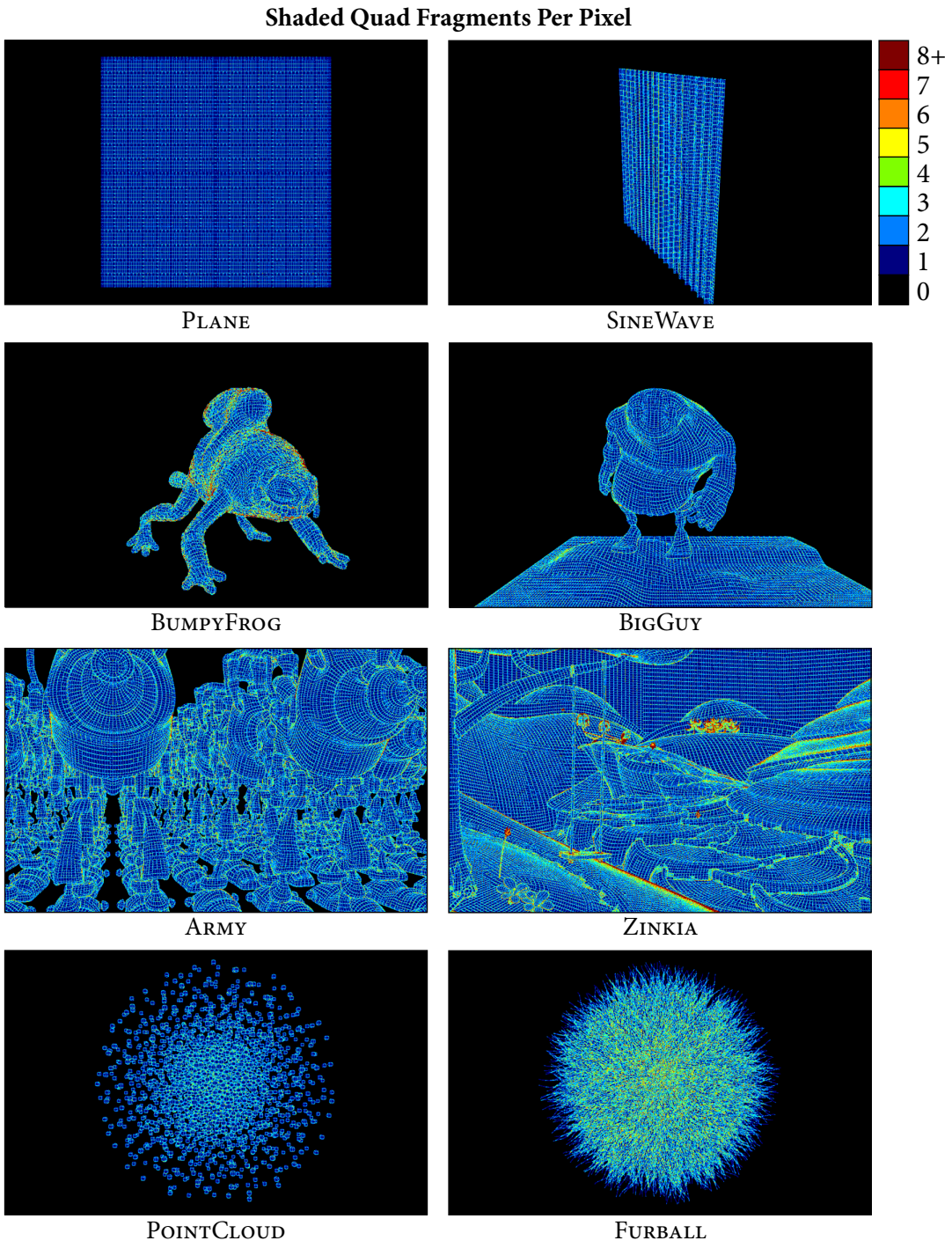


Figure 6.13: Shaded fragments per pixel produced by MERGE (images are colored according to the number of fragments shaded per pixel). Most scenes exhibit large regions of dark blue, indicating one shaded fragment per pixel.

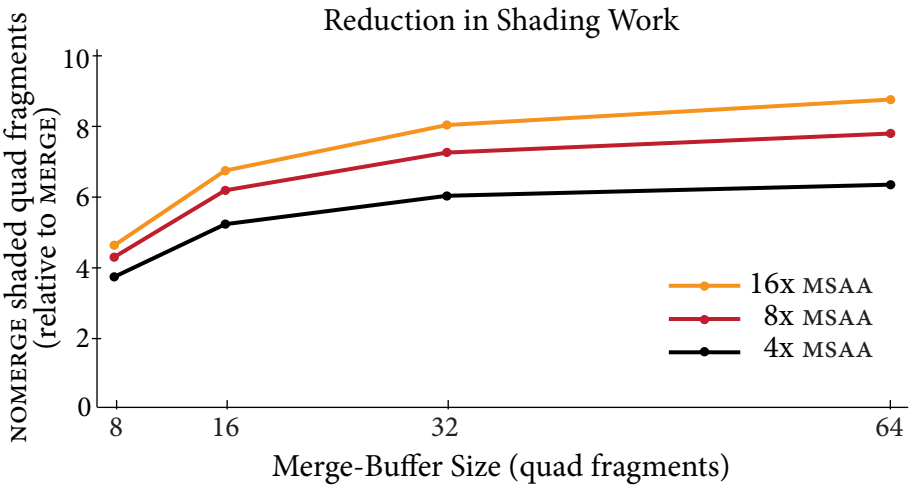


Figure 6.14: Quad-fragment merging requires only a small amount of buffering to capture a high percentage of possible merges. A 32-quad-fragment buffer captures 91% of the merges captured by a buffer of unbounded size.

in these images are dark blue. Pixels near grid boundaries are shaded more than once because merging does not occur across grids. Extra shading at grid boundaries creates the grid-like structure apparent in the visualizations. Shading also increases near object silhouettes because the screen-projection of grids becomes long and skinny (the area-to-perimeter ratio of these grids is small).

Comparison with Nomerge

Figure 6.14 plots the benefit of quad-fragment merging as the size of the merge buffer is changed. Higher values on this graph indicate greater reduction in shading work (in comparison to NOMERGE). On average, a 32-quad-fragment merge buffer reduces the number of shaded fragments 8.1 times (16× multi-sampling). A merge buffer of this size captures 91% of the merges found by an "ideal" buffer of unbounded size. The benefits of merging decrease when multi-sampling is low. This result is not due to any change in the behavior of MERGE: the number of quad fragments generated by NOMERGE decreases at low multi-sampling because triangles are less likely to cover multi-sample points (NOMERGE is less inefficient).

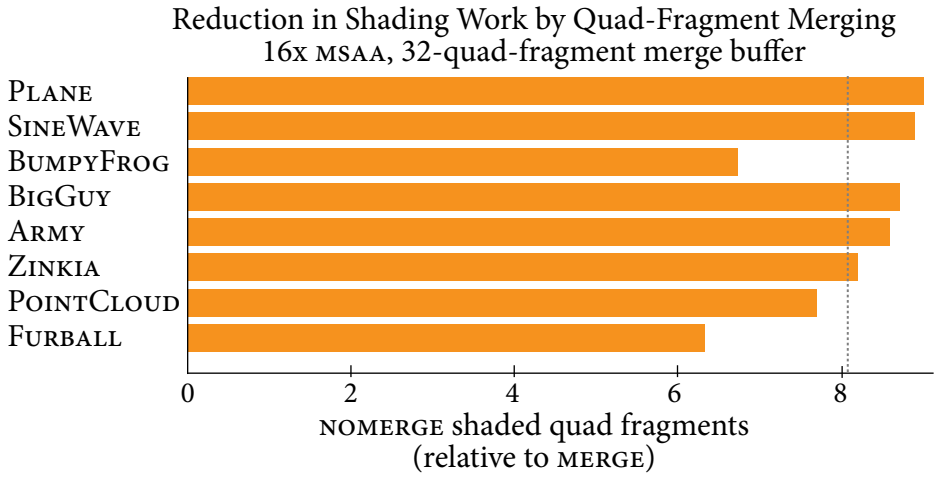


Figure 6.15: On average, MERGE performs 8.1 times less shading work than NOMERGE(average indicated by vertical dotted line). Scenes where the benefit of MERGE is less, such as BUMPYFROG and FURBALL provide fewer opportunities for the pipeline to perform valid merges.

A 32-quad-fragment merge buffer constitutes only a small increase in the current storage requirements for a modern GPU shader core. High-end GPU processing cores [NVI 2009] simultaneously shade more than 256 quad fragments and must already store shading inputs and shader intermediate values for these quad fragments. For the remainder of this evaluation, MERGE is configured to use a 32-quad-fragment merge buffer and render images using 16× multi-sampling.

Figure 6.15 illustrates the benefit of MERGE on a scene-by-scene basis. The average reduction in shading across all scenes (8.1×) is shown as a vertical dotted line. The benefit of MERGE is the least for BUMPYFROG and FURBALL because these scenes exhibit characteristics that limit opportunities for merging. BUMPYFROG's high-frequency surface displacement creates many folds and silhouettes. As described in Section 6.2.3, extra shading is required for these features. All of FURBALL's grids are long and skinny (they represent hairs). Their poor area-to-perimeter ratio yields fewer merging opportunities.

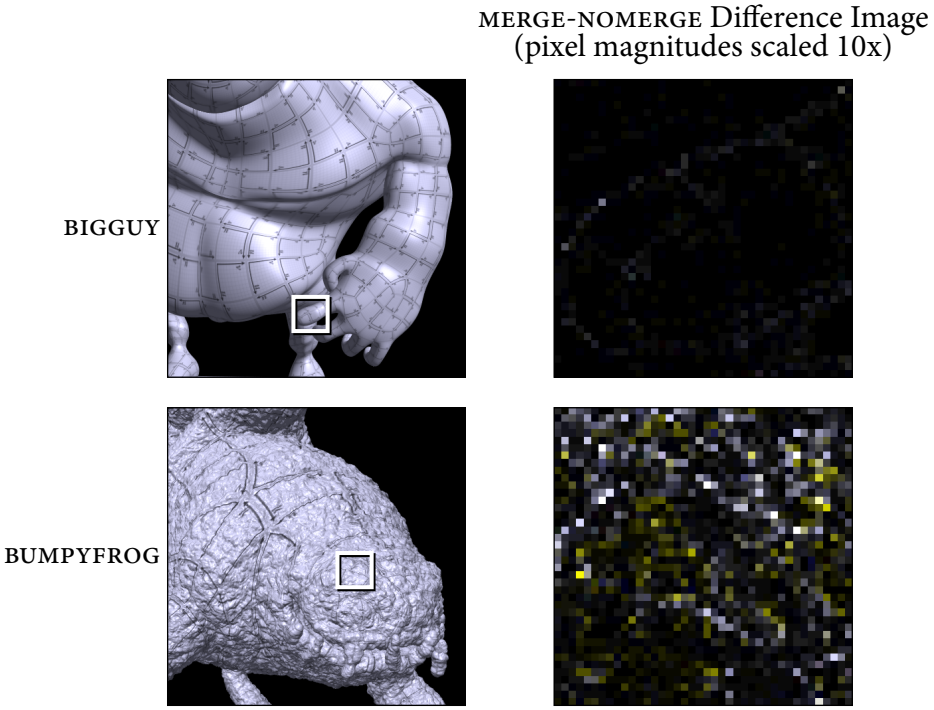


Figure 6.16: Difference images comparing MERGE and NOMERGE output (right column). For clarity, pixel intensities have been magnified ten times. Image output when rendering BIGGUY is nearly identical. MERGE's sampling of BUMPYFROG's detailed surface does result in many pixels with numerically different values, but these differences are difficult to notice when visually comparing rendered output. The most notable differences between MERGE and NOMERGE output occur in areas where the surface is not flat in a 2×2-pixel region, such as along surface silhouettes.

6.3.2 Visual Quality

We have visually inspected the quality of many rendered animations and observed that MERGE commonly generates high-quality output that is comparable to that of NOMERGE. For example, the zoomed difference image in Figure 6.16-top indicates that many pixels in the two pipeline's renderings of BIGGUY are exactly the same (pixel intensities in the difference image have been magnified ten times). Although the quality of images produced by MERGE is high, the two pipelines do produce different images. As expected, we have observed increased aliasing when high-frequency bumpy surfaces are undersampled by MERGE (both in a single frame, see Figure 6.16-bottom, as well as temporal aliasing across frames). In addition to undersampling, two additional sources of artifacts in images created by MERGE require mention.

Attribute Extrapolation Errors

Quad-fragment merging can exacerbate artifacts caused by extrapolating triangle attributes to a shading sample points outside the triangle. Figure 6.17 highlights the contents of the multi-sample frame buffer (top row) for one pixel of a rendering of BUMPYFROG. In this example, BUMPYFROG is rendered using a phong shader with a blue ambient term and a white specular highlight. The pixel boundary is shown as a white square. The bottom row of the figure shows a portion of the final image surrounding this pixel. No triangle covers the pixel center, but the multi-sample point closest to the pixel center is covered by a nearly edge-on triangle. For this triangle, shading at the pixel center produces an inaccurate, bright white result. In NOMERGE, only one covered multi-sample is assigned this color, resulting in a subpixel error in the final image (bottom-left). By contrast, MERGE uses the erroneous shading result for all covered multi-samples in the pixel, producing a noticeable bright spot in the final image (bottom-center). If extrapolation errors impact quad-fragment derivatives, artifacts may spread to neighboring pixels as well. Centroid sampling avoids these extrapolation errors, removing this artifact from both the multi-sample results (top-right) and the final image (bottom-right).

Although centroid sampling is not commonly used in GPUs today, it is a useful

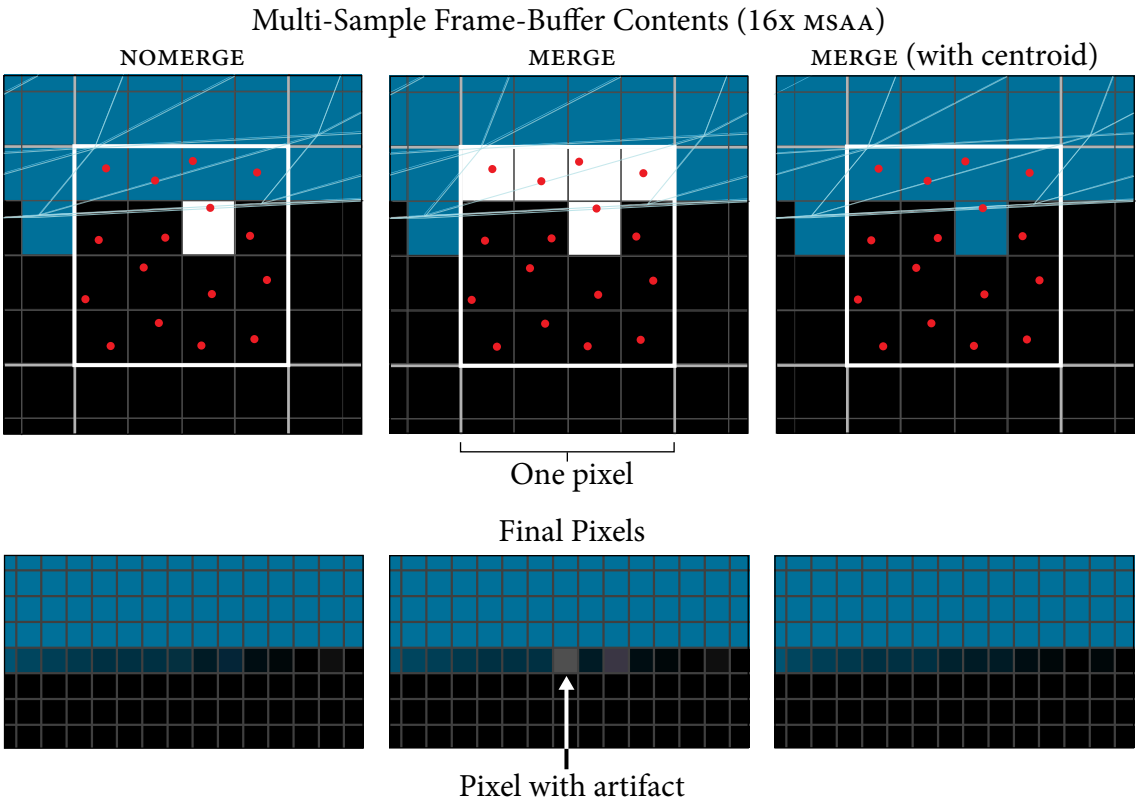


Figure 6.17: Sampling shading outside a grazing triangle can produce artifacts in both NOMERGE (left column) and MERGE (center column). Shading artifacts from the grazing triangle are more noticeable in MERGE because they are applied to all multi-sample points in the pixel, not just the points covered by the triangle. Centroid sampling avoids attribute extrapolation and is used to correct these artifacts in the MERGE pipeline (right column).

technique for the MERGE pipeline. To obtain accurate derivative estimates when using centroid sampling, it is important to modify finite-difference calculations in Fragment Processing to account for the actual positions of shading sample points. It is more expensive to accurately compute derivatives when shading sample points are not distributed uniformly on screen, so current GPU implementations assume shading sample points are located at pixel centers when estimating derivatives [Mic 2010a], regardless of whether centroid sampling is enabled. This implementation choice results in inaccurate derivative estimates when centroid sampling is used on current GPUs.

Alternative mechanisms for mitigating artifacts caused by extrapolating shading inputs from grazing triangles were considered during this work, but not explored. For example, it may be possible to minimize extrapolation errors by modifying quad-fragment merging's shading input selection rules to draw shading inputs from the triangle covering the most multi-sample points in a pixel (although this scheme risks increasing temporal flicker). Future work should also explore augmenting the merging conditions to prevent merges with grazing triangles.

Derivative Errors Near Silhouettes

Given the merging conditions discussed in Section 6.2.3, the Merge stage can emit quad fragments corresponding to surface regions with high curvature (recall surfaces 1-3 in Figure 6.9). In these situations, finite difference computations during Fragment Processing, which assume the corresponding surface is bilinear within a 2x2-pixel region, may be inaccurate. Our experience indicates that derivative errors on bumpy surfaces (e.g., Figure 6.9, surface 1) are largely imperceptible (errors are masked by the detail in the surface), but it is possible to observe effects of derivative errors on smooth surfaces that have rapidly varying screen-space derivatives (Figure 6.9, surface 2). This behavior most commonly exists near silhouettes.

Figure 6.18 visualizes the mip-map level used to sample texture data for pixels near the BIGGUY character's silhouette (map-map level is used as a proxy for visualizing surface derivatives). The top images in the figure show the contents of the multi-sample frame buffer after rendering is complete (16× multi-sampling). The bottom

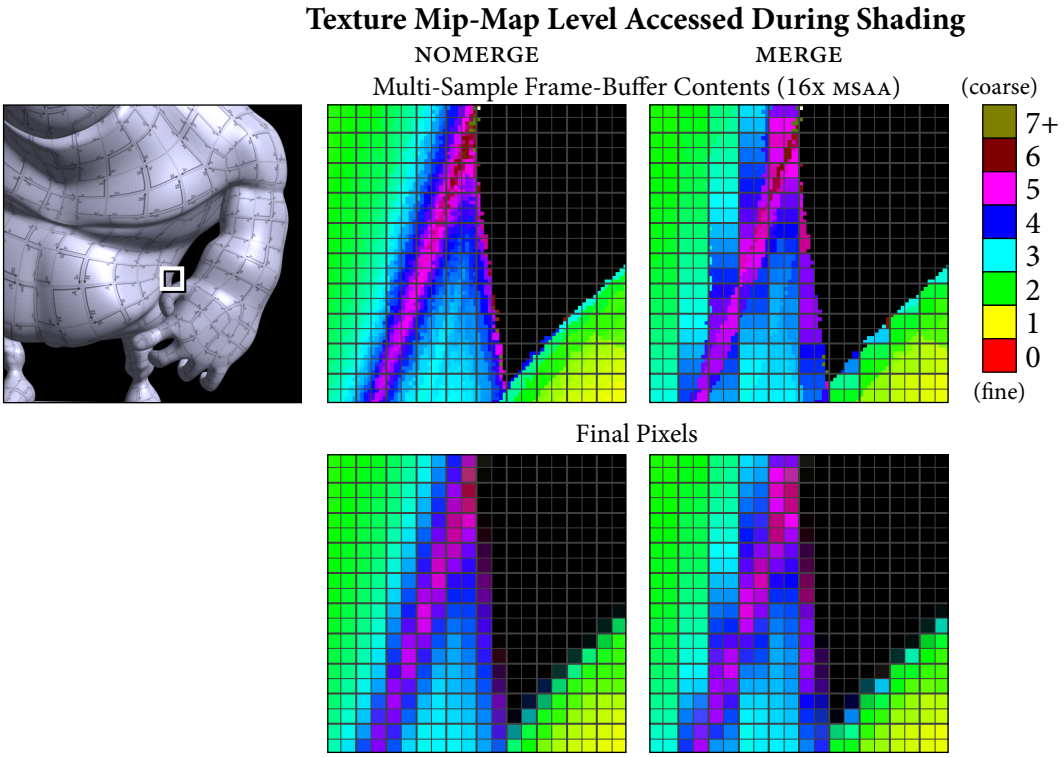


Figure 6.18: Visualization of the texture mip-map level accessed when shading BIG-GUY. MERGE (right column) estimates derivatives less accurately than NOMERGE (left column) in 2x2-pixel regions where the surface exhibits high curvature, such as object silhouettes (this is particularly clear when inspecting the output of the multi-sample frame buffer (top row). The 2x2-granularity of quad-fragment derivatives is clearly visible in the images generated by the MERGE pipeline.

images show final pixel values after filtering the multi-sample frame buffer. The results of the MERGE pipeline clearly show the 2×2-pixel granularity of quad-fragment derivatives. Shading computations from pixels containing the silhouette access too low a mip-map level (risking texture aliasing). Pixels adjacent to the pixels containing the silhouette access too high of a mip-level (risking over-blurring). Errors in the mip-map level accessed during shading can result in visible differences in rendered output when high-frequency texture data is used. Notice that unlike the extrapolation error example in Figure 6.17, which amplified a subpixel artifact into a pixel-sized artifact, derivative errors in the MERGE pipeline impact rendered output for an entire 2×2 block of pixels.

Fortunately, we've found visual artifacts due to inaccurate derivatives to be rare and difficult to observe under most rendering conditions. Even so, quad-fragment merging improvements, such as preventing merges when quad-fragments originate from triangles with widely varying normals, or, as suggested previously, preventing merges with sufficiently grazing triangles, should be explored.

6.4 Quad-Fragment Merging Alternatives

Quad-fragment merging is an attractive solution to the problem of redundant shading when rendering micropolygons. It achieves a shading density of nearly one sample per pixel and, by building upon the quad-fragment representation, it also supports important GPU optimizations like efficient derivatives, fine-granularity occlusion culling, and multi-sample anti-aliasing. However, there are two popular, alternative shading strategies that also offer the promise of shading micropolygon meshes approximately once per covered pixel. The first is deferred shading, which is implementable on current GPUs and used by many games today. The second is Reyes-style per-vertex shading. This section briefly contrasts quad-fragment merging with the merits of these two alternative shading approaches.

6.4.1 Deferred Shading

Rather than shade fragments immediately after rasterization, "deferred shading" postpones shading computations until all scene geometry has been rasterized and all occlusions resolved at the frame buffer [Deering et al. 1988]. The initial geometry processing phase of deferred shading populates a "deep frame buffer" (often called a geometry buffer, or G-buffer) that stores attributes sampled from scene geometry visible to each screen sample point. The contents of the G-buffer are used as inputs to shading computations performed in subsequent rendering passes.

Deferred shading offers the possibility of shading an entire scene (not just entire triangles or surfaces) exactly once per pixel regardless of geometric detail or scene depth complexity. However, it has traditionally been avoided by GPU architects as a core pipeline mechanism because it interacts badly with multi-sample anti-aliasing. To support anti-aliasing, previous hardware implementations of deferred shading stored the G-buffer at multi-sample resolution and computed shading once per multi-sample point, rather than once per pixel [Molnar et al. 1992].

Today, many game engines implement deferred shading as a software layer running on GPUs, but most do so by disabling multi-sample anti-aliasing and accepting the resulting loss in image quality. (Akenine-Möller et al. [2008] and Lauritzen [2010] provide excellent overviews of modern deferred shading implementations in games.) The addition of multi-sample frame-buffer access in Direct3D 10.1 makes it possible to augment software deferred shading implementations with a limited form of multi-sample anti-aliasing, however these approaches must compute and store shading inputs at all multi-sample points, then analyze G-buffer contents to determine if one or multiple shading computations are required at each pixel. Heuristics for detecting scene discontinuities or object silhouettes based only on G-buffer contents are prone to error [Lauritzen 2010]. Similar challenges arise if shading computations require finite-difference derivatives. Quad-fragment merging overcomes these problems by propagating mesh connectivity through the pipeline and using this information to robustly make merging decisions (it does not have to reconstruct surface connectivity from a point-sampled representation of geometry).

Despite its drawbacks, use of deferred shading is becoming increasingly popular

in games because it scales well to scenes with large numbers of lights. When the rendering pipeline shades quad fragments immediately after rasterization (sometimes referred to as "forward rendering"), illuminating surfaces with many lights has significant storage cost because shadow maps for all lights must be available and bound to the pipeline during rendering. Complex lighting also introduces conditional control-flow in shader programs because most surfaces receive illumination from only a small subset of scene lights. A common solution to these problems is to partition shading work into multiple rendering passes (e.g., one for each light source). Unfortunately, multi-pass rendering is cost prohibitive in a system intended for micropolygon rendering because the overhead of processing scene geometry each pass is large.

In contrast, deferred shading caches the results of geometry processing in the G-buffer, so multi-pass rendering only incurs the bandwidth overhead of loading G-buffer data from memory in each rendering pass. However, compared to single-pass forward rendering, the bandwidth requirements of deferred shading, especially when combined with multi-sample anti-aliasing techniques described above, are large. Given that memory bandwidth is a scarce resource on modern compute-rich GPUs, ongoing work seeks to reduce the bandwidth costs of deferred shading [Andersson 2009].

Moving forward, it will be interesting to see whether the G-buffer storage and bandwidth costs of deferred shading, or the shadow map storage and conditional control-flow costs of shading in a forward rendering pipeline with quad-fragment merging, limit performance scaling on future GPUs. Given the wide variation in lighting and shading requirements across games, it is likely that developers will continue to find use for both techniques.

6.4.2 Reyes Shading

The Reyes pipeline samples surface shading once per micropolygon grid vertex, rather than once per fragment (readers may wish to review the summary of the Reyes pipeline architecture in Section 2.2). This approach has many advantages. For example:

- Sampling shading at points on the surface, rather than pre-determined screen-space locations, avoids shading errors caused by sampling outside triangles.

Further, no heuristics (e.g., merging rules) are needed to determine surface continuity between two adjacent shading sample points. However, the accuracy of heuristics used to set tessellation factors become very important in Reyes, as undertessellation and overtessellation directly impact the quality and amount of shading.

- Shading density can be adjusted on a per-object basis by modifying tessellation amounts. Per-vertex shading achieves a shading density of approximately one sample per pixel when surface tessellations contain about one vertex per pixel.

- The idea of shading prior to rasterization, then interpolating shading results to determine surface color at covered multi-sample points, elegantly extends to motion and defocus-blurred rasterization. Implementing 5D rasterization techniques is more challenging in a pipeline that shades quad fragments; it remains an active research problem and has not been implemented in commercial GPUs.

However, when compared to a pipeline that implements quad-fragment merging, Reyes-style shading presents a number of unique challenges. Two notable examples include:

- Reyes requires surfaces to be tessellated into micropolygons for high-quality shading, even if micropolygons are not necessary to adequately capture surface detail. As a result, adopting Reyes-style shading for real-time rendering requires an immediate transition to micropolygon-resolution geometry. In contrast, quad-fragment merging is a viable shading solution for both micropolygons and larger triangles. Quad-fragment merging permits application developers to adjust geometric complexity of a scene based on performance-quality needs while Reyes does not.

- Since Reyes performs shading computations prior to rasterization, it is prone to shading surface regions that are not visible to the camera (e.g., off-screen or occluded surfaces).

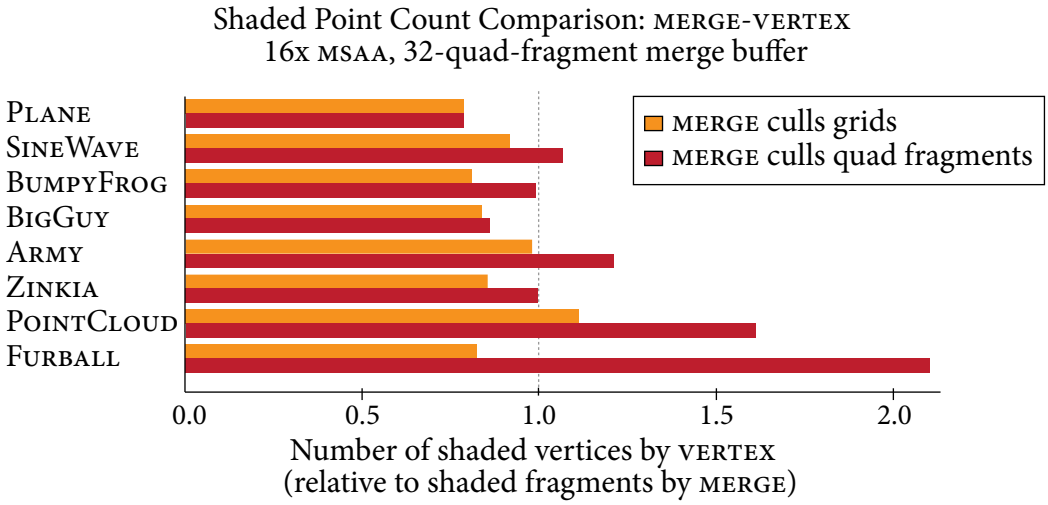


Figure 6.19: MERGE shades approximately as many fragments as Reyes (VERTEX) shades vertices. When fine-scale occlusion is present (POINTCLOUD, FURBALL), MERGE shades over two times less than Reyes because it culls individual quad fragments prior to shading.

We have not studied the image-quality implications of sampling surface shading at mesh vertices rather than uniformly on screen at pixel centers. However, we have compared the amount of shading performed by a Reyes pipeline with that of a pipeline implementing quad-fragment merging.

To conduct this study, we modified the NOMERGE pipeline from Section 6.3 to perform all shading computations at each grid vertex prior to rasterization (this pipeline is hereafter referred to as VERTEX). Like popular Reyes implementations [Cook et al. 1987; Apodaca and Gritz 2000], VERTEX shades at a granularity of entire grids. Shading grid vertices en masse enables data-parallel execution and allows derivatives to be computed by differencing shading quantities from adjacent vertices (grid topology is maintained in the pipeline to locate adjacent vertices). Grids also serve as the granularity of occlusion culling in VERTEX: either an entire grid is discarded prior to shading, or all vertices in the grid are shaded. Thus there is tension between the need to make grid sizes large (to increase the data-parallel efficiency of shading computations and to reduce redundant shading at grid boundaries) and the desire to keep grids small for culling (to eliminate unnecessary shading of occluded surfaces).

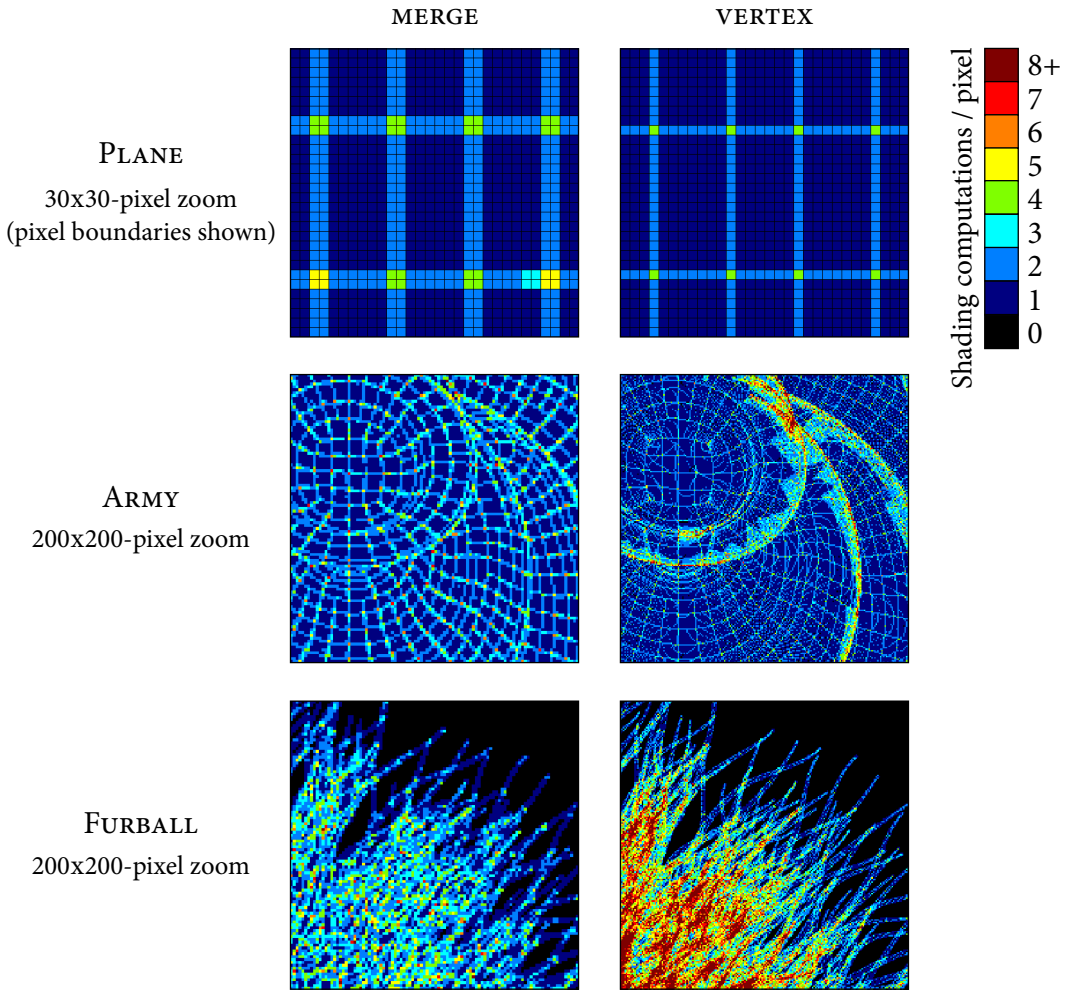


Figure 6.20: Comparison of the number of shading computations performed by the MERGE and VERTEX pipelines:

Top: Due to the 2×2-pixel granularity of quad-fragment shading, the overhead of extra shading at grid boundaries (light-blue and green regions) is greater in the MERGE pipeline than in VERTEX. Unlike MERGE, VERTEX only performs redundant shading in pixels containing grid boundary vertices.

Middle and bottom: MERGE occlusion culls individual quad fragments prior to shading, while VERTEX culls at the granularity of grids. As a result, VERTEX shades more surface regions that are not visible to the camera. This is particularly severe in the FURBALL example, where complicated occlusions prevent many grids from being culled (MERGE shades more than two times less than VERTEX in this case).

Figure 6.19 compares the number of fragments shaded by MERGE with the number of vertices shaded by VERTEX when both pipelines are configured to generate grids containing approximately 0.5-pixel-area triangle micropolygons (recall from Section 6.3.1 that actual average micropolygon area varies in the test scenes from 0.37 to 0.50 pixels). In accordance with the requirements of MERGE, maximum grid size produced by tessellation is limited to 512 micropolygons. The graph plots the ratio of the two pipeline's shaded point counts, so values greater than one indicate that MERGE shades fewer fragments than VERTEX shades vertices. On average, when both pipelines only occlusion cull entire grids (that is, when per-quad-fragment occlusion culling is disabled in MERGE), MERGE shades 12% more than VERTEX (orange bars).

Figure 6.20-top explains this difference using shaded-point-count visualizations for a zoomed view of the PLANE scene. The visualization for VERTEX is produced by tracking the number of shaded vertices falling within each pixel. Both VERTEX and MERGE perform extra shading at pixels near grid boundaries, which appear light blue in the images. In VERTEX, extra shading occurs because adjacent grids both contain a vertex at these pixels. MERGE also generates multiple quad fragments near grid boundaries because quad fragments from different grids are not merged. Since MERGE performs extra shading in entire 2×2-pixel regions, instead of only in pixels containing boundary vertices, extra shading near boundaries is more pronounced.

However, when MERGE occlusion culls individual quad fragments prior to shading (a common optimization in modern GPUs) on average it shades 17% less than VERTEX (red bars). The benefit of fine-granularity culling is particularly large in scenes such as FURBALL and POINTCLOUD that exhibit fine-scale, overlapping geometry. For example, VERTEX performs more than two times as many shading computations as MERGE when rendering FURBALL (see Figure 6.20-bottom). Culling shading work at grid granularity can be inefficient even if fine-scale geometry is not present. The center-right image in Figure 6.20 shows that VERTEX shades vertices belonging to back-facing micropolygons when grids wrap around object silhouette edges.

6.5 Discussion

This chapter showed that when surfaces are tessellated into micropolygons, the GPU pipeline performs a substantial amount of redundant shading. Because future high-quality rendering will require both high-resolution meshes and expensive shaders, this inefficiency is alarming. Adding quad-fragment merging to the GPU pipeline eliminates most of this redundant shading. Quad-fragment merging reduces the number of shaded quad fragments by over a factor of eight, preserves high image quality, and requires only modest extensions to the GPU pipeline's current shading mechanisms.

The design of quad-fragment merging places heavy emphasis on enabling highly optimized implementations. Merging operations are cheap (both checking merging conditions and performing merges require only bitwise operations) and the buffering requirements of merging are low. These properties should map well to efficient, fixed-function hardware implementation. The GPU pipeline architecture extensions proposed here encapsulate quad-fragment merging logic in a separate Merge stage (preserving the data-parallel programming model of Fragment Processing). This design provides future optimized GPU implementations the opportunity to encapsulate Merge-stage functionality in fixed-function components (preserving the data-parallel optimized design of a GPU's programmable cores).

A key idea of quad-fragment merging is the use of micropolygon connectivity when making merging decisions. Connectivity information allows quad-fragment merging to robustly piece back together the surface from independent rasterized quad fragments. As a result, quad-fragment merging need not infer surface connectivity like point-based rendering approaches. Although edge-adjacency and the other three merging rules presented in this chapter provide a good balance between reducing shading cost and avoiding shading artifacts, these rules are by no means definitive. For example, considering surface normal during merging may reduce the shading artifacts discussed in Section 6.3.2. Exploring the performance-quality trade-offs associated with different merging conditions is clearly a useful future study.

Immediate future work should investigate the interaction of quad-fragment merging with motion- and defocus-blurred rasterization. Recently, Ragan-Kelley et al.

[2010] proposed an extension of the GPU pipeline that allows quad-fragments to be generated and shaded based on the results of 5D rasterization. Supporting both 5D rasterization and multi-sample anti-aliasing is challenging in a pipeline that shades quad fragments because the relationship between shading samples and the corresponding covered multi-sample points cannot be represented efficiently by a dense coverage mask. (When an object undergoes blur, multi-sample points distributed widely across the frame buffer receive contributions from the same shading sample.) Ragan-Kelley et al. overcome this problem by caching shading results, then retrieving them from the cache on-demand for each covered multi-sample point.

Although motivated by different reasons, both Ragan-Kelley's et al.'s shading cache and the quad-fragment merging merge buffer serve to identify complex relationships between multi-sample points and shading samples. Motivated by this observation, Hegarty [2010] implemented motion blur in a quad-fragment merging pipeline by leveraging the merge buffer to simultaneously identify merges and maintain the irregular mapping between shading samples and multi-sample points. (In doing so he implemented a feed-forward, rather than demand-driven, version of Ragan-Kelley's technique.) However, to efficiently perform merges in this system, Hegarty disabled the non-overlapping multi-coverage merging condition (rule four). The implications of this decision on image quality and quad-fragment merging's depth and stencil buffer invariants have not yet been rigorously analyzed. Nonetheless, the prospect of successfully performing quad-fragment merging in a pipeline that also features 5D rasterization is promising. Perhaps most importantly, the potential of using a single mechanism to implement both techniques increases the likelihood of their adoption in future GPUs.

Last, quad-fragment merging makes shading quad fragments a viable alternative to Reyes-style shading for micropolygon workloads. While current GPU implementations of quad-fragment shading have significant performance advantages over Reyes-style shading systems, it is not obvious whether, in terms of image quality, it is preferable to sample shading uniformly along the surface (object-space shading) or uniformly on screen. There is value in developing a better understanding of the image-quality implications of these two popular shading approaches.

Chapter 7

The Real-Time Micropolygon Pipeline

Improving the efficiency of micropolygon rendering requires extending existing GPU pipeline abstractions and modifying core rendering algorithms. Although for simplicity, the previous chapters addressed tessellation, rasterization, and quad-fragment shading largely in isolation, many of the GPU pipeline and rendering algorithm modifications suggested in this dissertation are highly interrelated. Figure 7.1 brings all of these changes together and illustrates the resulting real-time micropolygon rendering architecture. It also enumerates key differences between the real-time micropolygon pipeline and the GPU pipeline (red and blue lines differentiate changes to the pipeline architecture from changes that impact only its implementation). These differences, as well as the relationships between important components of the micropolygon pipeline, are summarized below.

7.1 Summary of Modifications

DiagSplit Tessellation

First, the micropolygon pipeline features a new stage that enables adaptive surface tessellation using the DiagSplit algorithm (Figure 7.1, pipeline differences 1 and 3).

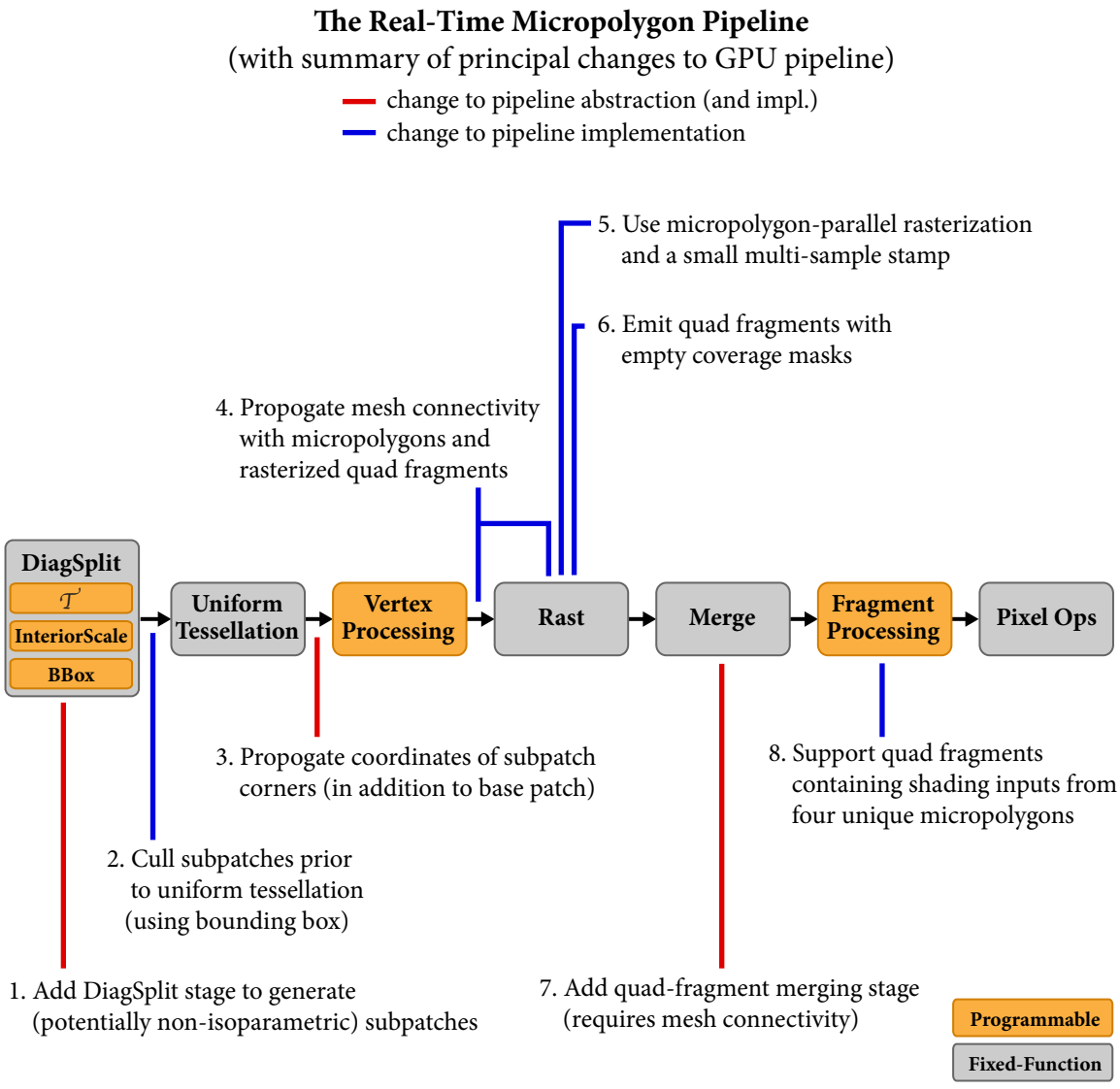


Figure 7.1: A micropolygon rendering pipeline designed for real-time rendering. The micropolygon pipeline is an evolution of the current GPU pipeline. Notable differences between the micropolygon pipeline and the current GPU pipeline are listed above. For clarity, differences in pipeline architecture are highlighted in red. Changes to pipeline implementation (that do not require significant architectural modification) are shown in blue.

Tessellation becomes the primary source of work generation in the micropolygon pipeline. As a result, DiagSplit's behavior has significant impact on nearly all subsequent processing in the system. Most notably:

- Through the use of non-isoparametric cuts, DiagSplit is able to adapt to surface complexity or view and also produce subpatches that can be processed independently and in parallel (without cracks) by the remainder of the pipeline.
- Adaptive splits are inexpensive to compute and yield accurate tessellations using fewer micropolygons than current GPU tessellation schemes. Lower micropolygon counts reduce the amount of culling, rasterization, fragment merging, and (potentially) shading work the pipeline must perform.

While DiagSplit was initially designed to achieve the two benefits listed above, it has become clear that split-dice adaptive tessellation has numerous additional benefits. For example:

- Performing splits defines an implicit hierarchy over the micropolygons generated from large base patches. Given shader programs that bound subpatches, pipeline implementations can utilize this hierarchy to efficiently cull subpatches at varying scales (the implementation in this dissertation culls occluded subpatches after each split) and avoid dicing many micropolygons that are not visible in the final image (Figure 7.1, difference 2). In addition to culling, the hierarchical structure of micropolygons generated by DiagSplit should facilitate many additional pipeline optimizations. For example, RenderMan bounds subpatches produced by split (prior to dicing) to efficiently implement sort-first (bucket-parallel) rendering [Apodaca and Gritz 2000].
- DiagSplit generates micropolygons that are approximately uniform in area, orientation, and aspect ratio. This makes simple algorithms for micropolygon rasterization more efficient by reducing variance in per-micropolygon rasterization cost. Chapter 4 showed low variance in micropolygon size results in high utilization during micropolygon-parallel rasterization.

- Tessellation (specifically, the Uniform Tessellation stage) generates micropolygon-connectivity information required by quad-fragment merging to robustly determine when two quad fragments can be merged. Diced subpatches define a reasonable window over which to search for merges, and micropolygon grids provide a convenient namespace to uniquely identify micropolygons.

- Just as the rasterizer in a current GPU controls the order fragments are generated from a triangle spanning many pixels, the DiagSplit stage controls the order subpatches are split and the size of diced grids. (Also, the Uniform Tessellation stage controls the order of micropolygons in a grid.) By adjusting parameters such as these, DiagSplit implementations can tailor their micropolygon output stream to match specific hardware performance characteristics (e.g., parallelism/scheduling needs, cache sizes) and meet invariants assumed by subsequent processing stages. For example, in Chapter 6, the efficacy of quad-fragment merging was increased by limiting the number of micropolygons in diced grids to match the size of the pipeline's merge buffer.

Reoptimized Rasterization

The second focus of this work was reoptimizing the implementation of the pipeline's rasterization stage for micropolygon workloads (Figure 7.1, difference 5). The small and uniform size of micropolygons allows for a simple bounding-box-based rasterization scheme (MPRAST) that is efficiently parallelized by processing many micropolygons at once. Experiments showed that in comparison to a rasterizer employing a 64-multi-sample stamp, this approach reduced the number of point-in-polygon tests performed during rasterization between five and eighteen times (Figure 4.4, 0.5-pixel-area micropolygons). Despite this improvement, the sheer number of operations needed to rasterize micropolygon geometry is high. The high cost and algorithmic simplicity of micropolygon rasterization make acceleration via custom hardware an attractive implementation choice. Although summarized only briefly in this dissertation, subsequent work indicates that significant power and area savings result from the use of custom hardware for micropolygon rasterization [Brunhaver et al. 2010].

In addition to the modifications made to improve performance, the micropolygon pipeline rasterizer also underwent two minor modifications to support quad-fragment merging (differences 4 and 6). First, the rasterizer propagates micropolygon-adjacency information with quad fragments emitted to the Merge stage. Second, to provide the Merge stage with additional information about grid topology, the rasterizer generates a quad fragment with empty multi-sample coverage when a micropolygon overlaps a 2×2-pixel region of the screen but does not cover any multi-sample points.

Quad-Fragment Merging

Third, the micropolygon pipeline includes a new fixed-function stage that implements quad-fragment merging (Figure 7.1, differences 4, 7, and 8). The key idea of quad-fragment merging is to propagate micropolygon-connectivity information (generated by Uniform Tessellation) through the pipeline and use it to identify when quad fragments from adjacent mesh micropolygons can be safely merged into a single quad fragment prior to shading. The benefit of this optimization is substantial; it reduces the number of quad fragments shaded by the pipeline by more than eight times. As a result of quad-fragment merging, when rendering equivalent scenes, the micropolygon rendering pipeline performs approximately as many shading computations as Reyes.

The adjacency condition is a unique and critical feature of quad-fragment merging; it prevents many merges that would otherwise result in objectionable shading artifacts. Compact encoding of mesh adjacency allows the adjacency condition to be verified inexpensively. Intelligent ordering of grid micropolygons by dicing allows the Merge stage to identify nearly all possible merges using only a small amount of buffering.

Unmodified Stages

Although Figure 7.1 highlights changes made to the GPU pipeline in this dissertation, it is important to recognize that a significant fraction of the GPU pipeline architecture undergoes only minor modification or remains the same. Minimizing changes

to the GPU pipeline was not an initial design goal for this work, but it became an increasingly valuable feature as development of the micropolygon pipeline architecture progressed. It provides continuity for application developers (programming the micropolygon pipeline is essentially the same as the GPU pipeline: only a few extra shader programs are needed for DiagSplit) as well as for GPU implementers (many highly optimized components in current GPUs can be reused). Moreover, as an extension of the existing GPU pipeline, the micropolygon pipeline remains capable of rendering surfaces represented by lower resolution polygon meshes.

Even with increasing GPU compute capability, the ability to efficiently render a mixture of micropolygon and non-micropolygon meshes will continue to be important for real-time graphics applications. For example, tessellation to micropolygon-resolution meshes is not necessary to adequately represent surfaces with low geometric detail. Conveniently, the micropolygon pipeline affords future application developers the ability to tune polygon size to optimize user experience. In situations where only small-polygon meshes (but not micropolygon meshes) are necessary for high-quality rendering, the DiagSplit, rasterization, and quad-fragment merging algorithms from this dissertation still improve rendering performance over the algorithms used in GPUs today.

7.2 Key Ideas and Design Principles

Three key principles clearly exerted heavy influence on the design of the micropolygon pipeline architecture. I suspect variants of these ideas will also prove valuable to future researchers seeking to advance the capabilities of real-time graphics systems.

Reason about surfaces, not individual micropolygons. When rendering micropolygons, it is advantageous to think about performing key pipeline operations on surfaces, not individual micropolygons. For example, the micropolygon pipeline partitions surface patches into surface subpatches. It accepts bounding boxes that allow subpatches to be culled before micropolygons are even created. It samples shading for an entire grid approximately once per pixel. The motivation for this shift

in thinking is performance. In a micropolygon rendering pipeline, a subpatch (or its corresponding grid), not an individual micropolygon, is the key unit of locality and coherence. (This is logical, since subpatches, like the polygons used by most real-time applications today, generally cover tens to hundreds of pixels.) Many opportunities to optimize execution by sharing computation and data across rendering operations do exist in a micropolygon pipeline. However, the small size of micropolygons requires pipeline implementations to be able to identify these opportunities over regions of a surface that extend beyond a single micropolygon's boundaries.

Rasterization is one of the few places in the micropolygon pipeline where micropolygons are treated independently. The triangle-pairs optimization discussed in Chapter 4 does leverage surface semantics to reduce rasterization work, but in general it is not obvious how to extend the idea of directly processing surfaces to multi-sample coverage testing. The micropolygon pipeline deconstructs grids into individual triangles for rasterization, but it includes connectivity information with rasterized quad fragments to allow grid topology to be "reconstructed" as needed by the Merge stage.

Extend, rather than replace. Rather than replace or significantly modify the GPU pipeline's existing components, I often elected to add new components to the micropolygon pipeline. These additions manage micropolygon-specific complexity and convert processing into operations that existing GPU pipeline components handle well (again, rasterization stands as the one exception to this strategy). For example, DiagSplit partitions input base patches into subpatches that are tessellated and evaluated using the GPU pipeline's existing Uniform tessellation and Vertex Processing stages. Similarly, quad-fragment merging produces quad fragments that are much like quad fragments generated when rasterizing large triangles: they have dense multi-sample coverage, support computation of derivatives using finite differences, and are shaded independently by Fragment Processing (without requiring the graphics-application programmer to modify shader programs).

Fortunately, for the case of rendering micropolygons, new pipeline functionality, such as performing splits and merging quad fragments, has low cost in comparison to

existing compute-heavy stages like Vertex and Fragment Processing. From an architectural perspective, DiagSplit and quad-fragment merging are attractive algorithms because the extra work needed to transform adaptive tessellation and micropolygon shading tasks into a workload that runs efficiently on GPUs is small.

Be cognizant of opportunities for fixed-function acceleration. When incorporating new functionality into the micropolygon pipeline, I identified irregular, data-dependent computations that were important to efficient rendering ("regularizing" these computations was either inefficient or impractical due to data dependencies). Then I isolated this logic from regular, data-parallel processing in the pipeline abstraction. In both DiagSplit and quad-fragment merging, there is little benefit to exposing irregular parts of these algorithms to the application programmer. However, there are significant opportunities for the graphics system to provide highly optimized implementations of the irregular parts of these algorithms. For example, a wide SIMD processor is not an ideal platform for merging quad fragments, but it is likely these operations can be carried out using a small amount of custom hardware (the algorithm was designed with this implementation in mind).

Although modern graphics research often calls for expanding the capabilities of programmable GPU cores in new ways, the transition to micropolygon rendering required me to focus on the fixed-function components of GPU pipeline architecture. In this work, I believe considering the implications of heterogeneous processing led to simpler solutions and more elegant abstractions than what would have been possible considering only "massively data-parallel" algorithms.

7.3 Next Steps

In previous chapters, I have listed many aspects of tessellation, rasterization, and shading that invite further study. However, the results of this dissertation point clearly to two sizable areas of future work that require thinking about the pipeline as a whole. The first is best attempted by product teams in the graphics industry: developing an optimized end-to-end implementation of the micropolygon pipeline

architecture that achieves real-time rendering performance on complex scenes. The second demands attention from all graphics researchers: understanding how best to integrate motion blur (and potentially defocus blur) into a real-time graphics system.

GPU Integration

I hope see optimized implementations of the algorithms proposed in this dissertation integrated into future GPUs. Although the evolutionary design of the micropolygon pipeline architecture should simplify this process, it remains a substantial task.

One important integration challenge not addressed in this dissertation is micropolygon pipeline scheduling. GPU scheduling polices are highly proprietary and finely tuned to the properties of specific GPU implementations (thus it would be hard to evaluate scheduling and workload distribution policies without an optimized end-to-end system). Although current GPUs adapt well to variation in pipeline load, micropolygon rendering does present unique challenges. Micropolygons shift the balance of work in the GPU pipeline (vertex processing work increases substantially) and adaptive tessellation performs unbounded data amplification early in the pipeline (maintaining pipeline ordering semantics is challenging). For reasons such as these, I predict pipeline scheduling, rather than engineering optimized software/hardware implementations of DiagSplit, micropolygon rasterization, and quad-fragment merging, will be the most difficult aspect of integrating these algorithms into future GPUs.

Adding Motion Blur

Efficiently integrating motion blur effects into the real-time micropolygon pipeline remains an open problem. Rendering blurred micropolygons affects many parts of the graphics pipeline and, as was the case to improve support for micropolygon geometry, requires innovation in both algorithms and pipeline architecture. As stated previously, the best algorithms for rasterizing blurred micropolygons have low efficiency (despite the contributions of Chapter 5) and the interaction of quad-fragment merging with proposals to add motion blur to the GPU pipeline [Ragan-Kelley et al. 2010] has not

been fully explored. Knowledge of surface motion provides opportunities to optimize how the pipeline implements culling [Boulos et al. 2010], tessellates surfaces, and samples shading (blur is an excellent context in which to compare vertex and quad-fragment shading). While advances in all of these areas may not be required to make accurate motion blur practical for real-time rendering, the possibilities for innovation are large.

Chapter 8

Conclusion

For the foreseeable future, advanced real-time graphics applications will find use for as much compute capability as graphics systems can deliver. The demand for high performance, combined with practical constraints on chip area, cost, and (most importantly going forward) power, requires graphics systems to be very efficient. In this dissertation I studied the efficiency of the modern real-time graphics pipeline when executing an advanced graphics workload: rendering scenes with surfaces represented accurately using micropolygon meshes. I identified that the fundamental pipeline operations of tessellation, rasterization, and shading execute inefficiently when rendering micropolygons. In response, I proposed an evolution of the GPU pipeline architecture that overcomes key problems in each of these three areas.

Of course, synthesizing beautiful images involves more than accurately rendering complex surfaces. Simulating high-quality materials and lighting are also essential, and reducing the cost of these computations is the focus of much rendering research today. Moreover, the richness of an interactive experience depends on more than just the color of image pixels. Animation, sound, game play, and storytelling are just as critical to a user's experience as the imagery placed before his or her eyes. Still, I challenge you to watch your favorite scene from a Pixar film. Take a look at the wild contortions of Woody's face, at a sad Boo's eyes, and at the detail in Sulley's blue fur or Carl's old sweater. Even with the masterful animation, beautiful lighting, and powerful musical score that drives these scenes, I assert they would not be nearly as

iconic or nearly as emotive if we could see clear evidence that the characters were merely computer models formed by polygons.

This belief was staunchly held by early researchers exploring the use of computer graphics in film, and it should be central to the design of interactive graphics systems going forward. Not long ago demanding this level of quality from an interactive application was a fantastic goal. However, near-future GPUs will boast many teraflops of compute capability. I assert that only modest algorithmic and architectural changes to the current real-time graphics pipeline are required to make micropolygon rendering on these future systems efficient. Given these two facts, I am confident that soon you will be hard pressed to find visual artifacts caused by coarse polygon meshes in interactive rendering.

Appendix A

Interleaved Sampling Tile Permutations

In the following pseudocode, the function `compute_multisample_position` computes the XY screen position of the multi-sample point in frame-buffer tile (`tile_x`, `tile_y`) that is associated with UVT tuple indentified by `uvt_index`. The resulting interleaved sampling pattern features 2×2-pixel tiles ($K_x = K_y = 2$), but unlike the patterns described by Keller et al. [2001], the tile-relative position of the multi-sample point associated with UVT tuple uvt_i is tile dependent. As a result, multi-sample points sharing the same UVT value do not form a regular grid over the image plane (see interleaved sampling with permutations in Section 5.3.2). Figure A.1 illustrates the UVT tuple indexing scheme assumed by `compute_multisample_position`.

The XYUVT multi-sample points produced by `compute_multisample_position` are stratified over individual pixels as well as over each tile. Using a small, precomputed table of jitter magnitudes, different subpixel xy-jitter values are generated for multi-sample points in each pixel following the patented technique described by Cook et al. [1990]. The use of Cook's spatial jittering technique is unrelated to the implementation of interleaved sampling with permutations. Its inclusion in the pseudocode below is provided for completeness and consistency with the rasterizer implementations evaluated in Chapter 5.

Kx x Ky = K pixels per tile
P multi-sample points per pixel
P x K = N total UVT tuples

uvt_index:	uvt_0	uvt_1	uvt_2	...	uvt_{K-1}	uvt_K	uvt_{K+1}	uvt_{K+2}	...	uvt_{2K-1}	...	uvt_{N-P}	uvt_{N-P+1}	uvt_{N-P+2}	...	uvt_{N-1}
strata_index:	0	0	0	...	0	1	1	1	...	1		P-1	P-1	P-1	...	P-1
substrata_index:	0	1	2	...	K-1	0	1	2	...	K-1		0	1	2	...	K-1
	UVT tuples assigned to xy strata 0 of different pixels in the same tile					UVT tuples assigned to xy strata 1 of different pixels in the same tile						UVT tuples assigned to xy strata P-1 of different pixels in the same tile				

Each tile is stratefied in UVT-space (there are N stratefied samples)
Each pixel is also stratefied in UVT-space
(it receives one UVT-tuple from each of the P groups of K)

Figure A.1: UVT tuple indexing scheme and corresponding values of `strata_index` (the current subpixel strata associated with the tuple) and `substrata_index` (tile-relative pixel index prior to permutation) computed by the function `compute_multisample_location`.

.

```
int    K_X = 2;
int    K_Y = 2;
int    PIXEL_PER_TILE = 4;     // given as K in figure A.1
int    SAMPLES_PER_PIXEL = 16; // given as P in figure A.1

int    NUM_PERMUTATIONS = impl dependent (Ch. 5 impl uses 64)
int    NUM_JITTERS      = impl dependent (Ch. 5 impl uses 512)

float  X_JITTERS[NUM_JITTERS];  // x offset from xy strata center
float  Y_JITTERS[NUM_JITTERS];  // y offset from xy strata center

float  STRATA_BASE_X[SAMPLES_PER_PIXEL];  // stores xy strata X centers
float  STRATA_BASE_Y[SAMPLES_PER_PIXEL];  // stores xy strata Y centers

// each permutation is a permutation of PIXELS_PER_TILE integers mapping a
// substrata_index to a tile-relative pixel_index: (0-4) -> (0-4) in this case
int    PERMUTATION_TABLE[NUM_PERMUTATIONS][PIXELS_PER_TILE];
```

```
// Compute XY screen position of multi-sample point in screen
// tile (tile_x, tile_y) that has UVT values given by uvt_index
void compute_multisample_position(
  int    uvt_index,
  int    tile_x,
  int    tile_y,
  float& multisample_x,
  float& multisample_y)
{
  int strata_index = uvt_index / PIXELS_PER_TILE;
  int substrata_index = uvt_index % PIXELS_PER_TILE;

  // select a permutation
  int permutation_index = compute_permutation_index(tile_x, tile_y, strata_index);

  // compute pixel xy
  int pixel_index = PERMUTATION_TABLE[permutation_index][substrata_index];
  int pixel_x = K_X * tile_x + pixel_index % K_X;
  int pixel_y = K_Y * tile_y + pixel_index / K_X;

  // compute subpixel offset
  int jitter_index = compute_jitter_index(pixel_x, pixel_y, strata_index);
  multisample_x = pixel_x + STRATA_BASE_X[strata_index] + X_JITTERS[jitter_index];
  multisample_y = pixel_y + STRATA_BASE_Y[strata_index] + Y_JITTERS[jitter_index];
}
```

The helper functions `compute_permutation_index` and `compute_jitter_index` hash their inputs to compute indices into precomputed permutation and xy-jitter tables. Implementations used to evaluate the rasterizers in Chapter 5 are given below. The constants `C0`-`C4` are relatively-prime integers.

```
int compute_permutation_index(
  int tile_x,
  int tile_y,
  int strata_index)
{
  return (C0 * tile_x + C1 * tile_y + C2 * strata_index) % NUM_PERMUTATIONS;
}

int compute_jitter_index(
  int pixel_x,
  int pixel_y,
  int strata_index)
{
  return (C3 * pixel_y + pixel_x + C4 * strata_index)) % NUM_JITTERS;
}
```

Appendix B

5D Point-in-Micropolygon Tests

The INTERVAL and INTERLEAVE rasterization algorithms described in Chapter 5 must determine a moving, defocus-blurred micropolygon's coverage of frame buffer multi-sample points. While Chapter 5 focused on strategies for reducing the number of point-in-micropolygon tests performed, it is equally important that these tests are executed efficiently. This appendix describes the 5D (XY,UV,T) point-in-micropolygon test used in the implementation of INTERVAL and INTERLEAVE.

A simple way to compute 5D micropolygon-multi-sample point coverage is to position the micropolygon in world space at the time t associated with the multi-sample point, then perform a ray-micropolygon test with the appropriate ray originating from the lens of the virtual camera [Cook et al. 1984]. (Kolb et al. [1995] describe how to compute such a ray from parameters (x,y,u,v) using Gaussian approximations to real lens systems.) In the language of rasterization, this test positions the micropolygon at time t, projects its vertices into screen space (using the virtual lens position u,v as the point of projection), then performs a conventional 2D point-in-polygon test given the micropolygon's screen-space projection. These solutions provide accurate, perspective-correct point-in-micropolygon tests, but have high computational cost.

To reduce computational cost, the 5D point-in-polygon test employed by the INTERVAL and INTERLEAVE implementations in Chapter 5 operates almost entirely on post-projection vertex positions. Pseudocode for the algorithm is given at the end of this appendix (see function `5d_point_in_triangle_test`).

In this code, the helper function `compute_screen_circle_of_confusion_rad` (implementation not shown) uses the thin lens approximation to compute the circle-of-confusion radius, in screen-space units, for a point a given distance from the virtual camera [Kolb et al. 1995]. The helper function `defocus_shift_direction` returns a unit vector defining the direction to shift the projected vertex position to account for finite lens aperture. These directions can be precomputed for a frame given virtual camera parameters, reducing this function call to a table lookup.

Although inexpensive to compute, interpolating vertex positions in screen space does not produce perspective-correct motion blur. My experiences indicate that errors due to this approximation are difficult to observe when viewing animated sequences played back at full speed. Similarly, the implementation of defocus does not account for small perspective changes when viewing the scene from different parts of the lens (e.g., given the code above, a triangle cannot flip from being front facing to back facing when viewing it from different lens positions).

When only 3D (XY,T) or 4D (XY,UV) sampling is required, additional optimizations are possible. For example, when motion blur is not required, each vertex's circle of confusion is computed only once per triangle instead of once per sample test. One benefit of the pseudocode below is that it leverages the same 2D point-in-triangle kernel required by MPRAST. Since only one sample is tested against the triangle each time it is positioned, `2D_point_in_triangle_test` does not perform edge-equation setup (see Section 4.2). However, when only 3D or 4D sampling is required, other point-in-triangle approaches exist. Möller et al. [2007] describe how to compute time-dependent edge equations to more efficiently perform (XY,T) point-in-polygon tests. Similarly, Ragan-Kelley et al. [2010] construct edge functions for the case of (XY,UV) sampling. Although these methods offer elegant solutions for performing point-in-triangle tests, INTERLEAVE must already position and project the triangle to determine potentially overlapped tiles. Given this constraint, the 2D point-in-polygon test is cheaper than point-in-polygon tests in either Möller et al.'s or Ragan-Kelley et al.'s approaches.

```
// Determine if the moving, defocus-blurred triangle covers
// the given 5D multi-sample point
void 5d_point_in_triangle_test(
  Vertex2D open_shutter_pos[3],
  Vertex2D close_shutter_pos[3],
  float    open_shutter_camz[3],
  float    close_shutter_camz[3],
  float    sample_x,
  float    sample_y,
  float    sample_t,
  float    sample_u,
  float    sample_v,
  bool&    is_covered)
{
  Vertex2D cur_time_pos[3];
  float    cur_time_camz[3];

  // compute screen position and camera-space depth of all
  // triangle vertices at the time given by sample_t
  for (int i=0; i<3; i++) {
     cur_time_pos[i] = lerp(sample_t, open_shutter_pos[i], close_shutter_pos[i]);
     cur_time_camz[i] = lerp(sample_t, open_shutter_camz[i], close_shutter_camz[i]);
  }

  // shift the screen position of the vertex due to defocus.  The lens
  // position of the sample point determines the direction of the shift.
  // The camera-space depth of the vertex determines the magnitude.
  for (int i=0; i<3; i++) {
     float radius = compute_screen_circle_of_confusion_rad(cur_time_camz[i]);
     cur_time_pos[i] += radius * defocus_shift_direction(sample_u, sample_v);
  }

  // screen position of triangle vertices is now set for the
  // multi-sample point.  Now perform a conventional 2D
  // point-in-triangle test
  is_covered = 2D_point_in_triangle_test(cur_time_pos, sample_x, sample_y);
}
```

Bibliography

ABRASH, M. 2009. Rasterization on Larrabee. *Dr. Dobb's Portal*. Available at `http://www.drdobbs.com/high-performance-computing/217200602`.

ADVANCED MICRO DEVICES. 2010. *ATI Radeon HD 5870 GPU feature summary*. Available at `http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5870/Pages/ati-radeon-hd-5870-specifications.aspx`.

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on GPUs. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, ACM, 145–149.

AKELEY, K. 1993. RealityEngine graphics. In *Proceedings of SIGGRAPH 93*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, 109–116.

AKENINE-MÖLLER, T., MUNKBERG, J., AND HASSELGREN, J. 2007. Stochastic rasterization using time-continuous triangles. In *Graphics Hardware 2007*, 7–16.

AKENINE-MÖLLER, T., HAINES, E., AND HOFFMAN, N. 2008. *Real-Time Rendering*, third ed. A. K. Peters, Ltd.

ANDERSSON, J. 2009. Parallel graphics in Frostbite - current and future. In *ACM SIGGRAPH 2009 Courses: Beyond Programmable Shading I*. Available at `http://s09.idav.ucdavis.edu`.

ANDERSSON, J. 2010. Five major challenges in interactive rendering. In *ACM SIGGRAPH 2010 Courses: Beyond Programmable Shading I*. Available at `http://bps10.idav.ucdavis.edu`.

APODACA, A. A., AND GRITZ, L. 2000. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann.

ARIJON, D. 1991. *Grammar of the Film Language*. Silman-James Press, ch. 28, 602.

Blinn, J. F. 1978. *Computer display of curved surfaces*. PhD thesis, The University of Utah.

Blythe, D. 2006. The Direct3D 10 system. *ACM Transactions on Graphics 25*, 3 (Aug), 724–734.

Blythe, D. 2008. Rise of the graphics processor. *Proceedings of the IEEE 96*, 5 (May), 761–778.

Boulos, S., Luong, E., Fatahalian, K., and Hanrahan, P. 2010. Space-time hierarchical occlusion culling for micropolygon rendering with motion blur. In *HPG '10: Proceedings of the Conference on High Performance Graphics 2010*, Eurographics Association.

Brunhaver, J., Fatahalian, K., and Hanrahan, P. 2010. Hardware implementation of micropolygon rasterization with motion and defocus blur. In *HPG '10: Proceedings of the Conference on High Performance Graphics 2010*, Eurographics Association.

Catmull, E. E. 1974. *A subdivision algorithm for computer display of curved surfaces*. PhD thesis, The University of Utah.

Clark, J. H. 1979. A fast scan-line algorithm for rendering parametric surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 79)*, ACM, vol. 13, 174.

Cook, R. L., Porter, T., and Carpenter, L. 1984. Distributed ray tracing. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, ACM, vol. 18, 137–145.

Cook, R., Carpenter, L., and Catmull, E. 1987. The Reyes image rendering architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, ACM, vol. 27, 95–102.

Cook, R. L., Porter, T. K., and Carpenter, L. C., 1990. Pseudo-random point sampling techniques in computer graphics. United States Patent 4,897,806.

Cook, R. L. 1986. Stochastic sampling in computer graphics. *ACM Transactions on Graphics 5*, 1, 51–72.

Deering, M., Winner, S., Schediwy, B., Duffy, C., and Hunt, N. 1988. The triangle processor and normal vector shader: a VLSI system for high performance graphics. In *Computer Graphics (Proceedings of SIGGRAPH 88)*, ACM, vol. 22, 21–30.

Demers, J. 2004. Depth of field: A survey of techniques. *GPU Gems*, 375–390.

Egan, K., Tseng, Y.-T., Holzschuch, N., Durand, F., and Ramamoorthi, R. 2009. Frequency analysis and sheared reconstruction for rendering motion blur. *ACM Transactions on Graphics 28*, 3, 93:1–93:13.

Eisenacher, C., and Loop, C. 2010. Data-parallel micropolygon rasterization. In *Proceedings of Eurographics 2010*.

Eisenacher, C., Meyer, Q., and Loop, C. 2009. Real-time view-dependent rendering of parametric surfaces. In *I3D '09: Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, ACM, 137–143.

Fatahalian, K., and Houston, M. 2008. A closer look at GPUs. *Communications of the ACM 51*, 10, 50–57.

Fatahalian, K., Luong, E., Boulos, S., Akeley, K., Mark, W. R., and Hanrahan, P. 2009. Data-parallel rasterization of micropolygons with defocus and motion blur. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, ACM, 59–68.

Fatahalian, K., Boulos, S., Hegarty, J., Akeley, K., Mark, W. R., Moreton, H., and Hanrahan, P. 2010. Reducing shading on GPUs using quad-fragment merging. *ACM Transactions on Graphics 29*, 4, 67:1–67:8.

Fatahalian, K. 2010. From shader code to a teraflop: How GPU shader cores work. In *ACM SIGGRAPH 2010 Courses: Beyond Programmable Shading I*. Available at `http://bps10.idav.ucdavis.edu`.

Fisher, M., Fatahalian, K., Boulos, S., Akeley, K., Mark, W. R., and Hanrahan, P. 2009. DiagSplit: parallel, crack-free, adaptive tessellation for micropolygon rendering. *ACM Transactions on Graphics 28*, 5, 150:1–150:10.

Foster, C., 2009. Aqsis renderer. `http://www.aqsis.org/`.

Fowler, M. 2010. ATI Radeon HD5000 series: An inside view. In *High Performance Graphics 2010 (Hot3D talk)*. Available at `http://www.highperformancegraphics.org/media/Hot3D/HPG2010_Hot3D_AMD.pdf`.

Fuchs, H., Goldfeather, J., Hultquist, J. P., Spach, S., Austin, J., Frederick P. Brooks, J., Eyles, J., and Poulton, J. 1985. Fast spheres, shadows, textures, transparencies, and image enhancements in Pixel-Planes. In *Computer Graphics (Proceedings of SIGGRAPH 85)*, ACM, vol. 19, 111–120.

Fuchs, H., Poulton, J., Eyles, J., Greer, T., Goldfeather, J., Ellsworth, D., Molnar, S., Turk, G., Tebbs, B., and Israel, L. 1989. Pixel-planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, ACM, vol. 23, 79–88.

Greene, N., Kass, M., and Miller, G. 1993. Hierarchical z-buffer visibility. In *Proceedings of SIGGRAPH 93*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, 231–238.

Greene, N. 1996. Hierarchical polygon tiling with coverage masks. In *Proceedings of SIGGRAPH 96*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, 65–74.

Haeberli, P., and Akeley, K. 1990. The accumulation buffer: hardware support for high-quality rendering. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, ACM, vol. 24, 309–318.

Hegarty, J. 2010. *An Implementation of Quad-Fragment Merging For Micropolygon Rendering*. Undergraduate thesis, Stanford University.

Henne, M., Hickel, H., Johnson, E., and Konishi, S. 1996. The making of Toy Story. In *Compcon '96. Technologies for the Information Superhighway Digest of Papers*, 463–468.

Houston, M. 2008. Anatomy of AMD's TeraScale graphics engine. In *ACM SIGGRAPH 2008 Classes: Beyond Programmable Shading: Fundamentals*. Available at `http://s08.idav.ucdavis.edu`.

Keller, A., and Heidrich, W. 2001. Interleaved sampling. In *Eurographics Workshop on Rendering*, 269–276.

Kessenich, J. 2009. *The OpenGL Shading Language Specification, Language Version 4.0*.

Khronos. 2010. *The OpenCL Specification (Version 1.1)*. Khronos OpenCL Working Group.

Kolb, C., Mitchell, D., and Hanrahan, P. 1995. A realistic camera model for computer graphics. In *Proceedings of SIGGRAPH 95*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, 317–324.

Kovacs, D., Mitchell, J., Drone, S., and Zorin, D. 2009. Real-time creased approximate subdivision surfaces. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ACM, 155–160.

Lane, J. M., Carpenter, L. C., Whitted, T., and Blinn, J. F. 1980. Scan line methods for displaying parametrically defined surfaces. *Communications of the ACM 23*, 1, 23–34.

Lauritzen, A. 2010. Deferred rendering for current and future rendering pipelines. In *ACM SIGGRAPH 2010 Courses: Beyond Programmable Shading II*. Available at `http://bps10.idav.ucdavis.edu`.

Lee, S., Eisemann, E., and Seidel, H.-P. 2010. Real-time lens blur effects and focus control. *ACM Transactions on Graphics 29*, 4, 65:1–65:7.

Levinthal, A., Hanrahan, P., Paquette, M., and Lawson, J. 1987. Parallel computers for graphics applications. In *ASPLOS-II: Proceedings of the second international conference on Architectual support for programming languages and operating systems*, IEEE Computer Society Press, 193–198.

Lien, S., Shantz, M., and Pratt, V. 1987. Adaptive forward differencing for rendering curves and surfaces. *Computer Graphics (Proceedings of ACM SIGGRAPH 87) 21*, 4, 111–118.

Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. 2008. NVIDIA Tesla: A unified graphics and computing architecture. *Micro, IEEE 28*, 2 (Mar), 39–55.

Loop, C., and Schaefer, S. 2008. Approximating Catmull-Clark subdivision surfaces with bicubic patches. *ACM Transactions on Graphics 27*, 1, 8:1–8:11.

Loop, C., Schaefer, S., Ni, T., and Castaño, I. 2009. Approximating subdivision surfaces with Gregory patches for hardware tessellation. *ACM Transactions on Graphics 28*, 5, 151:1–151:9.

Mark, W. R., Glanville, S. R., Akeley, K., and Kilgard, M. J. 2003. Cg: a system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics 22*, 3 (Aug).

Mark, W. 2008. Future graphics architectures. *Queue 6*, 2, 54–64.

McCool, M. D., Wales, C., and Moule, K. 2001. Incremental and hierarchical hilbert order edge equation polygon rasterization. In *Graphics Hardware 2001*, 65–72.

McCormack, J., and McNamara, R. 2000. Tiled polygon traversal using half-plane edge functions. In *Graphics Hardware 2000*, 15–21.

McGuire, M., Enderton, E., Shirley, P., and Luebke, D. 2010. Hardware-accelerated stochastic rasterization on conventional GPU architectures. In *Proceedings of High Performance Graphics 2010*.

Microsoft Corporation. 2010. *Rasterization Rules (Direct3D 10)*. Available at `http://msdn.microsoft.com/en-us/library/cc627092(v=VS.85).aspx`.

Microsoft Corporation. 2010. *Windows DirectX Graphics Documentation*. Available at `http://msdn.microsoft.com/en-us/library/ee663301(v=VS.85).aspx`.

Mitchell, D. P., 1990. The antialiasing problem in ray tracing. SIGGRAPH 90 Course Notes: Advanced Topics in Ray Tracing. Available at `http://www.mentallandscape.com/Papers_siggraph90tutorial.pdf`.

Mitchell, D. P. 1991. Spectrally optimal sampling for distribution ray tracing. *Computer Graphics (Proceedings of SIGGRAPH 91) 25*, 4, 157–164.

Molnar, S., Eyles, J., and Poulton, J. 1992. PixelFlow: high-speed rendering using image composition. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, ACM, vol. 26, 231–240.

Morein, S. 2000. ATI Radeon HyperZ technology. In *Graphics Hardware 2000 (Hot3D talk)*. Available at `http://www.graphicshardware.org/previous/www_2000/presentations/ATIHot3D.pdf`.

Moreton, H. 2001. Watertight tessellation using forward differencing. In *Proceedings of the Eurographics Workshop on Graphics Hardware*, ACM, 25–32.

Ni, T., Castaño, I., Peters, J., Mitchell, J., Schneider, P., and Verma, V. 2009. Efficient substitutes for subdivision surfaces. In *ACM SIGGRAPH 2009 Courses*, ACM, 13:1–13:107.

NVIDIA Corporation. 2009. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. Available at `http://www.nvidia.com/object/IO_89570.html`.

NVIDIA Corporation. 2010. *NVIDIA CUDA C Programming Guide (Version 3.1.1)*.

NVIDIA Corporation. 2010. *NVIDIA GF100 Whitepaper*. Available at `http://www.nvidia.com/object/IO_89569.html`.

Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., and Stich, M. 2010. OptiX: a general purpose ray tracing engine. *ACM Transactions on Graphics 29*, 4, 66:1–66:13.

Patney, A., and Owens, J. D. 2008. Real-time Reyes-style adaptive surface subdivision. *ACM Transactions on Graphics 27*, 5, 143:1–143:8.

Patney, A., Ebeida, M. S., and Owens, J. D. 2009. Parallel view-dependent tessellation of Catmull-Clark subdivision surfaces. In *HPG '09: Proceedings of High Performance Graphics 2009*, ACM, 99–108.

Pharr, M., Lefohn, A., Kolb, C., Lalonde, P., Foley, T., and Berry, G. 2007. Programmable graphics - the future of interactive rendering. Tech. rep., Neoptica, Inc.

Pineda, J. 1988. A parallel algorithm for polygon rasterization. In *Computer Graphics (Proceedings of SIGGRAPH 88)*, ACM, vol. 22, 17–20.

Ragan-Kelley, J., Lehtinen, J., Chen, J., Doggett, M., and Durand, F. 2010. Decoupled sampling for real-time graphics pipelines. *ACM Transactions on Graphics (to appear)*.

Ritchie, M., Modern, G., and Mitchell, K. 2010. Split second motion blur. In *ACM SIGGRAPH 2010 Talks*, ACM.

Roca, J., Moya, V., Gonzalez, C., Escandell, V., Murciego, A., Fernandez, A., and Espasa, R. 2010. A SIMD-efficient 14 instruction shader program for high-throughput microtriangle rasterization. *The Visual Computer 26*, 707–719.

Rockwood, A. P., Heaton, K., and Davis, T. 1989. Real-time rendering of trimmed surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, ACM, vol. 23, 107–116.

Segal, M., and Akeley, K. 1994. The design of the OpenGL graphics interface. Tech. rep., Silicon Graphics Computer Systems.

Segal, M., and Akeley, K. 2010. *The OpenGL Graphics System: A Specification (Version 4.0)*. The Khronos Group, Inc.

Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., and Hanrahan, P. 2008. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics 27*, 3, 18:1–18:15.

Snider, B. 1995. The Toy Story story. *Wired Magazine 3*, 12 (Dec).

Stam, J. 1995. Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values. In *Proceedings of SIGGRAPH 98*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, 395–404.

Sugerman, J., Fatahalian, K., Boulos, S., Akeley, K., and Hanrahan, P. 2009. GRAMPS: A programming model for graphics pipelines. *ACM Transactions on Graphics 28*, 1, 4:1–4:11.

Sung, K., Pearce, A., and Wang, C. 2002. Spatial-temporal antialiasing. *IEEE Transactions on Visualization and Computer Graphics 8*, 2, 144–153.

Sweeney, T. 2009. The end of the GPU roadmap. In *High Performance Graphics 2010 (keynote talk)*. Available at `http://www.highperformancegraphics.org/previous/www_2009/presentations/TimHPG2009.pdf`.

Wald, I., Boulos, S., and Shirley, P. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1, 6:1–6:18.

Wexler, D., Gritz, L., Enderton, E., and Rice, J. 2005. GPU-accelerated high-quality hidden surface removal. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, ACM, 7–14.

Zhou, K., Hou, Q., Ren, Z., Gong, M., Sun, X., and Guo, B. 2009. RenderAnts: interactive Reyes rendering on GPUs. *ACM Transactions on Graphics 28*, 5, 155:1–155:11.