FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Micropolygon Rendering on the GPU

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Thomas Weber
Matrikelnummer 0526341

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Associate Prof. John D. Owens

Wien, 4.12.2014

(Unterschrift Verfasser)

(Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Micropolygon Rendering on the GPU

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Thomas Weber
Registration Number 0526341

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Associate Prof. John D. Owens

Vienna, 4.12.2014

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Thomas Weber
Zieglergasse 27, 1070 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

I'd like to thank Anjul Patney, Stanley Tzeng, Julian Fong, and Tim Foley for their valuable input. Another "thank you" goes to Nuwan Jayasena of AMD for supplying me with testing harware and giving support on driver issues.

This thesis was supported by a scholarship from the Austrian Marshall Plan Foundation.

Abstract

Recent advances in graphics hardware have made it feasible to consider implementing alternative rendering algorithms on the GPU. One such candidate would be Reyes, an algorithm that subdivides surfaces into sub-pixel sized polygons called micropolygons before rasterizing them. This allows rendering of curved and displaced surfaces without any visible geometric artifacts.

While the overall Reyes pipeline seems to map well to the data-parallel programming mode of GPUs, implementing a Reyes renderer on the GPU has so far been hampered by several factors. One of those reasons is that current rendering hardware is not designed for sub-pixel sized polygons. Another problematic component of Reyes is its bound-and-split phase, where surface patches are recursively split until they are smaller than a given screen-space bound. While this operation has been successfully parallelized for execution on the GPU using a breadth-first traversal, the resulting implementations are limited by their unpredictable worst-case memory consumption and high global memory bandwidth utilization.

This thesis presents a full Reyes renderer implemented in the GPU programming language and platform OpenCL. All surface tessellation, shading, and rasterization is performed as a kernel on the GPU. A software rasterizer specialized for drawing micropolygons is used for rasterization. Our major contribution is a data-parallel implementation of bound-and-split that allows limiting the amount of necessary memory by controlling the number of assigned worker threads. This allows us to render scenes that would require too much memory to be processed by the breadth-first method.

Kurzfassung

Die zunehmenden Flexibilität und Leistung von Grafikhardware macht es möglich über die Implementierung alternativer Renderingalgorithmen nachzudenken. Ein Kandidat hierfür ist Reyes, ein Algorithmus bei dem Oberflächen in Polygone kleiner als Pixel unterteilt werden, bevor sie rasterisiert werden. Das erlaubt die artefaktfreie Darstellung gekrümmter Oberflächen mit Displacement Mapping.

Obwohl sich die Komponenten der Reyes-Pipeline generell gut in das datenparallele Model aktueller GPUs integrieren, gibt es einige Faktoren die die Implementierung eines praxistauglichen Reyes-Renderers auf der GPU verhindert haben. Ein solcher Grund ist, dass die Rasterisierungshardware von aktuellen GPUs nicht darauf ausgelegt ist, sehr kleine Polygone effizient zu rendern. Ein weiteres Problem stellt die „Bound-and-Split” Phase von Reyes dar, in der Oberflächen so lange rekursiv geteilt werden, bis sie kleiner als eine vorgegebene Größe auf dem Bildschirm haben. Zwar wurde dieser Schritt bereits erfolgreich parallelisiert, indem man die tiefenorientierte Traversierung der Oberflächen in eine breitenorientierte umwandelt, aber der resultierende Algorithmus ist insofern limitiert, dass sein Speicherverbrauch schwer vorherzusehen ist. Schlimmstenfalls kann mehr Speicher benötigt werden als zur Verfügung steht.

Diese Diplomarbeit präsentiert einen GPU-basierten Reyes-Renderer, der in OpenCL programmiert wurde. Sämtliche Oberflächenunterteilung, Shading und Rasterisierung ist als GPU-Kernel implementiert. Zur Rasterisierung der Mikropolygone wird ein GPU-basierter Softwarerasterizer verwendet, der auf diesen Anwendungsfall spezialisiert ist. Allem voran präsentieren wir einen parallelen Unterteilungsalgorithmus, der es möglich macht, den maximalen Speicherverbrauch zu kontrollieren, indem man die Anzahl an parallel bearbeiteten Oberflächen limitiert. Das erlaubt es uns, Szenen darzustellen die einen unannehmbaren Speicherverbrauch mit rein breitenorientierter Unterteilung hätten.

Contents

CHAPTER 1

Introduction

The demand for ever-increasing visual fidelity in computer graphics has led to the development of incredibly powerful hardware. Current graphics processors are highly parallel devices that can process thousands of individual work items using hundreds of individual cores. Graphics devices have also become more flexible. While early GPUs used very specialized hardware to solve very specific render tasks quickly, later hardware has become more and more programmable to the point where current GPUs can be considered general-purpose compute hardware with some added functionality specific to rendering problems.

This is why it is now becoming common to use graphics hardware for performing tasks other than rendering. Thus APIs specialized for programming these general-purpose applications on graphics hardware like OpenCL or CUDA have been developed. This new flexibility also makes it feasible to step away from the classic real-time graphics pipeline based on polygon rasterization and think about implementing alternative rendering algorithms on these powerful devices.

One domain to look for such alternative approaches is in production rendering, which is the branch of computer graphics that is concerned with creating visuals for film and print. These applications usually require very high image quality in terms of detail and freedom from visual artifacts. The rendering algorithm that is most commonly associated with production rendering is Reyes [6].

While the real-time rendering pipeline can only draw scenes composed of planar triangles and polygons, Reyes is able to draw arbitrary curved surfaces without any visible artifacts. It does this by tessellating the surfaces into sub-pixel sized polygons, which are then drawn to the screen. Using these tiny polygons – also called micropolygons – as intermediate representation allows rendering a large array of different, possibly displaced surface types without any geometric artifacts while using a single back-end for shading and sampling. Reyes performs surface shading on the polygon-level, with the sampler only performing simple color interpolation. Decoupling shading from surface sampling also allows for higher-order rasterization techniques like for instance stochastic rasterization of motion blur.

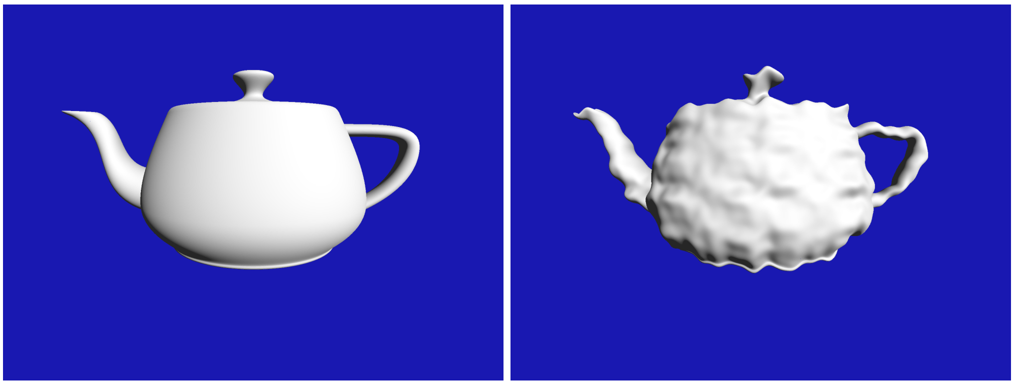


Figure 1.1: Example of an object rendered using our real-time implementation of Reyes. The teapots are defined using 32 Beziér surface patches with the version on the right having additional procedural displacement applied. Note that the teapot's silhouettes are perfectly smooth and no artifacts can be seen in the displaced version.

The flexibility of Reyes and the fact that many content pipelines for production rendering are tooled for this algorithm make it an interesting candidate for implementation on the GPU. Having a full micropolygon rendering pipeline running on graphics hardware would allow for some very impressive visuals. It would also free artists from some performance optimization tasks necessary when designing assets for current graphics pipelines. A GPU-accelerated version of Reyes would also be invaluable for production rendering artists, since it would allow them to do more rapid design iterations.

Furthermore, the overall structure of Reyes generally maps well to the nature of the GPU, exposing ample parallelism that can be taken advantage of. Especially the surface evaluation and shading phases are a very good match for parallel computing, since they operate on grids that can be easily vectorized. In fact the first implementations of Reyes were designed to perform shading on special vector hardware for this very reason.

Despite this, we have yet to see Reyes being used for applications in the real-time domain. One reason for this is that current rasterization hardware is inefficient when rendering micropolygons, since it performs triangle coverage tests and shading in tiles. This has the effect that a large number of cores are occupied even for polygons that only cover a single pixel.

Another problem lies in how Reyes performs the initial subdivision of surface patches. This phase recursively subdivides surfaces until they are smaller than a defined screen-bound. In a sequential implementation this can be implemented as a depth-first traversal with a stack for storing intermediate surfaces. However, this is poorly suited for parallelization on the GPU. It is possible to turn the depth-first traversal into a breadth-first operation, by always operating on all surfaces [28]. This shows good parallel performance, but suffers from the problem that it can have a very high worst-case memory consumption.

Figure 1.2 demonstrates how the memory consumption for breadth-first bound-and-split can look as a camera moves through a scene. (Figure 7.1 shows the same data as a regular graph with axis labels.) Note how this value stays at a mostly constant level for most of the path, with a small number of sharp spikes in memory consumption at certain locations. Properly rendering

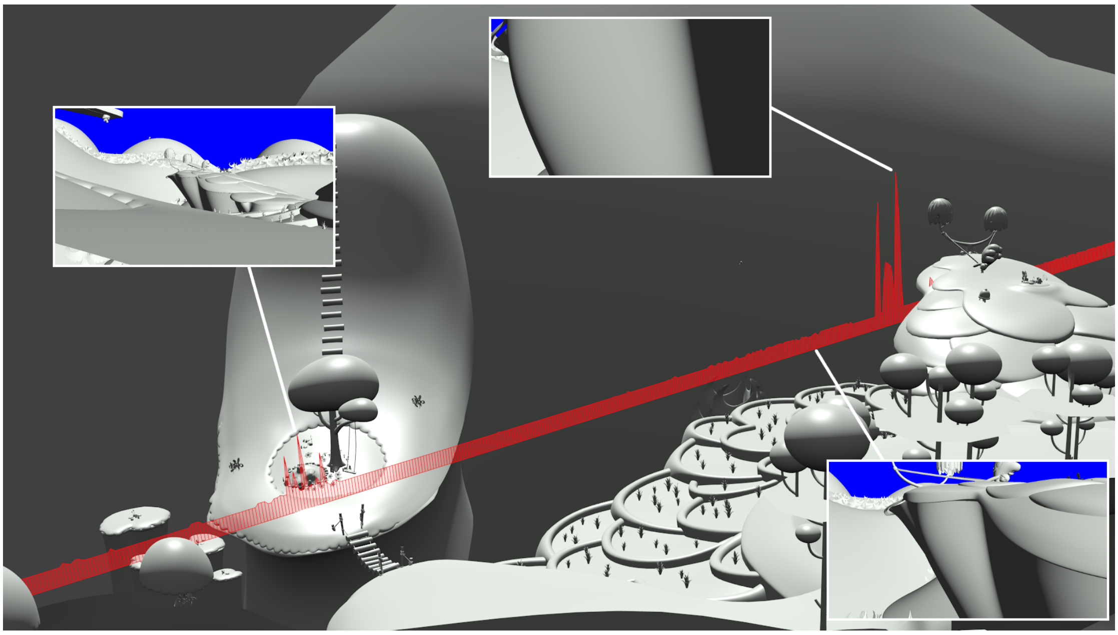


Figure 1.2: Illustration of the memory usage for breadth-first adaptive subdivision as a camera moves through a scene. The thumbnails in the scene show the view from the camera at the highlighted positions. While the overall memory consumption of breadth-first remains mostly constant, there are locations where significantly more memory can be necessary. Scene courtesy of Zinkia Entertainment, S.A.

from views such as these can easily exceed the memory budget of an application or even the available physical memory.

In this thesis we present a renderer based on the Reyes algorithm that is implemented in pure OpenCL. All subdivision, dicing, shading and rasterization is implemented in software as a kernel. Our rasterizer is optimized for processing batches of micropolygons instead of single large triangles.

Furthermore, we present a method that allows choosing the memory budget for parallel bound-and-split. When there is enough available memory, the performance and behavior of the algorithm is the same as the breadth-first approach. In case the breadth-first memory requirements exceed our memory budget, we can get a smooth, asymptotic tradeoff between memory usage and performance. This makes it possible to write rendering systems that perform well in the general case, while still being robust enough to render arbitrary scenes and viewpoints at reasonable performance.

CHAPTER 2

Overview

This chapter is intended to give the reader an overview of this thesis and to explain some concepts that will be necessary later on.

Chapter 3 discusses work related to the topic of GPU-based Reyes rendering. Chapters 4 and 5 describe the methods used in our implementation. Chapter 4 can be seen as the central part of this thesis, since this is where we describe our major contribution: A method for efficient parallel subdivision with a bounded memory usage. Chapter 5 discusses how we implement micropolygon-conversion, shading, and rasterization once the subdivision phase has been performed.

Chapter 6 describes the source architecture of our renderer and gives details on our implementation. The performance of our implemenation is discussed in detail in Chapter 7. This focuses primarily on the performance of adaptive subdivision, since this was chosen to be the focus of this thesis, but overall render performance and its interplay with subdivision is also discussed.

Chapter 8 concludes this work with an overview of our findings and discusses directions for future work.

The rest of this chapter will give an overview on the Reyes rendering architecture in Section 2.1. Section 2.2 gives a short introduction to the terminology and concepts of GPU computing as they are used in this thesis.

2.1 Reyes

The seminal paper on the Reyes image rendering architecture was published by Cook, Carpenter, and Catmull in 1987 [6]. It described an image rendering system developed during the 1980s at a computer animation group at LucasFilm Ltd., which would later form the computer animation studio Pixar. From its inception, Reyes had very high quality and performance requirements.

It was designed for rendering digital imagery for animation in motion pictures. As such it had to be able to render very complex and diverse geometry while still being fast enough to

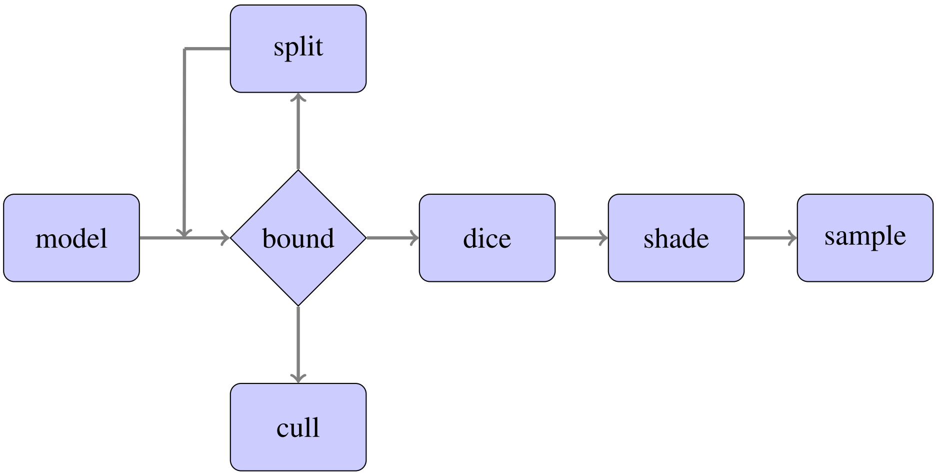


Figure 2.1: The Reyes pipeline

render an entire 2 hour movie within about a year. At 24 frames per second, this allows for a render time of about 3 minutes per frame. It had to perform this with the very limited hardware available at the time.

This is the algorithm that was used for all major Pixar films throughout the 1990s and 2000s. The first critical success was Pixar's 1986 short film Luxo Jr., which received an Academy Award nomination for best animated short film. The 1995 movie Toy Story – also by Pixar – was the first fully computer-animated feature-length film and both a critical and financial success. Both films were animated using Renderman, Pixar's implementation of Reyes.

Reyes draws surfaces by converting them into grids of polygons smaller than individual pixels, which are then shaded and rasterized with a z-buffer. These sub-pixel sized polygons are called micropolygons and form the basis of the Reyes approach. By converting all surfaces into a single common representation before performing shading and rasterization, a large amount of different primitives can be supported with a single optimized back-end for shading and point-sampling. This approach also lends itself well to vectorization since the generated micropolygons share a large amount of locality. It also avoids having to perform complex intersection tests or perspective corrected texture interpolation.

The target micropolygon size of the original Reyes algorithm was a side length of about half a pixel along both axes. This was chosen since it represents the Nynquist limit for the sampled geometry. By going below this and performing multisampling when rasterizing the polygons, any aliasing due to sampling of the shaded microgeometry can be prevented. However, all current production implementations target a side length of a whole pixel and perform per-vertex shading with Gouraud interpolation of the final color instead. This reduces the amount of shading calls by about a factor of four.

Figure 2.1 gives an overview on the Reyes pipeline. Reyes tessellates surfaces into micropolygons using a two-stage approach. The first phase is a bound-and-split operation that estimates the eye-space bound of a surface and checks if the surface is visible and diceable. If it is not visible, then the surface can be culled and no further rendering has to be performed. In case the surface is diceable, the adaptive subdivision phase is over and the surface is ready to be diced. A surface is not considered diceable if micropolygon conversion would create a grid with too many polygons or if the surface shows a large amount of distortion. If the surface is not ready to be diced, then it needs to be subdivided. For parametric surfaces, this usually means splitting the surface patch in half along an iso-parametric axis. Chapter 4 describes how we map adaptive subdivision to the GPU.

After this, the surfaces are evaluated at uniform grid positions to create the micropolygons. This phase is also called *dicing*. The user can also supply a program for applying procedural displacement at this point. If this is the case, then this displacement also needs to be considered when calculating the bound during adaptive subdivision. Since dicing converts single primitives into a large number of micropolygons, it can be considered a data-amplification phase. Our implementation of dicing is described in Section 5.1.

The reason for separating tessellation into these two steps is that it results in more uniformly sized polygons and better vectorization than either step could achieve on its own [11]. Applying only dicing would lead to problematic over- or under-tessellation for parts of surfaces that are strongly distorted, for instance, due to perspective projection. On the other hand, while doing full subdivision up to the micropolygon level is possible, this leads to unnecessary over-tessellation since surfaces can only be halved, effectively limiting the dicing rates to powers of two. Having dicing as a separate phase avoids this, since the optimal dicing rate for every bounded surface can be chosen. Performing shading and rasterization on grids instead of single polygons is also desirable for parallelization, since vertex and face operations can be vectorized.

As already mentioned, Reyes is able to handle a large array of different input primitives. All that needs to be supported for a new primitive type is a method for estimating the eye-space bound, a method for splitting the primitive into smaller ones, and a method for performing dicing. If this is supported, then Reyes is able to convert the primitive into grids of micropolygons. The most straightforward primitive types for this are parametric surfaces, but other primitives like subdivision surfaces, particles or even blobby objects can be supported just as well.

Surface shading is then applied at the micropolygon level. The original Reyes paper used flat-shaded polygons with the surface shader evaluated at the polygon centers. This makes the numeric estimation of derivatives and surface normals trivial. Current production renderers usually perform the shading at the vertex positions so that Gouraud shading can be used. This allows for slightly larger micropolygons and reduces shading overhead. However, this also makes the estimation of variable derivatives slightly more complicated, especially on grid boundaries. We chose to stick to the flat-shaded method with our implementation. You can find more details on this in Section 5.2.

Reyes was designed for programmable shading from the beginning. Since micropolygon grids are highly regular and exhibit a large amount of locality, they are well suited to perform shading in a vectorized manner. Pixar even manufactured a specialized SIMD-based computer system to perform this task in the 1980s [1].

After surface shading, the micropolygons are ready for rasterization. The micropolygons are sampled in screen-space and the final color is updated for the given sample position. In the most basic approach, a z-buffer is used for hidden-surface removal. Multiple jittered samples can be used to remove aliasing artifacts (“jaggies”). More advanced implementations can use stochastic

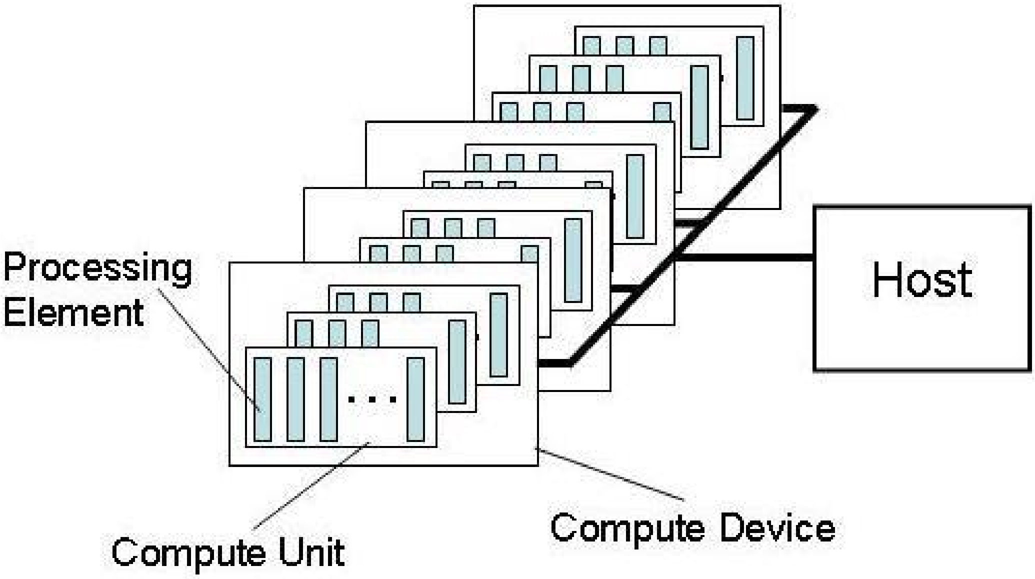


Figure 2.2: The OpenCL platform model. Picture from OpenCL specification [16]

rasterization for rendering surfaces with motion blur or adding depth of field. These advanced rasterization modes can be easily added since shading is separated from sampling, and all that needs to be rasterized are colored polygons with an optional alpha value. Scenes with complex transparency can also be rendered by using an A-buffer approach where a sample position keeps track of a list of fragments. Our implementation only implements simple z-buffered rasterization as described in Section 5.3.

2.2 General Purpose Computing on the GPU

This section is intended to give the reader a short overview on how general-purpose computing on the GPU works and to define the terminology used in this thesis. The specific vocabulary varies slightly between APIs. We are going to use OpenCL terminology, since this is what we used for our implementation.

An OpenCL system is called a platform. A platform is composed of a host and one or more devices as shown in Figure 2.2. The host is the main CPU process that sends compute tasks to the devices and controls memory transfers. The host portion of an application is programmed using a general-purpose programming language like C, C++, Java, or Python, making use of the OpenCL runtime API.

A device can be any of a large array of device types, among them GPUs, FPGAs, special accelerator modules, or even just a CPU backend. We will focus on GPUs for this thesis. These devices are programmed in the OpenCL C Language, a subset of C, which can be compiled at runtime to the target device. There exists a binary intermediate format called SPIR, which is

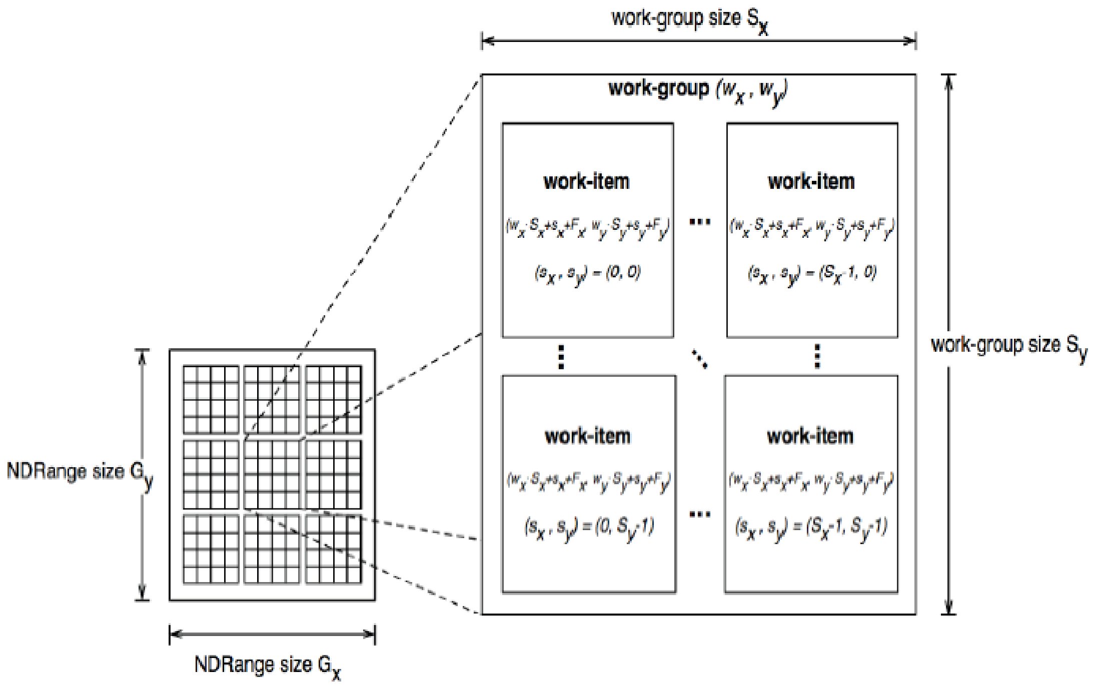


Figure 2.3: The work range an OpenCL kernel is divided into work-groups, which are themselves composed of work-items. Picture from OpenCL specification [16]

based on LLVM and can also be used for device programming.

Each device is composed of several compute units, which are themselves a group of processing elements, the smallest unit of computation. For a vectorized CPU device, a compute unit can be thought of as a single core, with the processing elements being the individual SIMD elements. In a modern GPU device, a compute unit is composed of a large number of SIMD or MIMD registers that get processed at once. Recent AMD GPUs use 64 processing elements per compute unit, for instance.

The host can submit compute tasks to devices by creating a command queue for a device and enqueuing memory transfers and kernel executions to it. A kernel is a function in a device program, which is executed in parallel over a large range of work-items, each having its own index. This index range or can be one-, two-, or three-dimensional and is itself evenly divided into smaller work-groups. This can be seen in Figure 2.3.

The entire range is processed by many compute units with each work-group being worked on by a single unit. Work-items within a work-group can synchronize to each other and share information. A work-group can be larger or smaller than the number of processing elements in a compute unit. However, for best performance its size should be a multiple of this number. There is usually a device-specific upper limit to the number of work-items in a work-group.

While kernels usually don't have access to host memory, there are several types of memory available within a device as shown in Figure 2.4. The most basic is perhaps global memory,

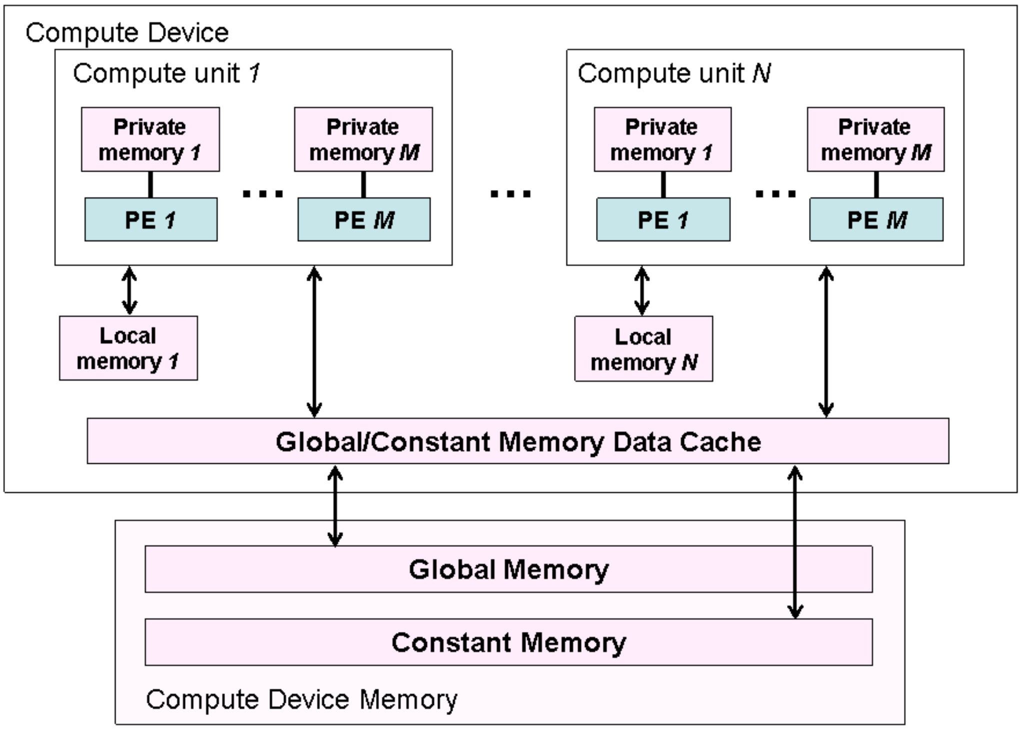


Figure 2.4: The OpenCL memory model. Picture from OpenCL specification [16]

which is shared between all compute units. This is also the only type of memory that the host can access by performing memory transfers or mapping buffers into host memory. Constant memory can be though of as a read-only variant of global memory, which can make use of more aggressive caching within the compute units.

Global memory is shared between devices, making memory objects allocated in it visible from all devices. However the memory coherency guarantees for concurrent kernels from different devices accessing the same buffer are rather loose. This allows keeping memory objects on a single device's memory in most cases, with device-to-device memory transfers being done implicitly where necessary.

Each work-group can allocate a small amount of local memory, which can be accessed by all work-items in the work-group. This allows communication between work-items within a work-group. Combined with the work-group synchronization function available in OpenCL C, this allows complex cooperation between work-items. Synchronization between work-items that don't share a work-group is generally not possible. The only way to make sure that all work-items within a kernel have passed a certain synchronization point is to wait for the entire kernel invocation to terminate and to start another.

Work-items can also allocate a small amount of private memory, which is only accessible to this work-item. The general variables defined within the kernel functions can also be thought of as part of private memory. In GPU devices, these values are usually stored in registers.

The total amount of local and private memory used by a kernel affects its runtime performance. This is because compute units have a fixed amount of memory and registers available and kernels which need only little memory allow them to keep more work-groups in-flight. By processing several work-groups at once, the compute unit's hardware scheduler can hide memory latency by rapidly switching between work-groups.

Recent hardware and the newest version of OpenCL also allows the sharing of memory objects between host and device memory. This is called shared virtual memory – or SVM – and was introduced in OpenCL 2.0. However, our implementation does not use this feature, since no stable OpenCL 2.0 driver was available for our hardware at time of writing.

CHAPTER 3

Related Work

3.1 Surface Tessellation

Owens et al. were the first to talk about the possibility and challenges of implementing Reyes on graphics hardware [26]. In their 2002 paper they compare the typical real-time triangle rasterization pipeline to an implementation of Reyes on the Imagine stream architecture, a high-performance programmable processor specialized for media applications. While this experimental architecture was not identical to current graphics processors, it still shared many similarities. Like current GPUs it consisted of a large number of ALUs which process data in parallel. The programming model for Imagine also was also already making use of the kernel concept. Their finding was that adaptive subdivision, which they implemented in a depth-first manner, was the major bottleneck in their renderer

In a very short 2003 paper Stephenson describes the implementation of Reyes on the Sony Playstation 2 game console [33]. This uses the vector units of the PS2's Emotion Engine processor for dicing and shading. The shaded grids are then sampled using the system's hardware rasterizer. Adaptive subdivision is apparently performed in a depth-first manner. The major limitations they mention is the limited available memory and the output quality of the hardware rasterizer due to coordinate rounding errors.

By transforming the typical depth-first recursive traversal of split surfaces into a breadth-first operation, Patney & Owens are able to parallelize adaptive subdivision for the GPU [28]. Their CUDA-based implementation processes all surfaces at each iteration. Each surface is bounded and a kernel decides whether it is diceable, requires further splitting, or can be culled. If splitting is necessary it is performed and the result is written to an output buffer. After this step the output buffer needs to be compacted, so that it can once again be used as input for the next iteration. This is done using parallel prefix sum and a copy kernel. Figure 3.1 outlines a single iteration of their subdivision algorithm.

While this performs well on the GPU, using a breadth-first traversal means that the peak memory consumption of this algorithm rises exponentially with the number of splits [11, 19, 35]. To alleviate this problem Patney & Owens suggest splitting the screen into a number of buckets

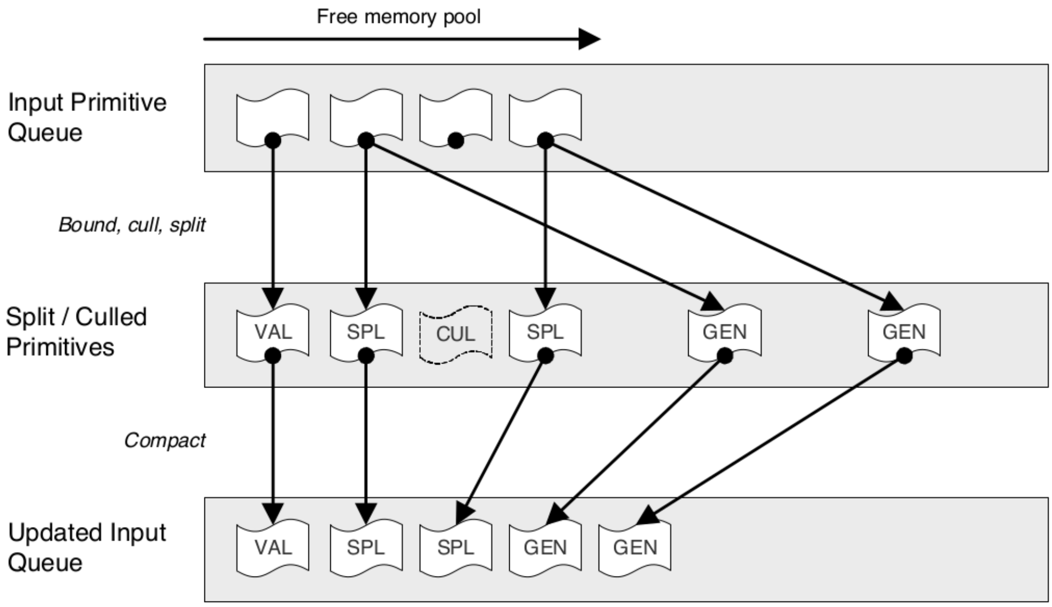


Figure 3.1: Schematic overview of the phases of breadth-first adaptive subdivision. Patney & Owens 2008 [28]

so that rendering is only performed at a part of the screen at a time. This reduces memory consumption, but obviously at the cost of parallelism. The per-bucket load and thereby the resulting memory consumption and performance is also non-trivial to predict. Sanchez et al. [30] also note the disadvantages of breadth-first scheduling (compared to other scheduling strategies) with respect to memory usage and locality. Nevertheless, several papers build on this method.

This thesis will present a generalization of the breadth-first adaptive subdivision algorithm that gives control over the maximum amount of memory used. Our method should work as a drop-in replacement for all applications that use breadth-first subdivision.

Zhou et al. use breadth-first adaptive subdivision as part of a full GPU-based Reyes renderer called *RenderAnts* [35]. While not real-time they are able to achieve interactive frame rates for simple scenes and is able to handle very complex scenes and shaders. Their renderer is also able to distribute the work-load over several GPUs and handle out-of-core textures using. It is implemented in BSGP, a general purpose GPU programming language of their design [14]. The out-of-core texture fetching is making use of BSGP's GPU interrupt mechanism based on a compiler technique that automatically splits kernels into several programs at interrupt points. RenderAnts uses dynamic scheduling to ensure bounded memory usage for fragment processing. However, no such bound is given for adaptive subdivision.

Patney et al. [27] use the breadth-first approach for crack-free view-dependent tessellation of Catmull-Clark subdivision surfaces, and Eisenacher et al. [9] adopt the same breadth-first approach for parametric surface subdivision, but also consider surface curvature, resulting in considerably fewer surfaces being created.

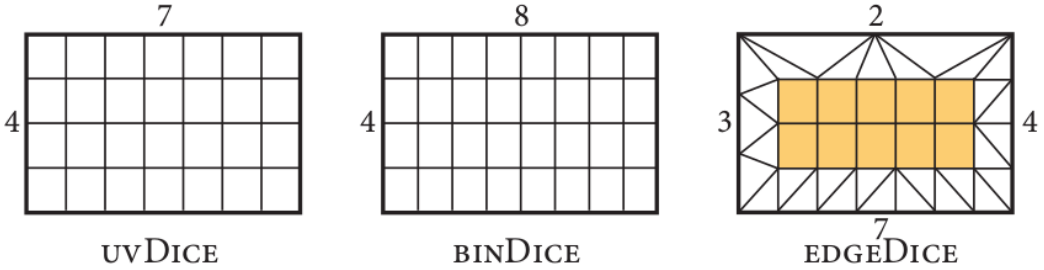


Figure 3.2: Example of different dicing patterns presented in DiagSplit paper. The classic Reyes approach is to choose tessellation factors once for each parameter axis (UVDICE). By restricting tessellation factors to powers of two (BINDICE) is an easy way to avoid surface cracks, however this leads to unnecessary over-tessellation. Both surface cracks and over-tessellation can be avoided by allowing separate dicing rates for all surface edges and the inside (EDGEDICE). Fisher et al. 2009 [11]

Tzeng et al. [34] consider adaptive subdivision from a scheduling point of view. They make use of persistent kernels and distribute the total work over many work-groups. To ensure load balance, they advocate a scheduling strategy based on work-stealing and work-donation. This approach has the advantage of avoiding host-device interaction for enqueueing additional iterations. However, while general memory consumption is greatly reduced with their approach, the peak memory usage remains unpredictable.

In a different application domain, Hou et al. consider the problem of memory-efficient parallel tree traversal during k-d tree construction [13]. With similar motivation to this work, they propose a partial breadth-first search traversal scheme that only evaluates a limited number of leaves in a tree.

Software-based Reyes renderers build a connectivity graph and add stitching geometry when the surfaces have been diced [1]. To do this, the renderer needs to keep the shaded micropolygon vertices available. This works in CPU-systems where virtual memory is plentiful and subdivision is performed in a sequential manner. Building and using such a data-structure in a highly parallel GPU environment is problematic.

Another method to avoid surface cracks is to restrict dicing rates to powers of two. That way adjacent grids can be made to agree on the tessellation level and in case of differing tessellation the grid edge with the higher tessellation factor can be truncated to the lower-tessellation edge. However this still results in T-junctions and results in over-tessellation by about a factor of two.

The dicing phase of Reyes maps to hardware tessellation supported on recent graphics APIs and GPUs [31]. This feature works well and is commonly used in current games. Hardware tessellation also allows the selection of separate tessellation levels for the inside and each boundary edge of a surface in order to avoid surface cracks. An example for this can be found in Figure 3.2 under the name EDGEDICE. In a system where no bound-and-split is performed, all that is necessary to avoid cracks between surfaces is for surfaces that share an edge to agree on the tessellation factor for this edge.

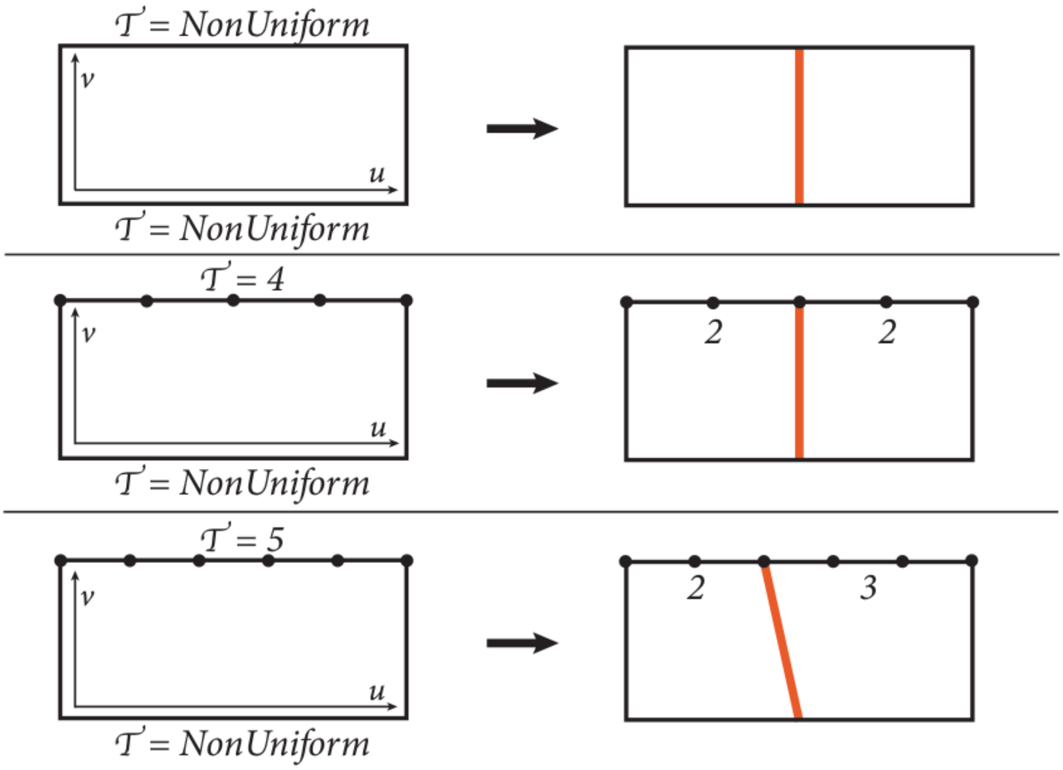


Figure 3.3: Split patterns described in the DiagSplit paper. In case an edge is labeled non-uniform or if it has an even dicing rate it can be split in half. If the dicing factor is odd, then the split interval must be trucated so that it aligns with one of the sample points near the edge's center. In this case the split line will be diagonal to the iso-parametric line. Fisher et al. 2009 [11]

Fisher et al. [11] present DiagSplit, a method for efficiently avoiding surface cracks during subdivision by combining the scheme used in hardware tessellation with Reyes. This allows crack-avoidance for hardware-accelerated Reyes without needing to keep track of surface connectivity while subdividing. However in order for integer tessellation levels between adjacent surfaces to align, it is necessary to take special care when performing split operations.

The algorithm works by first estimating the length of the edges. If it is above a certain threshold, it is marked non-uniform. If it is below this threshold, then the rounded edge length is used as the dicing rate of this edge. Now if the surface requires further split operations to be performed and a dicing rate has already been selected, then it might be necessary to subdivide the surface along a diagonal line in parameter space, so that the surface's new corners coincide with a vertex on the edge. This is illustrated in Figure 3.3.

Allowing splitting along non-isoparametric lines can be done by keeping track of the subsurface's corner points in parameter space. Once the surface is found to be diceable, it is transformed into micropolygons using the EDGEDICE pattern based on the estimated per-edge dicing rates. While this is an elegant solution to avoiding surface cracks, one slight draw-back of this is that surface sample points for shading aren't aligned in a neat iso-parametric grid anymore. This makes the calculation of differentials for things like texturing and normal estimation less straight-forward. This is especially the case for the vertices on the edge. EDGEDICE also requires, that surfaces can be evaluated at arbitrary positions in parameter space.

Fisher et al. also discuss the scalability issues of breadth-first subdivision and give this as a reason for their decision to implement their adaptive subdivision on the CPU using multithreading and balanced stacks. This gives excellent memory scalability and good locality, but does not scale well beyond a relatively small number of concurrent threads.

As mentioned the irregular grids created by the EDGEDICE pattern used in DiagSplit is problematic when numerically calculating differentials while shading. This is addressed by Burns et al. [3]. Their solution is decoupling the shading from surface evaluation so that surface shading still happens in a regular grid. The rasterizer will then look up the correct position in the shading grid for the point intersected by a sample. This also allows avoiding shading of hidden grid points by performing shading after rasterization and only evaluating grid points which are needed by generated fragments.

3.2 Micropolygon Rasterization

Hardware rasterization is robust and gives excellent performance when rendering typical triangular scenes. However as the size of polygons becomes smaller, this performance advantage quickly drops. This is because hardware rasterizers perform rasterization and shader evaluation in screen tiles. Even for micropolygons that only cover a single pixel, a complete screen tile needs to be evaluated. Figure 3.4 demonstrates this behavior. A tile size of 2×2 pixels is relatively small by hardware standards. 4×4 or 8×8 is more common. A micropolygon overlapping a single 8×8 tile would have a sample test efficiency of less than 2 percent.

One option to avoid this inefficiency is to sidestep the hardware rasterizer altogether and implement rasterization on the GPU as a software kernel. For general purpose triangle rasterizers this leads to a performance loss of about 50% compared to hardware rasterization [18]. However for micropolygons it is possible to get better efficiency than hardware rasterization by implementing a kernel specialized for rendering grids of micropolygons.

Fatahalian et al. implement the rasterization of micropolygons in a data-parallel manner [10]. They achieve good performance for micropolygons by transposing how the sampling loops are vectorized. Instead of only processing a single polygon per work-group and testing several sample points in parallel, several polygons are processed at once with each sample point overlapping the bounding box of a micropolygon being tested in sequence. This is justified, since micropolygons are so small that only a few sample points are considered candidates.

They also extend their rasterization to higher-order sampling for motion-blur and depth-of-field effects. For 4D or 5D space the number of sample points inside the bounding box of a micropolygon can quickly become very large, with only few of them actually intersecting the polygon. To avoid this, the time and lens dimensions are split into intervals and a separate bounding box is calculated and tested. This is shown in Figure 3.5.

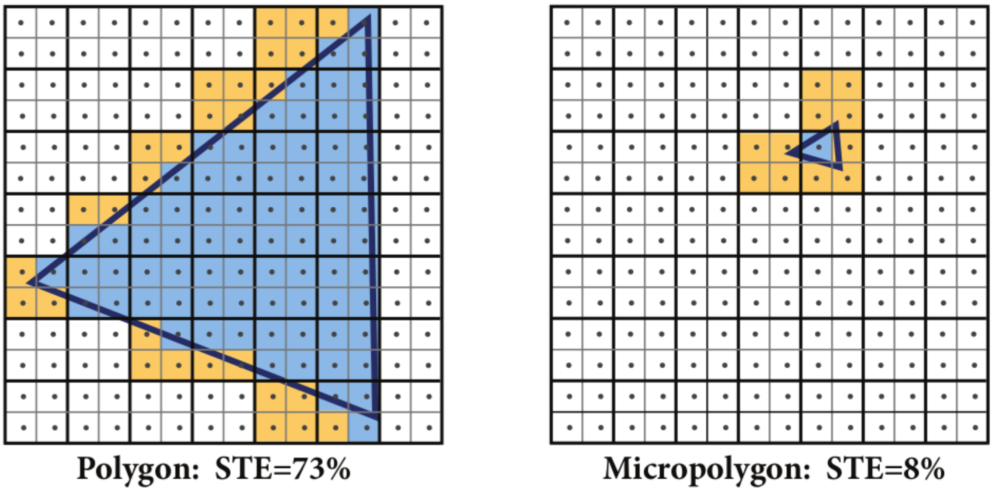


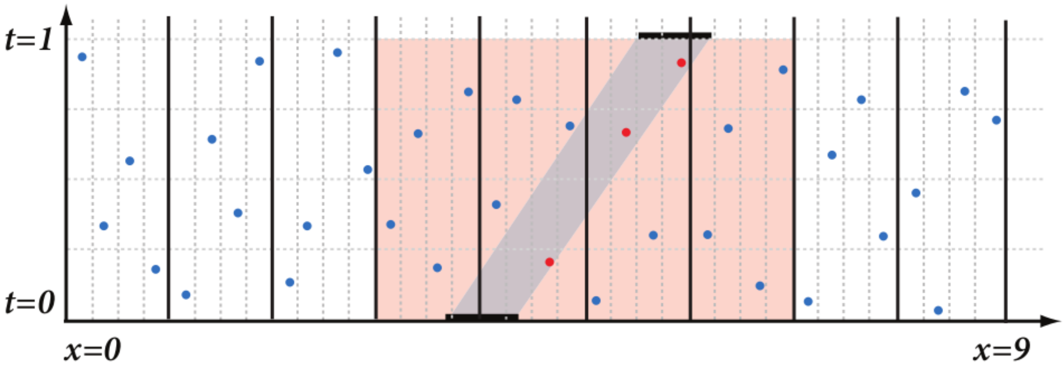
Figure 3.4: Hardware rasterizers find screen-tiles overlapping a polygon and then test each sample point within this tile for intersection. This approach works well for larger triangles, but is highly inefficient for micropolygons that only cover a single sample point. The values listed are the *sample test efficiency* (STE) which describes the percentage of samples tested that actually intersect the polygon. Fatahalian et al. 2009 [10]

One issue their paper doesn't discuss is the processing of resulting fragments. Their implementation also appears to be a CPU-based simulation of their algorithm, with benchmark results only showing relative execution times. The software rasterizer described in this thesis makes use of their finding that parallelization over many polygons gives better performance for micropolygons.

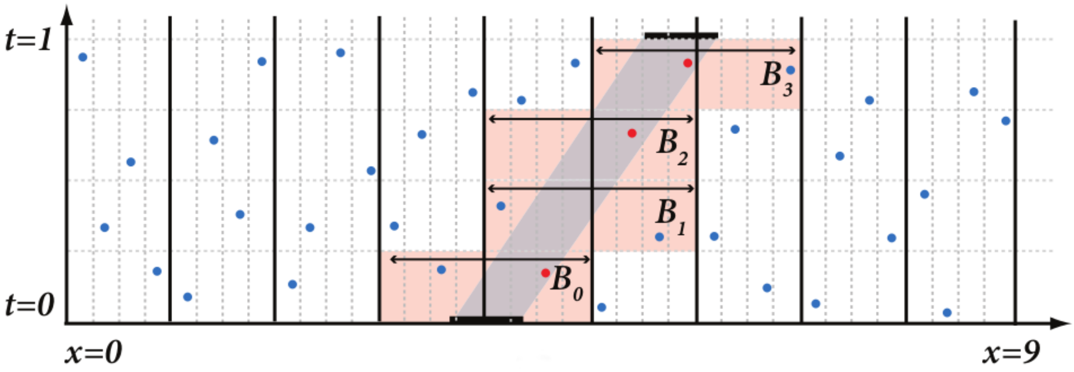
A CUDA-based micropolygon rasterizer is described by Eisenacher and Loop [8]. Their implementation takes a sort-middle approach where surface grids are assigned to tiles they overlap. For each of these tiles a work-group is started where all assigned grids are iterated over and tested. Each pixel of a tile is assigned to a work-item and all micropolygons within a grid are tested for intersection. While this is easy to implement, this approach should give poor performance due to the large number of needlessly tested polygons per pixel.

Their implementation is part of a larger paper about a full CUDA-based Reyes renderer by Loop and Eisenacher [19]. Bound-and-split is performed using the breadth-first approach. They mention the potentially high memory consumption of this method and subdivide surfaces in parameter space for this reason. They also describe a simple method for hiding surface cracks by displacing boundary vertices to cover them.

Brunhaver et al. [2]describe a hardware implementation of specialized micropolygon rasterization based on the method by described by Fatahalian et al. This is implemented as a specialized ASIC design. They found that specialized hardware could effectively rasterize micropolygons with motion blur and defocus and would only require a very small fraction of the overall die



(a) All points in the polygon's bounding box are tested.



(b) The time dimension is split into even intervals and separate bounding boxes are tested

Figure 3.5: Sampling of a polygon in the time dimension. Splitting the time dimension into separate intervals and sampling the covered bounding boxes of these sub-intervals helps reducing the total number of tested candidate samples. Fatahalian et al. 2009 [10]

area and power consumption of a GPU. This circuitry could be used as an alternative rendering path next to the regular triangle rasterization hardware. However no graphics card on the market supports this at the moment.

3.3 Real-Time Tessellation of Catmull-Clark surfaces

Catmull-Clark subdivision surfaces are one of the most commonly used graphics primitives in production rendering. They can be considered a generalization of bi-cubic B-Spline surfaces which allow for control meshes with arbitrary topology [4]. Surface meshes are adaptively refined with a fixed set of rules until they approach their limit surface. Figure 3.6 shows an example of this. This allows creating detailed detailed curved surfaces that can be easily animated. A later extension of this adds support for semi-smooth creases by [7].

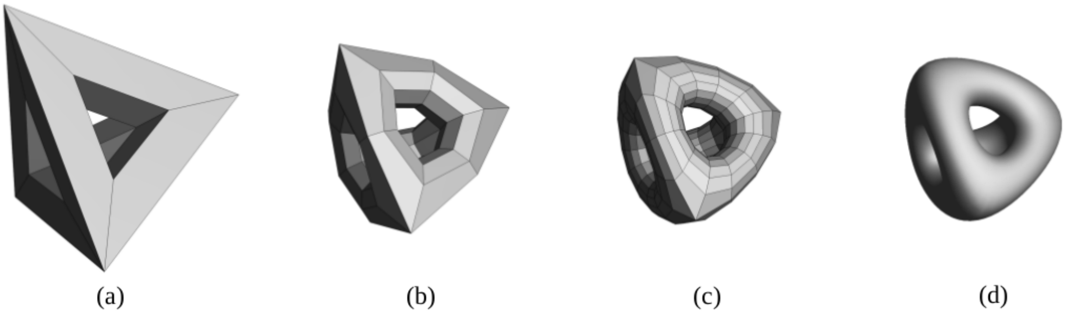


Figure 3.6: Catmull-Clark refinement stages a simple polygonal mesh. (a) is the base mesh, (b) is after one refinement step, (c) after two, (d) represents the limit surface after an infinite number of refinements. DeRose et al. 1998 [7]

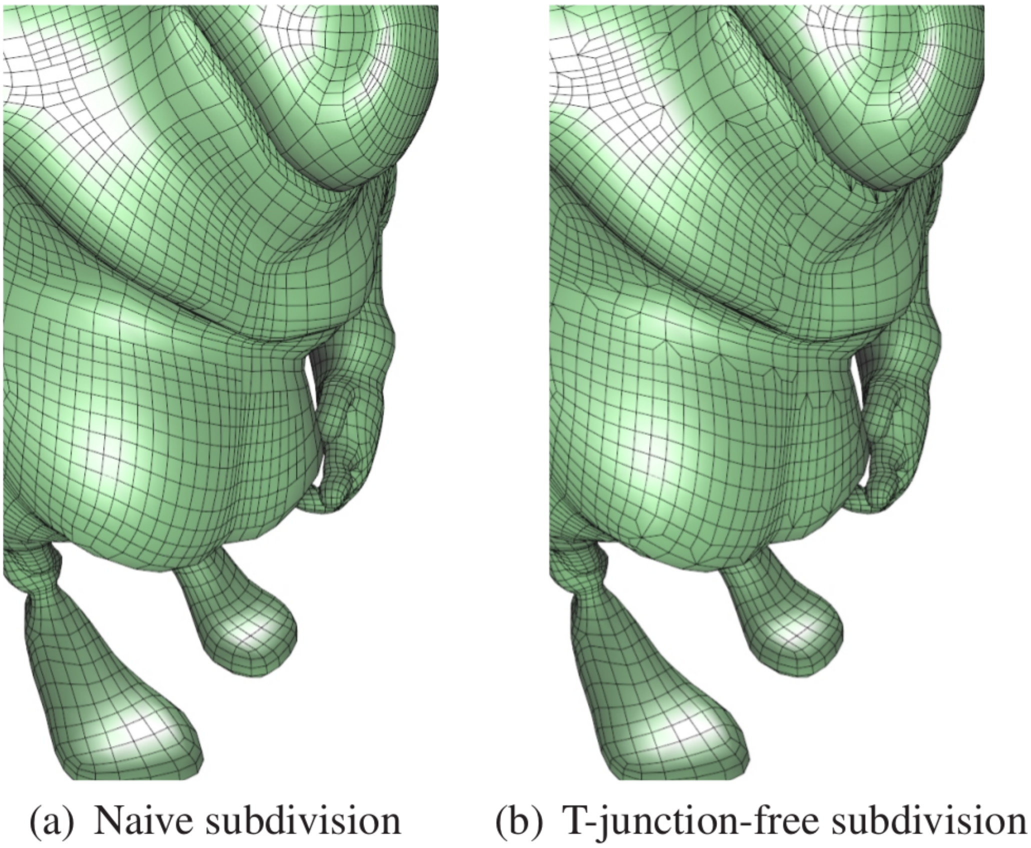

(a) Naive subdivision
(b) T-junction-free subdivision

Figure 3.7: Avoiding T-junctions when adaptively subdividing Catmull-Clark surfaces. Patney et al. 2009 [27]

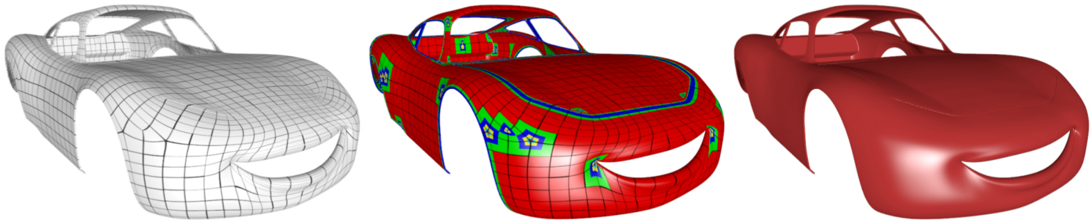


Figure 3.8: Example how a subdivision surface mesh can be evaluated. Faces sharing the same color can be directly evaluated as B-Spline surface. Note the contraction around extraordinary vertices and creased edges. Nießner et al. 2012 [24]

The ability to split Catmull-Clark surfaces allows for them to be integrated in a Reyes renderer. Bound-and-split is straightforward to implement since subdivision is supported out of the box and the bound can be computed from the convex hull of the control points. Dicing is more complicated, since direct evaluation at arbitrary parameter positions – while possible [32] – is usually not done due to the computational complexity and poor numeric stability. Instead, a surface is subdivided several times to create a grid of points, which are then projected onto the limit surface.

This approach is problematic on the GPU, due to its high memory usage. It also does not mesh well with DiagSplit, since it requires the ability to evaluate a surface at arbitrary parameter position. Because of this, several methods for approximating Catmull-Clark surface have been developed [17, 20, 21]. Newer variants of these work well, but are limited insofar that they don't exactly match the limit surface of actual Catmull-Clark surfaces. The renderer described in this thesis uses the Gregory patch approximation described by Loop et al. [21] for displaying Catmull-Clark surfaces.

Patney, Ebeida, and Owens present a method for crack-free view-dependent tessellation of Catmull-Clark surfaces based on the breadth-first subdivision algorithm [27]. They are able to avoid surface cracks from T-junctions by keeping track of the subdivision level in the neighborhood of a face (Figure 3.7). However their method performs adaptive subdivision all the way down to the micropolygon level without doing dicing. This leads to a rather high memory bandwidth consumption and general computational overhead.

Another method for the real-time tessellation of Catmull-Clark surfaces on the GPU was presented by Nießner et al. [24]. They avoid having to fully subdivide all surfaces by directly tessellating regular faces as B-Spline surfaces and only applying further subdivisions to faces containing an extraordinary vertex (Figure 3.8). This allows them to greatly reduce the overall memory consumption. In a follow-up paper, they discuss how semi-sharp creases can be handled efficiently [25].

CHAPTER 4

Adaptive Subdivision on the GPU

4.1 Introduction

The classic Reyes pipeline implements adaptive subdivision as a recursive operation. Reyes estimates the screen-space bound of a surface to decide whether the surface needs further subdivision or can be sent to the next pipeline stage for dicing. If further subdivisions are necessary, Reyes splits the surface and recursively calls bound-and-split on the new sub-surfaces. This process can be thought of as the depth-first traversal of a tree ("split tree"). While this is easy to implement on regular CPUs and requires minimal memory ($O(N+k)$, where N is the number of input surfaces and k is the maximum depth of the split tree), this approach is not suitable for the GPU since it is inherently sequential. Due to this exponential growth in memory consumption, the static preallocation of memory for this operation quickly becomes unfeasible.

Patney and Owens [28] parallelize the Reyes split phase by transforming this depth-first operation into a breadth-first traversal of the split tree. This way, a single iteration of the adaptive subdivision can be implemented using a parallel bound kernel, prefix sums, and a copy kernel.

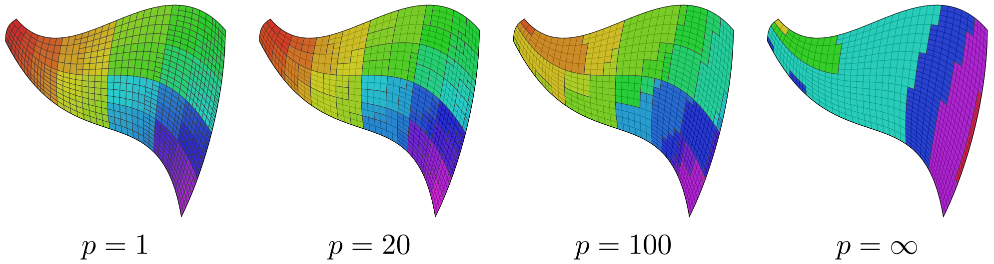


Figure 4.1: Comparison of evaluation order of surfaces for different batch sizes (p is the number of surfaces in a batch). Surfaces that are created in the same iteration are shaded in the same color. This shows the locality-preserving property of our subdivision algorithm: surfaces that are spatially close together are evaluated in the same iteration.

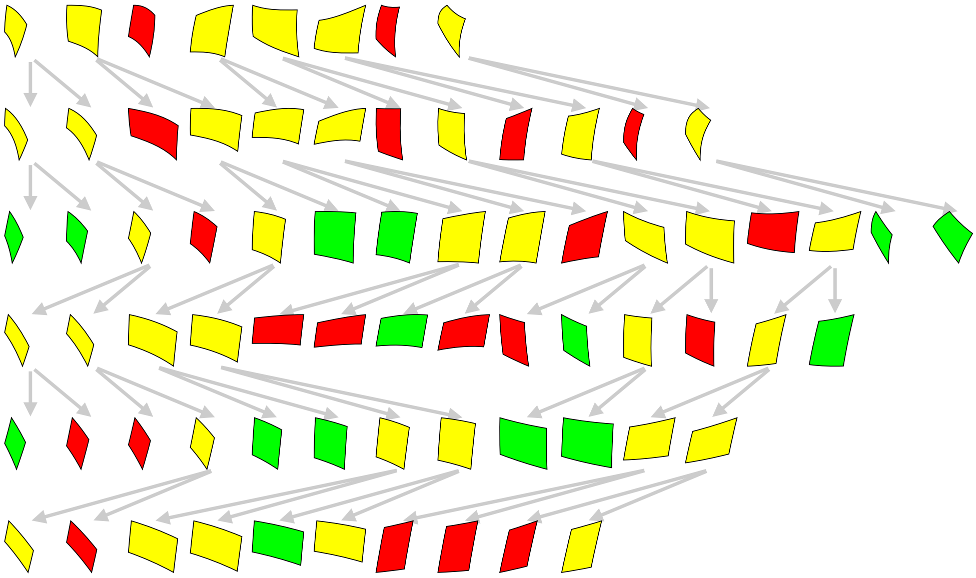


Figure 4.2: Schematic overview of breadth-first subdivision. Each row represents the state of the surface buffer during one iteration. Each surface can either be culled (red), split (yellow), or drawn (green). For each split surface in the previous iteration, two new surfaces are generated in the following iteration. This always happens for all surfaces in the surface buffer.

These are then iterated until all surfaces have been successfully bounded. Figure 4.2 gives an overview on how this approach works.

While this is simple to implement and yields excellent speedup, this approach suffers from high peak memory usage. Since all nodes of a single depth in the split-tree have to be held in memory, the worst-case memory consumption is the number of possible leaves of a binary tree of maximum depth k. This is $O(N \cdot 2^k)$, where N is the number of input surfaces processed at once. Due to this exponential growth in memory consumption, the static preallocation of memory for this operation quickly becomes unfeasible.

It is possible to split the input surfaces into several batches that are subdivided separately. This slightly reduces the worst-case memory consumption, but the overall memory consumption can still be very high, especially since the total memory consumption of the individual input surfaces varies highly due to perspective projection. The results section presents the test scene EYESPLIT, which has a very high memory requirement despite only containing a single surface. Furthermore, reducing the batch size also reduces the overall performance especially during the first few iterations.

4.2 Adaptive Subdivision with Bounded Memory

Instead, we propose an adaptation of this approach where the number of surfaces processed at a given iteration is limited by a constant value p. The buffer of surfaces is used as a parallel last-

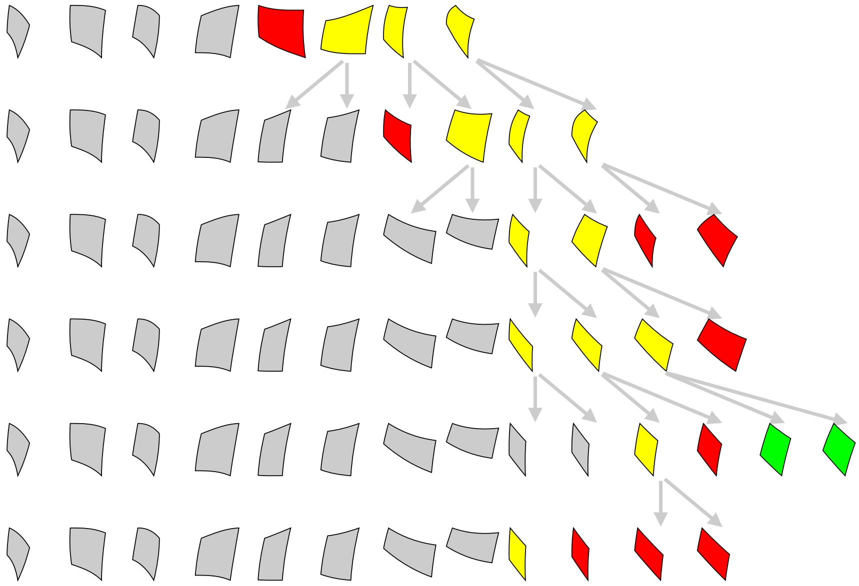


Figure 4.3: Schematic overview of how our memory-bounded subdivision operates. Unlike in Figure 4.2, the number of active surfaces at each iteration is constant (in this case, $p = 4$). The other surfaces are inactive and shaded in gray.

in-first-out data structure where surfaces are read from the end of the buffer, and any generated sub-surfaces are appended back to the end. By using this approach, we can bound the peak memory consumption by $O(N + p \cdot k)$. Figure 4.3 illustrates how this approach works.

Adding the batch size p as a tweakable parameter in the subdivision process allows us to balance between memory consumption and performance. Figure 7.2 shows the impact the chosen batch size and the amount of assigned memory have on the overall subdivision time. As the batch size increases, the subdivision time asymptotically approaches that of breadth-first subdivision. Our approach also preserves locality, as can be seen in Figure 4.1.

In our implementation, a *bound* kernel first copies the last p surfaces into a temporary buffer and estimates the screen-space bound for each of them. Depending on this bound, the kernel decides an action to be taken on this surface (*draw*, *split*, or *cull*), which is stored as a flag value in a separate buffer.

Whether a surface is ready to be drawn depends on the size of its screen-space bound, which is estimated by the kernel. Surfaces are culled when they are outside of the camera frustum or a surface has been split the maximum number of times. More advanced systems may also support occlusion culling, for instance by accessing a hierarchical depth buffer in GPU memory, however our implementation does not at the moment.

Procedural displacement also affects the screen-space bound of a surface. While there exist methods to efficiently estimate the bounds of displaced surfaces [22, 23], our renderer is limited to a configurable safety margin to avoid erroneous culling of displaced surfaces near the screen edge.

The temporary storage is necessary to avoid surfaces being overwritten by split surfaces before they have been read. This is not necessary in breadth-first subdivision, which uses a ping-pong buffer approach. While our temporary storage requires one additional write operation, the performance cost is minimal.

We then apply a prefix-sum operation to these flag buffers to calculate write locations. The *split* kernel checks the flag buffer and either copies the bounded surface into the output buffer or applies a split operation and places the resulting sub-surfaces at the end of the surface buffer.

For a surface P, the split-results P'_0 and P'_1 are placed at address $a_0 = S + f_c \cdot 2 + 0$ and $a_1 = S + f_c \cdot 2 + 1$ respectively, where S is the current size of the surface buffer and f_c is the prefix sum of the split flags. Using this particular order is necessary to prove the memory bound of our algorithm.

The flags accumulated by the prefix-sum operator are then used in a subsequent copy kernel to find the correct location for writing in the global-output and surface buffers. Surfaces remaining in the surface buffer will be further split by subsequent iterations of our subdivision algorithm, until the surface buffer is empty. The output surfaces of a single iteration are copied to an output buffer from where they are ready to be used by subsequent dicing and rasterization kernels.

In our implementation, the output surfaces are immediately consumed by subsequent pipeline stages. This way, we can make sure that the maximum number of surfaces that have to be processed in later stages is p. It is also possible to collect the output of several iterations before passing it on. However, collecting the entire output of the algorithm before passing it along further is not recommended, since this might once again lead to unbounded memory consumption due to the unpredictable amount of output surfaces.

Algorithm 4.1 gives an overview of our method. Note that this omits the handling of culled surfaces, which requires an additional flag buffer and prefix-sum operation.

Keeping the children of a surface that has been split close together also improves locality. Figure 4.4 shows the difference between placing the sub-surfaces in the order used by Patney and Owens as depicted in Figure 3.1 (NONINTERLEAVED) with our approach (INTERLEAVED).

Active surfaces are always read from the end of the surface buffer, and their potential children in the subdivision tree are always put back at that end again. As a result, and since the local order of the split products mirrors that of their parents, we can always expect that the surface buffer is sorted by subdivision level. This means that surfaces closer to the beginning of the buffer have had fewer subdivisions applied to them than those at the end.

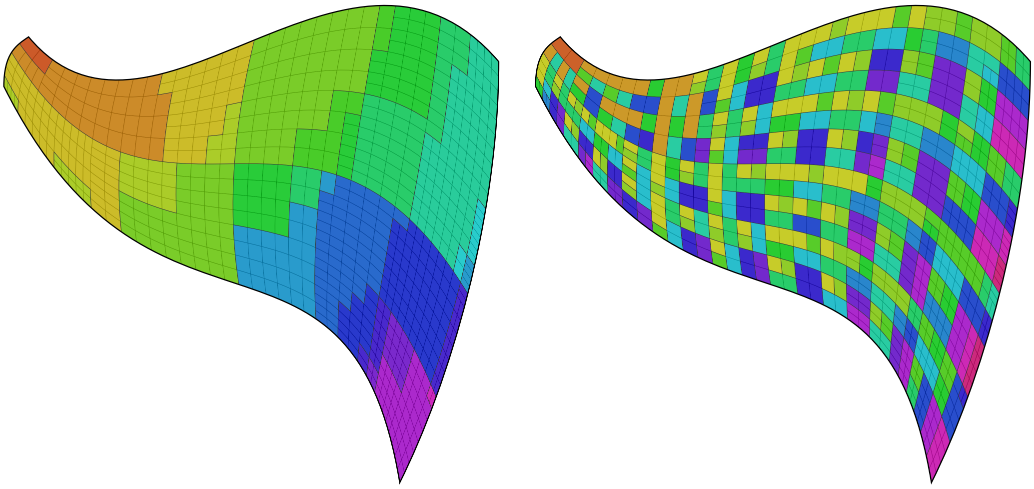
There can be at most p surfaces for any subdivision level in the buffer (safe for the root and top levels). This is because one iteration of the subdivision algorithm consumes p surfaces and appends at most $2p$ surfaces back to the buffer, which are guaranteed to have a higher subdivision level than the ones that were consumed. Since we are actively limiting the maximum allowed subdivision level to k and there can be at most p surfaces per subdivision level, we can make sure that there are at most $O(N + p \cdot k)$ surfaces in the buffer at any point in time.

Proof: Peak Memory Bound

To give a more formal proof for our asserted peak memory consumption of $O(N + p \cdot k)$, we consider the state of the ordered surface buffer $S_t = \{s_0, \ldots, s_n\}$ at every iteration $t \in \mathbb{N}$. If we

```
Data: maximal subdivision level $k$
Data: number of processors $p$
Data: surface buffer $S$ of size $N + k \cdot d$, containing $N$ input surfaces
Data: flag buffer $F$ of size $p$
Data: temp buffer $T$ of size $p$
Data: output buffer $O$ of size $p$
while $N > 0$ do
    $c \leftarrow min(p, N)$
    $N \leftarrow N - c$
    for $i \in \{0, \dots, c-1\}$ in parallel do
        $s \leftarrow S[N + i]$
        $T[i] \leftarrow s$
        if $s$ is bounded then
            $F[i] \leftarrow 0$
        else
            $F[i] \leftarrow 1$
        end
    end
    $G \leftarrow$ scan$(F)$
    $c_{split} \leftarrow G[c]$
    $c_{draw} \leftarrow c - c_{split}$
    for $i \in [0, c-1]$ in parallel do
        if $F[i] = 1$ then
            $j \leftarrow G[i]$
            $s'_1, s'_2 \leftarrow$ split$(T[i])$
            $S[N + 2j + 0] \leftarrow s'_1$
            $S[N + 2j + 1] \leftarrow s'_2$
        else
            $j \leftarrow i - G[i]$
            $O[j] \leftarrow T[i]$
        end
    end
    $N \leftarrow N + 2 \cdot c_{split}$
    draw the $c_{draw}$ surfaces in $O$
end
```

Algorithm 4.1: Adaptive-subdivision with a bounded amount of memory



INTERLEAVED NONINTERLEAVED

Figure 4.4: Illustration of the effect the placement order after split has on the locality of generated surfaces. Surfaces created during the same iteration share the same color. INTERLEAVED is the order described in this section while NONINTERLEAVED uses the order of the Patney and Owens paper.

can show that

$$\forall t \in \mathbb{N}_0 \colon \; \|S_t\| < N + p \cdot k \tag{4.1}$$

then the general memory consumption must also be bounded, since the size of all other buffers apart from S_t remains constant throughout execution. The split level $d\colon S_t \to \mathbb{N}_0$ gives the number of subdivisions that have so far been applied to a given surface. For example, for a fresh input surface s, $d(s) = 0$. If we split this surface into two sub-surfaces s_l and s_r, then $d(s_l) = d(s_r) = 1$. If we split s_l further into s_{ll} and s_{lr}, then $d(s_{ll}) = d(s_{lr}) = 2$ and so on. We also define the operator $D_i(S_t) = \|\{s \in S_t | d(s) = i\}\|$, which gives the number of surfaces in S_t that have subdivision level i. We will prove this memory bound via induction. To do so we will show that the following invariants are fulfilled at every iteration t:

$$\forall s \in S_t \colon d(s) < k \tag{4.2}$$

$$\forall s_i, s_j \in S_t, i < j \colon d(s_i) \leq d(s_j) \tag{4.3}$$

$$D_0(S_t) \leq N \tag{4.4}$$

$$\exists m \in \mathbb{N}_1 \colon \; \begin{array}{l} \forall i \in \mathbb{N}_1, i \neq m \colon D_i(S_t) \leq p, \\ D_m(S_t) \leq 2p - \sum_{j>k} D_j(S_t) \end{array} \tag{4.5}$$

(4.2) asserts that no surface in the surface buffer may have been divided more than k times. This can be ensured by always culling or drawing surfaces that have already received $k - 1$ subdivisions. (4.3) requires that the surfaces in the buffer are ordered by their subdivision level. (4.4) ensures that there can never be more than the initial number of surfaces with subdivision level 0 in S. Invariant (4.5) is the most important one. It requires that there always exists one positive subdivision level m so that all other positive subdivision levels $i \neq m$ must occur at most p times in S. m may occur up to $2p$ times, but only if there exist no surfaces with a higher subdivision level in the buffer. Otherwise the number of these is subtracted from $2p$ to give the number of allowed occurrences of m. The memory invariant (4.1) must be fulfilled when (4.2), (4.4), and (4.5) are fulfilled, since there can be at most k subdivision levels which occur at least once; among those, one can occur at most N times, one can occur at most $2p$ times, and the rest can occur at most p times. It should be easy to see that all invariants are fulfilled for S_0. Now we take a look at an arbitrary iteration t so that S_t fulfills all invariants. When we take the last p items from S_t, potentially split them if they have a subdivision level below $k + 1$, and append the split patches back at the end of the surface buffer, we get S_{t+1}, the state of the surface buffer in the next iteration. We can be sure that (4.2) is fulfilled for S_{t+1} since we take special care to never split surfaces that have already been split $k + 1$ times. Similarly, (4.4) should be easy to see, since we can only increase the subdivision level of surfaces, so $D_0(S_{t+1})$ can only be smaller than $D_0(S_t)$. Since we know that S_t is sorted by subdivision level, we know that the p surfaces we took for subdivision must have the maximal amount of subdivisions for S_t. By splitting them, we can only increase this, so the split surfaces we put back at the end are guaranteed to be larger than all of the remaining ones. And thanks to the specific order described in Section 4.2, we can ensure that if two split surfaces are in a specific order relative to each other, then the split products must also be in that order, only with their subdivision level incremented by one. Thus (4.3) is fulfilled for S_{t+1}. To see that (4.5) is fulfilled, we must consider the special subdivision level m for S_t. The range of surfaces with subdivision level m can start at most $2p$ elements away from the end of the buffer. At most p surfaces of this range can be outside the last p elements of the buffer. If we now remove the last p surfaces P and split them in any way, the resulting subsurfaces must have a subdivision level greater than m. This means that the number of remaining surfaces with tessellation level m must now be $\leq p$. m is no longer a special depth in S_{t+1}. What remains to be seen is that there can be at most one new level m' for which there are more than p surfaces in S_{t+1}, and that it must be at the end of the buffer. Since what remains of the old level m has $\leq p$ surfaces after the iteration, and the other surfaces of S_t are known to have $\leq p$, the new m' can only belong to the surfaces created from P. For any combination of non-overlapping ranges in P, there can be at most one subset that contains more than $p/2$ surfaces. If this range is x surfaces away from the end, then it can contain at most $p - x$ surfaces. If we now subdivide this range and maintain the order of the split products for this range, they must form the subdivision level m', fulfilling invariant (4.5). $\square$

4.3 Storing Intermediate Surfaces in Work-Group Local Storage

Section 4.2 describes an implementation that stores all surfaces (including intermediate surfaces) in GPU global memory. To reduce the amount of host interventions, we also explored a variant of our algorithm that keeps control within kernel work-groups. Instead of having a single surface buffer in global memory, each work-group keeps its own surface buffer in local on-chip memory. Single iterations of the subdivision algorithm described in Section 4.2 are performed in a loop within the work-group. Communication and flow control of threads within the group is done using local memory, work-group-local prefix sums, and barriers. Surfaces that have been successfully bounded are transferred to an output buffer in global memory. With this approach, no explicit host intervention is necessary to start another iteration of the subdivision algorithm, thus global memory bandwidth is reduced.

Instead of having to read and write surface data to and from global memory, the only times when surfaces need to be transferred out of local memory is during the initial reading of input surfaces and when writing the final bounded surfaces to the destination buffer. We can store the entire intermediate surface buffer in work-group local memory, since the amount of necessary memory for this derives from our memory bound $O(p \cdot k)$, where p is the work-group size of this kernel. In practice, this should be set to the native SIMD width. As an example, for recent AMD GPUs, this is 64. Let's say we allow a maximum subdivision level of 20. We would have to allocate enough shared memory to store $64 \cdot 20 = 1280$ surfaces. If one surface requires 24 B of storage, the total amount of necessary shared memory would be 30 KiB. If this exceeds the amount of available work-group local memory, then a small amount of global memory can be allocated for spill buffers.

Our implementation uses a single global input queue to store work and performs well for our test cases. While no load-balancing is necessary between threads within a work-group, it is still possible that some work-groups run out of work more quickly than others. In this case, a load balancing strategy similar to the one described by Tzeng et al. [34] could be used, but this was not implemented. If the output buffer is full and there is still work to be done, then the kernel must stop operation and return control back to the host so it can render the generated patches. The content of the work-group local surface buffer needs to be backed up to global memory so that it can be recovered once operation is resumed.

While this can reduce the overall subdivision time for very small data-sets, we found that using work-group storage local is usually outperformed by our regular algorithm for reasonably sized scenes. The main reasons for this are the missing load-balancing and the relative complexity of the persistent kernels, which is a source of overhead and impedes the GPU's hardware scheduler. This will be further explored in Section 7.1.

CHAPTER 5

Dicing, Shading, and Rasterization

Once adaptive subdivision has been performed, the surfaces have been subdivided to have a screen-space bound smaller than the configured value and are ready to be tessellated into grids of polygons. After this, they can be shaded and rasterized to generate the final image. One option at this point is to perform these operations using a standard graphics API like OpenGL or Direct3D. Another option is to implement these components as OpenCL kernels and only use OpenGL to display the final result. The reason why one might want to do this is that, while the rasterization hardware used in modern graphics APIs has excellent performance for larger polygons, it quickly becomes inefficient for smaller ones like the pixel-sized ones we are generating. This is due to the way rasterization is implemented in hardware, where the smallest unit for testing triangle coverage usually is an 8×8 sceen tile. For polygons that only cover a single pixel, this would mean that 63 out 64 compute threads are wasted.

Dicing and Shading is, for the most part, a straightforward vectorization of the basic Reyes algorithm. The partitioning of grids into 8×8 blocks and the way backface culling is performed is based on our own findings. The structure of the rasterizer is based on the findings by Fatahalian et al. The way fragment compositing is performed is based on our own sort-last design using tile-locking.

5.1 Dicing

Dicing is implemented as a kernel that evaluates the surfaces at uniform intervals at a fixed resolution. The kernel is called for a three-dimensional work-range with the x and y dimensions being the position in the surface grid and the z dimension giving the surface number. The maximum value of x and y is therefore the dicing resolution. Work groups are flat layers with a depth of one so that the individual threads can make use of memory caching when reading control-point data from the surface buffer. The dicing rate can have a larger size than the maximum work-group size supported by an OpenCL implementation. In that case the grid will be evaluated in tiles.

Surface information is stored in two parts. The actual control-point information for the unsubdivided surfaces, which remains unchanged during subdivision, and a buffer of surface ranges, which contain a surface ID, as well as an isoparametric sub-range of the $[0,1] \times [0,1]$ parameter interval in the original surface. This sub-interval is encoded as two 2D points p_{min} and p_{max}. The dicing kernel reads this information and calculates the exact parameter position from these values and its x and y global position.

Once the kernel thread has determined its location on the patch surface, it evaluates the surface patch at this position as shown in Figure 5.1. All threads in a work group operate on the same surface and there are no potentially diverging control structures used in the kernel. We can therefore be certain that all threads in a work-group access the same control-point information at the same time. This should mean that only a single memory transfer per control-point should need to be necessary on recent graphics hardware. The base functions used for surface evaluation depend on the type of surface used. For our implementation this can either be cubic Beziér surfaces (16 control points per surface) or Gregory surfaces (20 control points per surface). Different `dice` kernels are generated for these function types, and the application part picks the correct one on demand.

This is the time where procedural surface displacement might happen. After the final points have been calculated, they are projected into pixel coordinates using a supplied projection matrix. These pixel coordinates are converted into fixed-precision integer numbers. This is done because this is a more robust format for rasterization. All raster operations will be performed in fixed-point arithmetic. Both the three-dimensional surface point, the two-dimensional projected points, as well as the depth value of each vertex are stored in an output grid buffer.

5.2 Shading and Back-face Culling

While the dicing kernel is called once for each grid vertex, the shading kernel is executed once for each polygon between the vertices. The main task of the shading kernel is to calculate the surface color for the individual polygons. The result of this shading operation is then stored in a global color buffer. A renderer that supports programmable shading would need to generate a separate shade kernel for each material type used. The `shade` kernel as well as the following `sample` kernel are set up to operate on 8×8 tiles of polygons within the grids (Figure 5.2). These units we will call *blocks* from this point forward. If the grids produced by the dicing kernel are larger than this, then they are processed as several blocks. We have chosen the 8×8 size because 64 is the native SIMD width of current AMD graphics processors.

Our renderer performs per-face shading using a normal calculated from the eye-space vertex positions of the polygons. The subsequent `sample` kernel then just uses this constant color when drawing the polygon. While this simplifies the construction of our renderer and makes it easier to correctly handle procedural displacement, this is not usually the approach taken in production rendering systems. Instead, the surface is shaded at each vertex of the grid and the calculated color is interpolated over the polygon's area using Gouraud shading. In practise, this allows reducing the necessary polygon density by about a factor of two and therefore this is something that should be done for a production-ready system. However, we chose to restrict our renderer to flat shading for simplicity and robustness. Our surfaces still appear to be perfectly

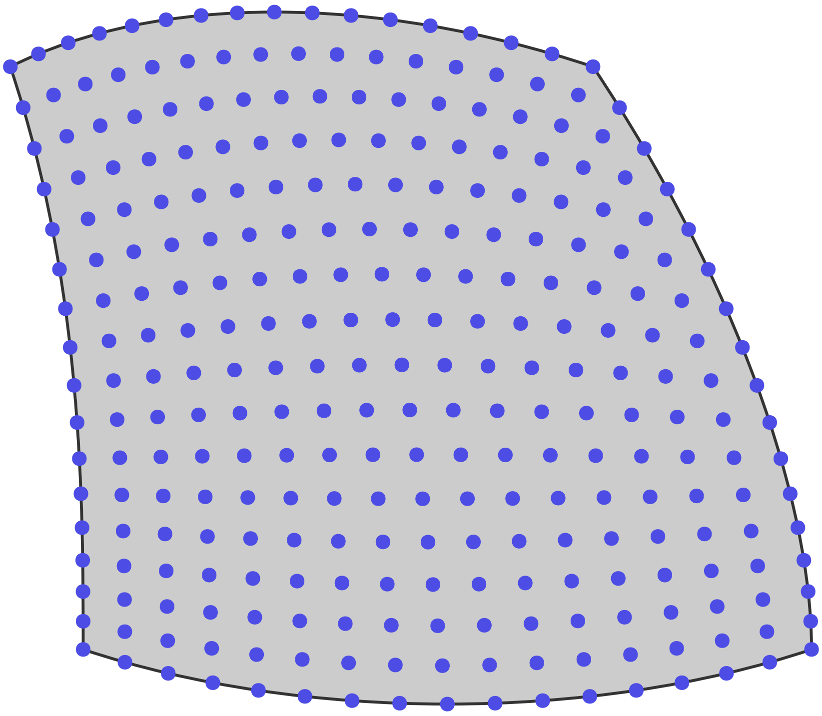


Figure 5.1: Surface evaluation in the dicing kernel. The parametric surface patch is evaluated at positions in a regular grid. The number of samples is 17 per axis to get a 16 × 16 grid of polygons.

smooth, and arbitrary displacement can be applied without having to recalculate the surface normals explicitly.

Apart from this, the `shade` kernel is also the place where grids are checked if they face away from the camera. This is done by each thread in the work-group checking the screen-space winding of its assigned polygon. The result of this is collected in an atomic work-group-local Boolean flag. If all surfaces are back-facing, the block can the culled and will not be shaded or rasterized. OpenCL 2.0 introduces the work-group wide voting function `work_group_all()`, which could also be used for doing this.

The `shade` kernel also has the task of calculating the screen-space bounding rectangle for each block. This is also done using work-group local atomic variables. Every thread determines the minimum and maximum screen coordinate for its polygon and updates the local variables using the built-in `atomic_min()` and `atomic_max()` functions. The resulting bounding rectangle is saved in a per-block buffer and will be used by the `sample` kernel when determining the framebuffer tiles that need to be considered during rasterization.

While it may seem more intuitive to calculate the bounding rectangle in the `dice` kernel, this is problematic since some of the grid vertices belong to several shading blocks.

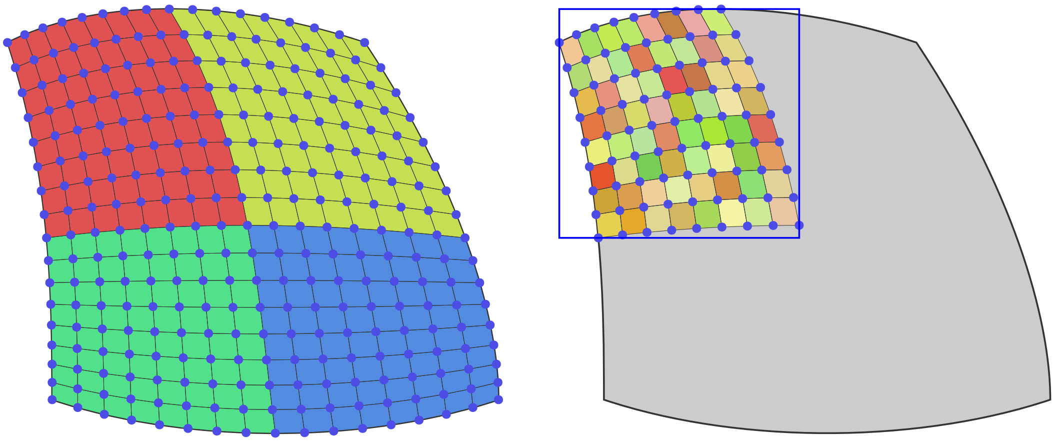


Figure 5.2: Grids are separated into 8×8 tiles, called *blocks*, which are shaded separately. The `shade` kernel also calculates the bounding rectangle for its block.

5.3 Rasterization

Once the position and final shade of the micropolygons has been determined and the bounding rectangle of the individual blocks has been calculated, the surfaces are ready to be rasterized. This is the task of the `sample` kernel, which calls a work group for each block.

To avoid unstructured global access patterns and the requirement for complex locking, the rasterizer will first work on a work-group local 8×8 canvas tile, which will be written to global memory after rasterization is done. Since we use z-buffering, it is necessary to lock a tile before writing the canvas contents to global memory. Other work-groups must wait for their turn if they want to access the same tile during this. The kernel finds the tiles that need to be considered for this by reading the bounding rectangle of the block.

Unlike general graphics hardware, which tests the intersection of a single polygon against many pixels, our kernel assigns a single thread to each polygon. This is more efficient since we know that one micropolygon can only overlap a small number of sample points. Every polygon iterates over the sample points overlapping its bounding box. The sample point is then tested for polygon intersection using their edge equations. The quadrangular polygons are split into two triangles for this. This approach is based on the paper by Fatahalian et al. [10].

If a polygon intersects a sample point, the depth value of the polygon will be interpolated for this position. This depth value is then compared against the current depth for the sample stored in the local canvas tile. If the depth value is smaller, then the sample is shaded in the polygon's color. This test-and-set operation needs to happen in an atomic fashion. Our implementation uses per-sample locks based on `atomic_xchg()` for this.

While most literature uses sort-middle rendering for software rasterization on the GPU [8, 10, 18], a sort-last approach is used in our implementation. This should avoid load-balancing issues and avoids the need for intermediate storage for bucket lists of grids covering a tile. Sort-middle rendering would have the advantage that all fragment processing and compositing could

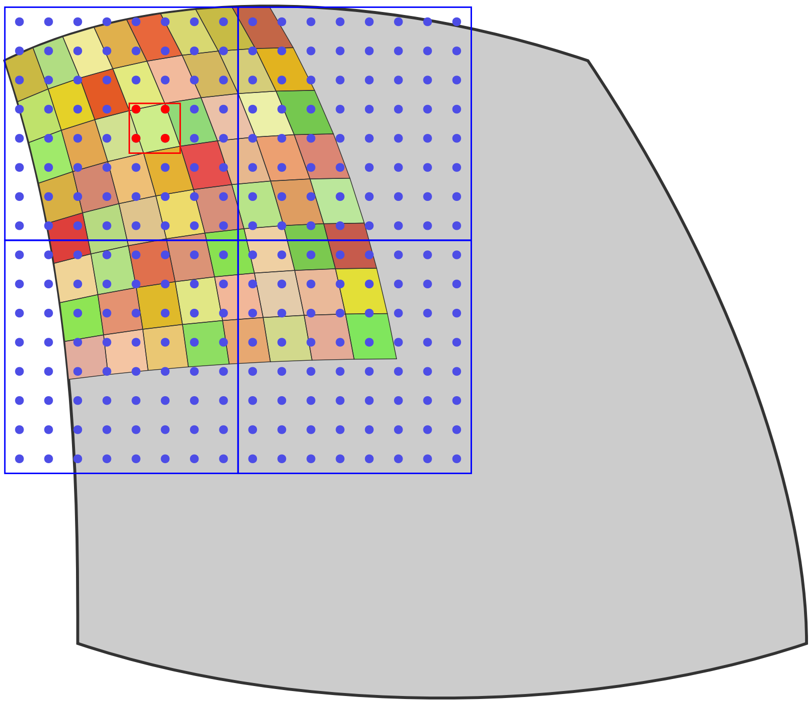


Figure 5.3: The `sample` kernel iterates over every framebuffer tile and checks the pixels within the bounding box for each micropolygon.

be serialized and performed locally. Our approach instead requires locking, which can cause a significant overhead for complex scenes as will be discussed in Chapter 7.

Algorithm 5.1 gives an overview on the `sample` kernel's structure. Figure 5.3 illustrates the units the kernel is operating on visually.

After all surfaces have been rasterized in this manner, the framebuffer is ready to be displayed on screen. This is done by copying its content to the OpenGL back buffer and performing a buffer swap. Ideally, we would want `sample` to be able to write directly to the OpenGL back buffer, but this is currently not possible.

```
Data: $8 \times 8$ block of polygons $B$
Data: work-group local $8 \times 8$ canvas tile $C$
for framebuffer tile $T$ overlapping $B$ do
    clear $C$
    for polygon $p \in B$ in parallel do
        calculate bounding rectangle of $p$
        for pixel $t \in T$ overlapping $p$'s bounding rectangle do
            if $t$ intersects $p$ then
                lock $t$
                update the color of $t$ in $C$
                unlock $t$
            end
        end
    end
    lock $T$
    update $T$ with the content of $C$
    unlock $T$
end
```

Algorithm 5.1: Pseudo-code for `sample` kernel.

CHAPTER 6

Implementation

For this thesis, we have implemented *Micropolis*, a real-time Reyes renderer written in C++11 and OpenCL C. Micropolis is open-source and freely available at `https://github.com/ginkgo/micropolis`.

6.1 Source Code Overview

The general folder structure of the Micropolis source code looks as pictured in Figure 6.1. The C++ source code can be found in the `src` directory. The folder `kernels` contains OpenCL kernel source code. `shaders` is the location of OpenGL shader files. The `tools` folder contains various tools for code generation, format conversion, and performance testing, which are for the most part written in the programming language Python.

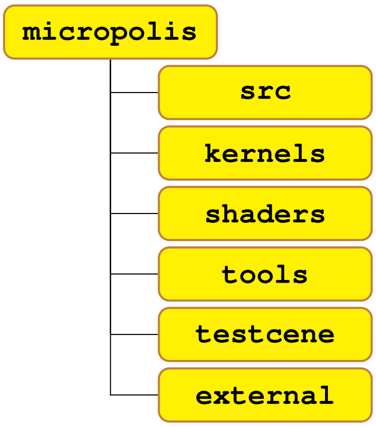


Figure 6.1: Overview of the top-level folder structure of Micropolis source code

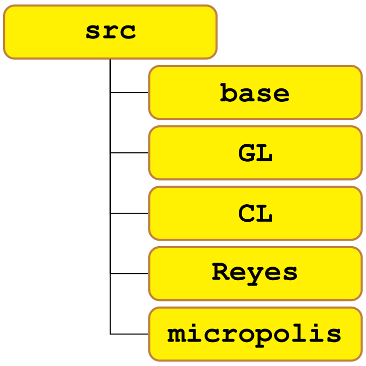


Figure 6.2: The contents of the `src` folder.

`external` contains the OpenCL header files and the header-only 3D math library GLM [5], which is used throughout the code base. The `testscene` folder is where various test scenes in our own file format along with their original Blender files are located.

Micropolis makes heavy use of source-code generation. OpenGL bindings are generated using `flextGL`, a Python-based tool that parses the OpenGL XML specification files and generates source code that is compiled along with our project. For configuration-file parsing, we are using another Python tool that reads XML files describing the configuration values and generates code for parsing configuration files and command-line arguments. This makes it very easy to add additional configuration values during development and testing.

Another use case for source generation is the `mscene` file format,which we use for loading scene data. For this we use the Cap'n Proto [15] interchange format, which takes a format description and generates parser code for various languages including C++ and Python. This is useful since it allows us to write and manipulate `mscene` files using Python while loading them with C++ with very good performance.

The C++ source folder itself is structured into five subfolders as shown in Figure 6.2. Apart from the `base` folder, which is the location of basic utilities and common header files, each of these folders represents a C++ namespace which is more or less independent from the others. `GL` contains wrapper classes for OpenGL concepts like shaders, textures, and buffer objects. `CL` does the same for OpenCL.

The `Reyes` namespace contains the major bulk of the implementation. This is where the Reyes algorithm, or at least the C++ part of it, is implemented. It makes use of both `CL` and `GL`.

The `micropolis` namespace can be thought of as the application side of Micropolis. It contains the program entry point and takes care of window creation, device setup and user interaction. This is also where scene loading and management is implemented. `Reyes` and the other namespaces are used from here.

6.2 Class Overview

This section will give a more detailed overview of the individual packages and the classes they contain.

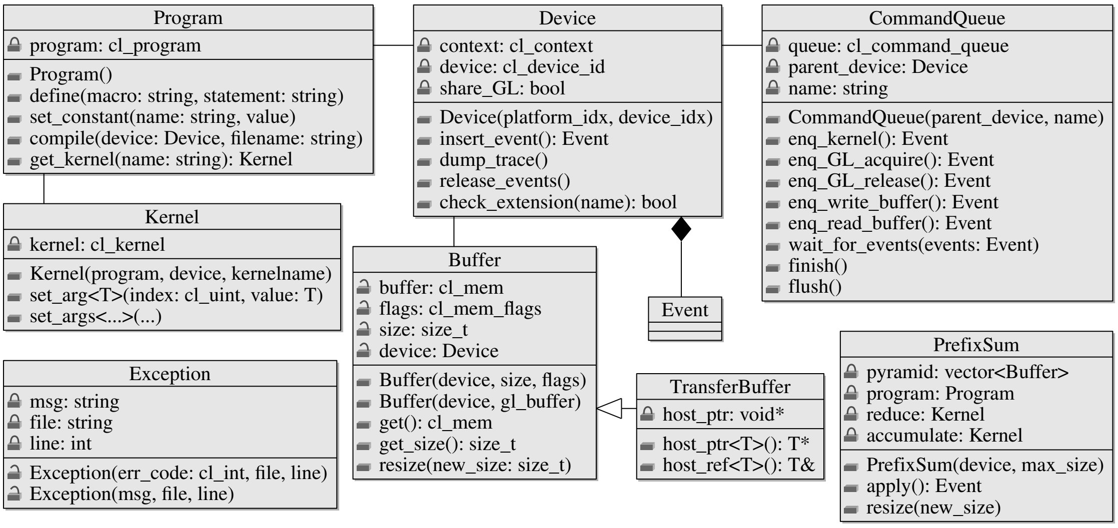


Figure 6.3: Classes in the `CL` package.

The `CL` Package

This package provides an object-oriented C++ wrapper to the OpenCL C API. `CL` wraps concepts like devices, command queues, kernels, or memory buffers in C++ classes for easier development. There now exists a semi-official C++ wrapper for OpenCL 1.2 supported by Khronos, but this was not available at the time of writing, and many features of `CL` are not supported by it. Figure 6.3 gives an overview of the classes of this package. The interrelations of these classes often mirror those of the respective OpenCL concept.

The first thing that a program using `CL` usually has to do is to create an instance of the `Device` class. This wraps an OpenCL context and a device into a single combined object. Multiple devices with a shared context are not supported. The constructor takes the platform and device index and sets up the associated OpenCL objects. If the program is configured to use OpenGL buffer sharing, this will also be set up with the current OpenGL context, if possible. After construction, applications can query a Boolean flag to check if OpenGL buffer sharing is supported by the device. Apart from being the C++ representation of OpenCL devices, this class also keeps track of `Event` objects, which will be discussed later.

OpenCL memory objects can be handled using the `Buffer` object. To allocate memory on an OpenCL device, an application simply needs to call the constructor with a device, the required buffer size in bytes and a `cl_mem_flags` value to define the buffer's accessibility. This creates buffer objects that can be used by the various wrappers in `CL`. An alternative constructor that takes an OpenGL buffer object handle to create a shared buffer also exists. There exists a specialized `TransferBuffer` class that inherits from `Buffer`, which has to be used when doing host-device memory transfers or when mapping memory. This subclass adds a host buffer that can be used as source and destination for read and write operations. If a platform supports zero-copy memory, then this host pointer should map directly to the contents of the OpenCL buffer.

To create executable OpenCL kernels, a program first needs to load and compile an OpenCL C program from which it can then extract the kernel objects. This is wrapped in the `Program` object. The simplest application of this class is to create an instance of it and call `compile()` with the selected device and the file name of the program's source code as parameters. This will look for the source code in the configured kernel folder and compile it for the given device. The OpenCL compiler supports preprocessor includes and is configured to look in the kernel folder for header files. After a program has been compiled, the wanted `Kernel` object can simply be created by calling the `get_kernel()` method with the kernel's name in the program. Another feature of `Program` is the ability to define compile-time constants and macros, which are appended at the beginning of the source code before compilation. This is useful for setting global parameters that don't change throughout the program's life-time.

While the `Kernel` object itself cannot be used for invocation, it is needed for setting kernel parameters. The class supplies the generic `set_arg()` and `set_args()` methods for doing this. `set_arg()` takes a parameter index as the first argument, and the passed argument value as the second. Template specialization is used to determine the correct action for the passed argument. If it is a primitive type, then `clSetKernelArg()` will be called with the data as-is. In case the argument is a `Buffer` object, then the wrapped `cl_mem` object will be passed to OpenCL. Since calling `set_arg()` with an index for each argument is verbose and prone to copy-paste errors, `Kernel` also supplies the variadic template method `set_args()`. This takes an arbitrary number of arguments and will call `set_arg()` for each of them with an incrementing index.

Objects of the `CommandQueue` class are needed to actually execute kernels on a device and start memory transfers. They are created from a device and a name, which is only used for debugging. Once a command queue object has been created, it can be used to call kernels, enqueue memory transfers, or acquire or release buffers from OpenGL sharing. These operations won't happen immediately but will be enqueued to be performed as soon as possible. The methods will finish immediately, returning `Event` objects that can be used to check if an operation has finished. The `wait_for_events()` method can be used to wait for certain events. Applications can call the `finish()` method to make sure that all operations up to this point have been performed. Command queues often collect several commands before submitting them to the device for performance reasons. The `flush()` can be used to make sure that the commands up to a given point have been submitted. This can be necessary for fine-grained host-device synchronization.

Events are a concept used throughout OpenCL for synchronizing various kernel invocations and device transfers at a fine-grained level. They are also used for querying profiling information after an event's associated operation has been performed. This is also `Event`'s function here. Every enqueue call to `CommandQueue` takes an `Event` object as its last argument. The OpenCL driver has to wait until all operations associated with the passed events have been finished before performing an operation. However, unlike the OpenCL C API, where events only represent a single operation and enqueue operations take an array of `cl_event` handles, our C++ wrapper only requires a single `Event` object. Instead, Events can be combined using the `|` operator to create `Event` objects representing a conjunction of the individual events.

The place where events are created and kept track of is the `Device` class. This is necessary to have a single location to acquire profiling traces if necessary. The list of events in `Device` grows over time. An application has to call the `release_events()` method to clear this list. This usually happens at the end of each frame for Micropolis. The `Device` method `dump_trace()` can be used to write a trace with precise timing information for each device operation to the file-system.

The OpenCL C API is designed in such a way that each operation immediately returns a value indicating if the operation was successful or if there has been an error. While this is good design for a C API, checking all operations for success can become tedious quite quickly. The `CL` package supplies an `Exception` class to alleviate this problem. All internal C API calls are checked and an exception is raised in case of an error. The constructor of `Exception` also takes care of converting the integer error flag into a human-readable message. Having a single piece of code that is called in case of an error is also useful for placing a debugger break-point. This makes finding bugs a lot easier since the compiler stops at the exact location where an error has been detected.

The `PrefixSum` class does not wrap a specific OpenCL concept. It is a utility class that implements the important prefix-sum operation. This class loads the necessary OpenCL kernels and allocates the buffer pyramid needed for this operation on a device. `kernels/prefix_sum.cl` contains the OpenCL C code for the kernels used by `PrefixSum`. The `apply()` operation enqueues the necessary succession of kernels to compute the prefix sum for a given input array.

The `GL` Package

Like `CL` for OpenCL, this package provides C++ wrappers for the OpenGL graphics API. Figure 6.4 gives an overview of `GL`'s classes. It should be easy to see that this only exposes a very limited part of OpenGL's functionality.

`Shader` wraps shader objects. The compiler takes a shader name and searches the configured `shader` in the Micropolis source tree for the shader files with this name, compiles them and links them into an OpenGL program object. The shader directory is automatically checked for vertex, fragment, geometry, and tessellation shader files, and the relevant features are activated on demand. This can then be used by binding it with `bind()` and performing any OpenGL draw call. Uniforms can be set using the generic `set_uniform()` method. This works in a fashion similar to `Kernel::set_arg()` in that template specialization is used to find the correct behavior for the passed argument. `Shader` also supports the binding of shared buffer objects using `set_buffer()`.

There exists a specialized `ComputeShader` class that inherits from `Shader`. This is used for compiling and dispatching OpenGL compute shaders. These can be thought of as the OpenGL equivalent to OpenCL kernels. `ComputeShader` mainly adds the `dispatch()` methods which are called by the application to dispatch compute tasks. Uniforms and shared buffer objects are set using the methods exposed in `Shader`.

OpenGL buffer objects are wrapped in the `Buffer` class. A buffer object is allocated in the class constructor and can be used immediately. The class supplies methods for manipulation of the buffer content(`send_data()`, `read_data()`) and for binding them for a particular purpose in the OpenGL pipeline(`send_data()`, `read_data()`). The naked buffer handle can also be extracted using `get_id()`. This is useful for setting up buffer sharing with OpenCL,

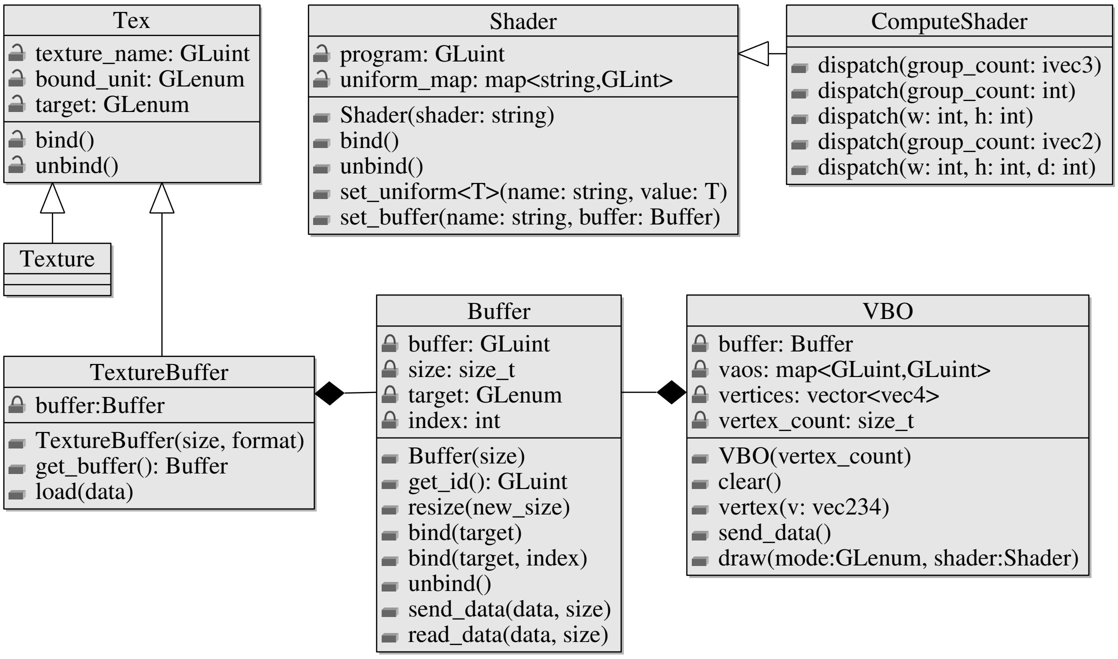


Figure 6.4: Classes in the `GL` package.

where the constructor of `CL::Buffer` needs this handle. The Buffer class also supplies a templated `bind_all()` function which helps with binding a number of buffers to an indexed target by picking the individual indices for us.

Both regular textures as well as texture buffers are implemented as sub-classes of the `Tex` base class, which take care of texture-name creation and keeping track of used texturing units for binding and unbinding(). The classic `Texture` class implements this class. This supports one to three-dimensional textures with arbitrary pixel formats and allows the configuration of typical filtering and wrapping modes. `TextureBuffer` wraps the concept of texture buffers, flat buffers that can be bound like textures in OpenGL and used by shaders for both reading and writing.

The `VBO` class is a very simple utility class for setting up a vertex buffer object and sending geometry data to the GPU. All that Micropolis needs to draw is a single screen-filling quad, but since core OpenGL dropped support for immediate-mode rendering, the geometry data for this has to be loaded using the regular VBO+VAO approach. `VBO` helps with this.

The `Reyes` Package

This package is where the major algorithmic part of the Micropolis rendering code is implemented. Figure 6.5 gives an overview of the classes in `Reyes`.

Applications mainly interact with the `Reyes` package through the `Renderer` interface, of which `RendererCL` is the major implementation. The exposed interface is rather compact. Parametric surface meshes can be loaded into device memory using the method `load-`

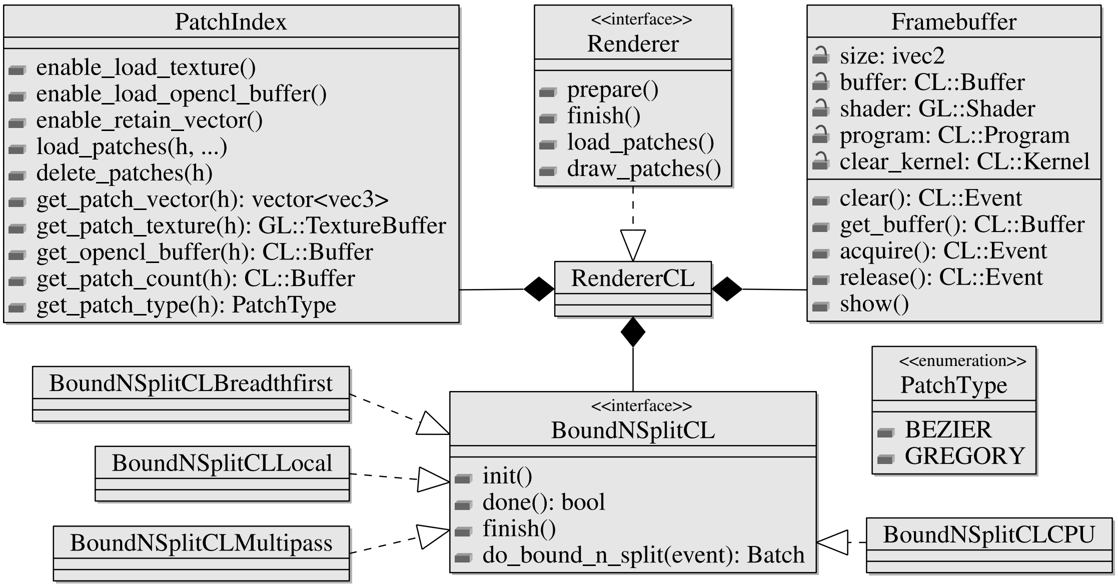


Figure 6.5: Classes in the `Reyes` package.

`_patches()`. The application can supply an arbitrary `void*` handle, which will be used for referring to the given mesh when rendering. Two types of surface mesh are supported: One is composed of cubic Beziér surfaces and the other is using Gregory patches for approximating Catmull-Clark subdivision surfaces. The renderer differentiates between these two types of geometry using the `PatchType` enumeration type.

There is a simple protocol to using `Renderer`. Once the renderer has been instantiated and the necessary geometry data has been loaded, the application can start rendering by calling the `prepare()` method. This will prompt the renderer to do the necessary preparations before rendering. After this, an arbitrary number of calls to `draw_patches()` can be performed. This method takes a handle, a model-view matrix, a projection, and the object's material color and will render the surface mesh with the defined transformation to the screen. After all scene objects have been rendered, the scene rendering is ended by a call to `finish()`, which will wait for completion of any pending draw operations and perform any necessary final steps, like blitting the finished frame to the screen. In code, this might look as shown in Algorithm 6.1

`RendererCL` is a concrete implementation of `Renderer` that uses OpenCL to display the surfaces. This class sets up the necessary OpenCL buffers and kernels in its constructor. Management of surface meshes is handled in a separate `PatchIndex` object. `RendererCL` contains an object of type `Framebuffer`, which is the target for render operations. Adaptive subdivision is encapsulated in objects of the `BoundNSplitCL` interface, which is used by `RendererCL` to subdivide surfaces before rendering.

`PatchIndex` keeps track of surface meshes in the application. Different implementations of `BoundNSplitCL` and `Renderer` may need the patch-data to be stored in a different way (in host memory, as OpenCL buffer, or as an OpenGL texture buffer). Because of this, `PatchIndex` supplies the methods `enable_load_texture()`, `enable_load-`

```
renderer.prepare();

for (obj : scene.objects) {
  mat4 mv = obj.model * scene.view;
  Projection p = scene.projection;
  renderer.draw_patches(obj.handle, mv, p, obj.color);
}

renderer.finish();
```

Algorithm 6.1: Drawing the objects in a scene using an instance of `Renderer`.

`_opencl_buffer()`, and `enable_retain_vector()`, which have to be called after `PatchIndex` has been constructed. This affects the way a mesh is stored when `load_patches()` is called. Rendering classes can then call the relevant getter methods with a handle to access the relevant data, if available. Alongside this, `PatchIndex` also keeps track of the number of patches in a surface mesh and the used patch type.

The `Framebuffer` class encapsulates an OpenCL buffer that can be written to by `RendererCL`. It also implements methods for clearing the framebuffer (`clear()`) and displaying the rendered surface (`show()`). This is done using a shared OpenGL buffer if possible. When OpenGL sharing is not supported, Framebuffer will read the framebuffer contents back onto host memory before writing it back to the OpenGL framebuffer. The framebuffer is organized in 8×8 pixel tiles, which are themselves organized in rows. The blit operation is performed using the `tex_draw` OpenGL shader, which will calculate the correct location in the flattened framebuffer texture for each pixel location.

As mentioned before, `Renderer` uses `BoundNSplitCL` to perform adaptive subdivision. This is encapsulated in an abstract interface so that the performance of different implementations can be compared. Like the `Renderer` interface, this is intended to be used in a certain manner as shown in Algorithm 6.2. To subdivide a mesh, the renderer first needs to call `init()`, which will initialize the working buffers and internal counters. After this, the renderer can call `do_bound_n_split()` to subdivide a chunk of the total data which can then be diced and rasterized. This has to be done as long as there is work left, which can be checked using the `done()` method. The nested `Batch` structure, which is returned by `do_bound_n_split()`, is a structure that contains the number of output patches, references to the OpenCL buffers to read from, and an Event that will be triggered when subdivision is done (since the subdivision may work in an asynchronous fashion). At the very end, when all surfaces have been rendered, the renderer will call `finish()`, which allows the subdivider to perform some clean-up operations.

There exist several implementations of `BoundNSplitCL`. The most basic one is `BoundNSplitCLCPU`, which does subdivision on the CPU. All surfaces are kept track of in an `std::vector`, and subdivision happens in a simple push-pop manner. This was the first subdivider to be implemented and it proved useful for testing purposes.

`BoundNSplitCLCPU` is designed to perform subdivision while the renderer backend is

```
bound_n_split.init(handle, mv, p);

while (!bound_n_split.done()) {
  Batch batch = bound_n_split.do_bound_n_split(ready);

  ready = process_batch(batch);
}

...

// After all meshes have been drawn
bound_n_split.finish();
```

Algorithm 6.2: Using `BoundNSplitCL` for subdividing surface meshes.

still busy drawing the previous batch. Several output buffers are allocated that are used so that the subdivider can write to an output buffer while the dicing kernel can read from the old output buffer. This setup gives good render performance for very small scenes since both the CPU and GPU can be utilized and the host never has to wait for results from the GPU when issuing commands. However, even for slightly larger scenes or screen resolutions, the CPU tends to become the limiting factor when processing surfaces.

`BoundNSplitCLMultipass` implements the adaptive subdivision algorithm described in Section 4.2. This uses three OpenCL kernels as well as `CL::PrefixSum` to perform the algorithm. At first, the `init_ranges()` kernel sets up the surface buffer with the initial unsubdivided surfaces. Each iteration then calls `bound_kernel()` on a configured number of surfaces at the end of the buffer. This will calculate the screen bound of the surfaces and creates decision flags depending on this bound. These flags are then accumulated using prefix-sums, and the `move()` kernel then uses these flags to split the surfaces and copy them to their correct position at the end of the surface buffer. These kernels can be found in the file `kernels/bound_n_split_multipass.cl`.

Two variants of the bound-kernels are loaded to differentiate between Beziér and Gregory surfaces. This is done by loading and compiling the OpenCL program twice, each time with a different preprocessor definition for the surface evaluation function. The subdivider knows which kernel to call, since the patch type of a mesh is always known in advance. The same preprocessor-based method is used in other places in the code where OpenCL kernels need to handle a specific patch-type.

The `BoundNSplitBreadthFirst` class is the basic breadth-first subdivision algorithm, which always processes all surfaces in device memory. The memory consumption of breadth-first subdivision is unbounded. This is why `BoundNSplitBreadthFirst` has to resize the internal buffers on the fly. In case there is not enough device memory available, the program will fail. Apart from this, the source code of this is similar to `BoundNSplitCLMultipass`.

`BoundNSplitLocal` uses persistent work kernels and work-group-local memory for subdividing surfaces. The algorithm for this is described in Section 4.3. This only requires two kernels. One for setting up the input buffers and distributing the surfaces to the different work groups (`init_range_buffers`) and one for doing the adaptive subdivision (`bound_n_split`). The `bound_n_split` kernel performs the subdivision within the individual work-groups and copies the fully bounded patches to the output buffer. In case the output buffer is full, the bound-and-split kernel copies its local surface buffer back to the input buffer and stops operation so that the renderer can consume the surfaces in the output buffer. Surface subdivision will be resumed afterwards. One practical flaw in the implementation of this is that there is no load balancing between work-groups. This can lead to very poor performance relative to the other implementations for some use cases.

Once the surfaces have been fully bounded, the renderer calls a `dice` kernel, which evaluates the surfaces at the points of a regular grid and saves the locations in a grid-buffer. These grids are then processed in a subsequent `shade` kernel, which calculates the shade for each polygon in the surface grid and computes the tight screen-space bounds for each 8×8 block within the grids. The actual rasterization happens in the `sample` kernel, with a work group for each of these blocks. The `sample` kernel iterates over all framebuffer tiles that a block overlaps and finds the pixels that are covered by the individual micropolygons. Each micropolygon is assigned its own thread within the work-group for this. Once the coverage of a tile has been processed, the kernel locks this tile in the framebuffer and writes its local results to it. There used to be a sort-middle variant of this where blocks are assigned to tiles, with a work group for each tile. This didn't need tile locking, but it suffered from sub-optimal load-balancing and the performance was worse overall.

6.3 The `micropolis` package

This package doesn't contain many classes. Its most important content is the `main.cpp` file, which contains the program's entry point. This loads the configuration files sets up the OpenGL and OpenCL contexts, loads the scene file, creates the renderer class and performs the render loop. Apart from this, the only significant part of `micropolis` is the `Scene` class, which is a very simple scene representation composed of a list of meshes, object instances with world transforms, light sources, and a camera objects. Figure 6.6 gives an overview of the `Scene` class and its helper structures.

Our scene description is very simple and only supports the bare necessities at the moment. Loading and saving of scenes happens over a binary format described as a Cap'n Proto schema. This description file can be found in `src/micropolis/mscene.capnp`, with the only major difference being that objects refer to their meshes by name instead of pointer. Its structure mirrors that of `Scene`. The different objects are kept track of in flat lists. We use a separate list for each object type because this made it easier to mirror the behavior of the Cap'n Proto schema, which is somewhat limited in its expressiveness. There can be several cameras within a scene, with one of them being set as active. By default this is the first camera in the list. The `draw()` method takes a `Renderer` as argument and uses it to draw all objects in the scene. The implementation of this looks more or less as the code fragment shown in Algorithm 6.1.

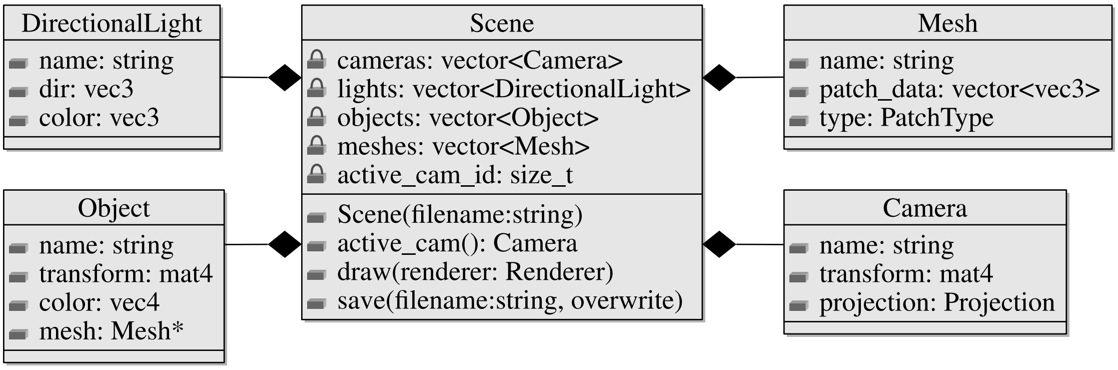


Figure 6.6: The `Scene` class.

6.4 Supporting Infrastructure

Micropolis uses a number of scripting tools for compiling and profiling the application. The one that interacts with the source code the most is the configuration-file generator script. This Python program generates a configuration file loader written in pure C++ from a simple XML description of configuration values. The result is a class file with a single global instance that is accessible from every point in the application. Each package has its own set of configuration values contained in their own class (`CLConfig`, `GLConfig`, `ReyesConfig`, `Config`). The XML files for these can be found in the source folders of the individual packages. There exists a separate configuration file for each of these in the source root folder (`cl.options`, `gl.options`, `reyes.options`, `micropolis.options`). These parameter values can also be supplied as a command line arguments.

The ability to modify many configuration values from the command-line is used heavily by the profiling programs in the `tools` folder for testing the performance of various combinations. Micropolis can be configured in such a way that it will dump performance metrics to a file and exit after a defined number of frames. The benchmark programs execute the Micropolis application several times to collect the necessary data.

The `testscene` folder contains the Python script `mscene_export.py`, which can be used in Blender to convert the current scene into an `.mscene` file. The method described by Loop et al. [21] is used to convert subdivision surface meshes into Gregory patches. This takes advantage of using a Cap'n Proto schema for defining our scene file format. Reading and writing scene files in Python is as easy as importing the schema and calling `Scene.read()` because of this.

6.5 Usage

Once the program has been built, it can be started by simply executing the generated binary in the root folder. The user can change the configuration in the described configuration files or locally override configuration values with a command line argument.

```
> ./micropolis --parameter="value"
```

It would take too much space to discuss all configuration values. What follows is a list of some of the more useful ones.

- `opencl_device_id`
  Two integers giving the platform and device index for the OpenCL device that is to be used.

- `window_size`
  Size of the render window in pixels. (Two integers)

- `input_file`
  The scene file that will be loaded.

- `bound_n_split_limit`
  The screen-space bound all surfaces will be subdivided to in pixels.

- `bound_n_split_method`
  Defines which subdivision method should be used. One of CPU, MULTIPASS, BREADTH-FIRST, or LOCAL.

- `reyes_patch_size`
  The resolution all surfaces are tessellated in. Ideally, this should be the same value as `bound_n_split_limit`.

- `reyes_patches_per_pass`
  This controls the batch size used for subdivision and rasterization. (Single integer)

This will create a simple window displaying the selected scene (Figure 6.7). The user can move the camera through the scene using the WASD keys and change the viewing direction by dragging the mouse cursor. When holding the left SHIFT key, the camera moves faster.

PGUP and PGDOWN control the subdivision bound. F9 dumps a kernel trace file. F12 saves the scene with the current camera view to a file. PRINT creates a screenshot.

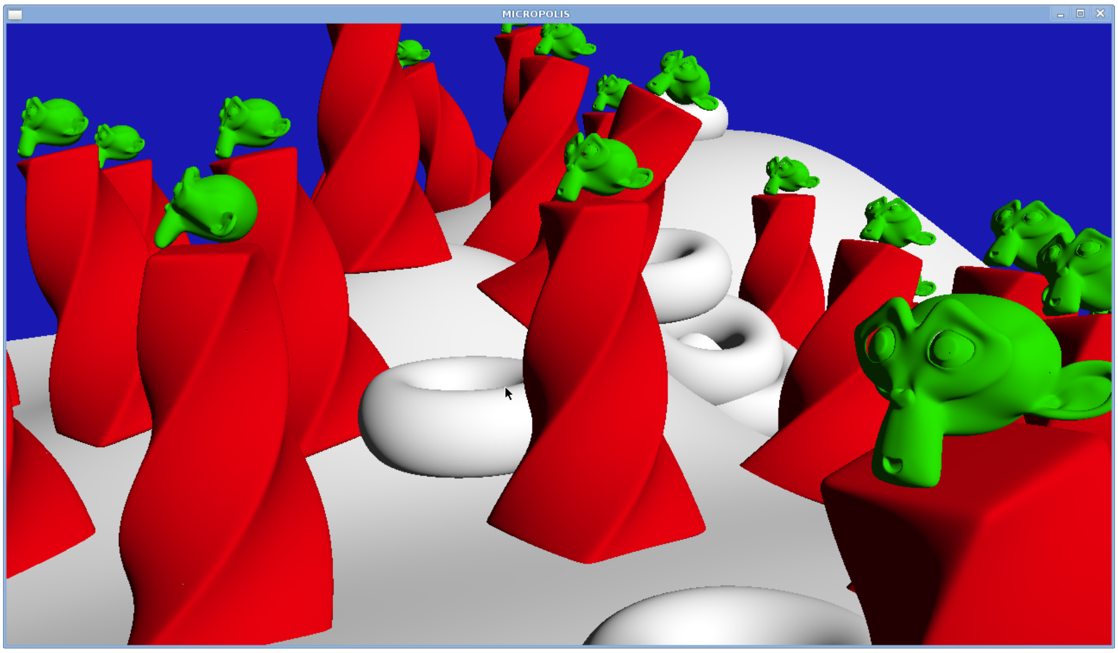


Figure 6.7: Screenshot of the micropolis application in action

CHAPTER 7

Performance Evaluation

7.1 Adaptive Subdivision

We have implemented both the breadth-first adaptive subdivision and our own subdivision approach with bounded memory guarantees for comparison purposes.

BREADTH implements the breadth-first approach of Patney and Owens [28]. In case this algorithm runs out of memory, it will allocate further memory on-the-fly. This is necessary since the worst-case memory consumption of breadth-first subdivision is so high that preallocation is not possible. This exact situation is what we want to avoid. Since we allow for a certain number of rendered frames before measurement, the necessary time-overhead for this does not affect the measured subdivision times.

BOUNDED implements adaptive subdivision with bounded memory as described in Chapter 4.

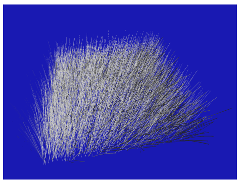
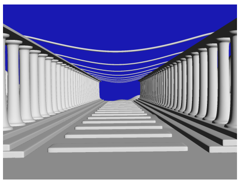
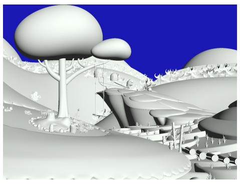
				
	TEAPOT	HAIR	COLUMNS	ZINKIA
N	32	10 000	12 850	999 812

Table 7.1: Overview of the different test scenes used for performance analysis. N is the number of surface patches in a scene before applying adaptive subdivision. Not pictured is the synthetic test scene EYESPLIT, because all that can be seen is a white rectangle over the entirety of the frame buffer. ZINKIA scene courtesy of Zinkia Entertainment, S.A.

scene	method	batch size	time [ms]	memory [MiB]	max patches	processed	processing rate [M patches/s]
TEAPOT	BREADTH	5030	1.72	0.52	5030	22172	12.92
TEAPOT	BOUNDED	10000	1.69	4.88	5030	22172	13.11
TEAPOT	BOUNDED	40000	1.70	19.53	5030	22172	13.03
TEAPOT	BOUNDED	200000	1.69	97.66	5030	22172	13.11
HAIR	BREADTH	150958	1.79	16.27	150958	430958	240.37
HAIR	BOUNDED	10000	6.98	5.08	49000	430958	61.71
HAIR	BOUNDED	40000	2.98	19.73	115488	430958	144.51
HAIR	BOUNDED	200000	1.80	97.86	150958	430958	239.91
COLUMNS	BREADTH	38712	3.02	4.14	38712	293178	96.98
COLUMNS	BOUNDED	10000	5.98	5.15	22326	293178	49.00
COLUMNS	BOUNDED	40000	3.02	19.80	38712	293178	97.03
COLUMNS	BOUNDED	200000	3.01	97.93	38712	293178	97.25
ZINKIA1	BREADTH	999812	6.21	107.78	999812	1402768	225.78
ZINKIA1	BOUNDED	10000	29.25	24.91	999812	1402768	47.95
ZINKIA1	BOUNDED	40000	12.50	39.56	999812	1402768	112.20
ZINKIA1	BOUNDED	200000	7.48	117.69	999812	1402768	187.50
ZINKIA2	BREADTH	3847162	18.92	414.74	3847162	9284930	490.81
ZINKIA2	BOUNDED	10000	138.89	24.91	999812	9284930	66.85
ZINKIA2	BOUNDED	40000	55.51	39.56	999812	9284930	167.26
ZINKIA2	BOUNDED	200000	25.62	117.69	1040464	9284930	362.44
ZINKIA3	BREADTH	9766796	33.90	1052.81	9766796	20946484	617.89
ZINKIA3	BOUNDED	10000	315.62	24.91	999812	20946484	66.37
ZINKIA3	BOUNDED	40000	120.62	39.56	999812	20946484	173.66
ZINKIA3	BOUNDED	200000	51.05	117.69	1305212	20946484	410.33
EYESPLIT	BREADTH	1950752	9.25	210.28	1950752	4024029	434.89
EYESPLIT	BOUNDED	10000	62.99	4.88	85236	4024029	63.89
EYESPLIT	BOUNDED	40000	23.68	19.53	260960	4024029	169.90
EYESPLIT	BOUNDED	200000	11.43	97.66	843844	4024029	352.01

Table 7.2: Test results for various combinations of test scenes and subdivision method. *max surfaces* is the maximum amount of surfaces stored in memory at any given point in time. *processed* is the total number of surfaces processed during subdivision including intermediate surfaces. The processing rate is the number of processed surfaces divided by the subdivision time.

Table 7.1 shows the test scenes we used for evaluating our renderer. TEAPOT contains a single large object composed of a small number of surfaces. HAIR is a single mesh with a large number of surfaces and moderate depth complexity. COLUMNS contains about the same number of surfaces as HAIR, but has a lower depth complexity. The ZINKIA scene is very detailed and contains almost a million surfaces.

We have prepared three different viewpoints to evaluate ZINKIA. These three views are extracted from a straight path that has the camera move along a line through the ZINKIA scene

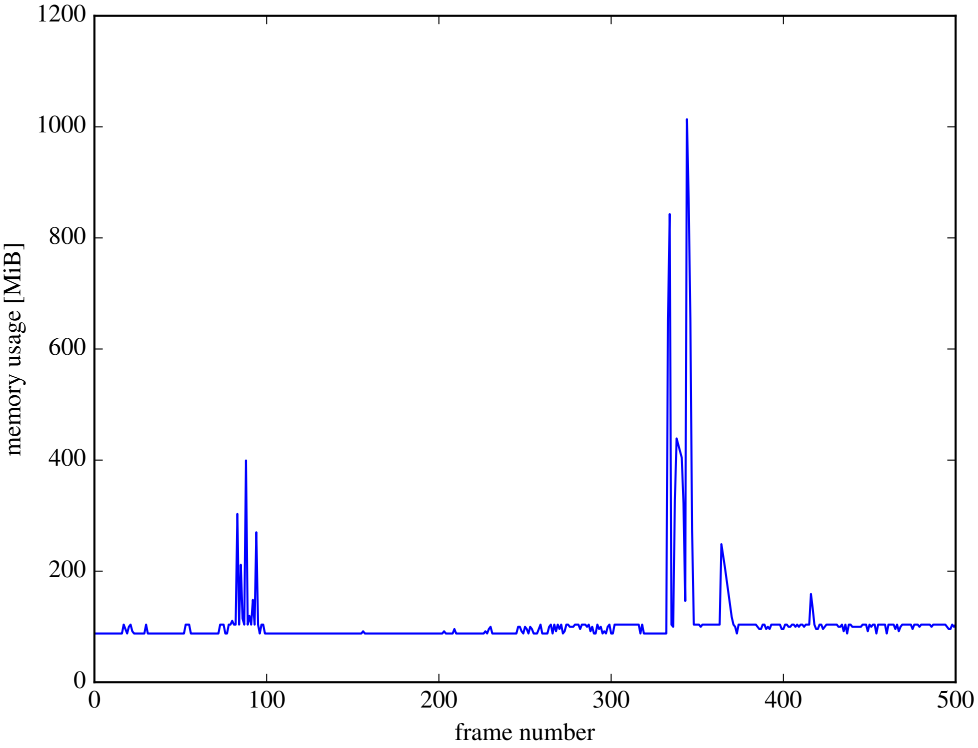


Figure 7.1: Memory usage of BREADTH as the camera moves along a straight path through the ZINKIA scene. Figure 1.2 gives the position and local context for the features in this graph.

as shown in Figure 1.2. Figure 7.1 shows the breadth-first memory usage at each position of this path. The views we chose are one representing the average case (ZINKIA1), one for the highest memory spike near the tree (ZINKIA2), and one for the 1 GiB spike close to the cliff (ZINKIA3). Larger screenshots of ZINKIA1, ZINKIA2, and ZINKIA3 can be found in Figures 7.12, 7.13, and 7.14 at then end of this chapter.

In addition, we have also prepared a synthetic test scene called EYESPLIT, which cannot be reasonably pictured. This is intended to demonstrate the possible worst-case behavior of our subdivision algorithms. EYESPLIT contains a single planar surface patch with the camera placed in such a way that the split axis of the surface falls onto the camera's eye plane. This has the effect that the subdivision of the surface does not terminate before the allowed number of recursive splits has been exhausted and the surface gets culled. The eye-split problem is an intrinsic property of the Reyes pipeline, and artists have learned to avoid it in production rendering [1]. Nevertheless, it is important that such a configuration can be evaluated without the subdivision pipeline stage of a renderer exceeding its memory budget.

All benchmarks have been measured on a system with an AMD Radeon R9 290 GPU and a 3.4GHz Intel Core i5-4670K CPU. The graphics driver used was Catalyst 14.9 on a 64-bit Linux system.

Table 7.2 lists the execution results for various combinations of adaptive subdivision methods and test models. The scenes are rendered at a resolution of 1280×720 and surfaces are split until they are smaller than 8 pixels along each dimension. For BOUNDED, three different batch sizes (low: 10000, medium: 40000, medium: 200000) are evaluated. The batch size of BREADTH is defined by the scene and view itself. The maximum number of recursive subdivisions k has been set to 23.

Note that the memory consumption of BREADTH is the actual amount of necessary memory, while BOUNDED is configured to allocate enough memory for the worst-case possible memory consumption. Especially for simple scenes, this can mean that the conservative amount of memory allocated by BOUNDED exceeds the amount of memory actually needed by both BREADTH and BOUNDED. The average case is usually a lot better. A good example for this is HAIR, which actually only requires at most 7 subdivisions to any surface in the scene. This can also be seen from the *max patches* value in Table 7.1, where the actual amount of stored patches for BOUNDED always remains lower than for BREADTH.

A variant of BOUNDED that reallocates memory buffers on-the-fly like BREADTH does could significantly reduce the amount of necessary memory for these scenes. Our own focus was more on handling extreme cases gracefully while accepting a constant memory budget for anything lower. This is why we have not implemented this.

For configurations where the view-inherent batch size of BREADTH does not exceed the configured batch size, we can achieve a similar performance with BOUNDED. This is expected, since the exact same amount of computation kernels with the same dimensions are executed. In case the assigned batch size of BOUNDED is lower than that of BREADTH, we get a smooth transition from low to high depending on the amount of assigned memory. Especially for scenes with high memory demand like ZINKIA3, assigning just 11% of the memory necessary for BREADTH can give 66% of the overall performance.

The 1 GiB spike of ZINKIA3 shows that doing naive breadth-first subdivision is not feasible for real-world graphics applications. The Zinkia scene is in no way extreme in what is to be expected of Reyes rendering for interactive applications, and the render settings we have chosen should be reasonable for the scene at hand. 1 GiB of memory is 25% of the total physical memory of a top-of-the line desktop GPU, and considering we are only rendering at 720p, this value would grow for higher resolutions. Figures like these seem especially prohibitive in the mobile space where such a memory consumption can easily exceed the total available memory on current devices.

Figure 7.2 demonstrates the impact of the chosen batch size on the performance of BOUNDED. The achievable processing rate depends highly on the intrinsic parallelism of a scene, with simpler scenes very quickly reaching a plateau. The performance of complex scenes like ZINKIA2/3 and EYESPLIT asymptotically approaches that of BREADTH when more memory is assigned. The curve of HAIR shows how the processing rate quickly rises with more assigned resources, starts to go flat, and then remains almost constant past a certain point. This is the point at which the batch size is large enough to keep all surfaces active at all times. It can be seen that the other curves mirror this behavior at different scales.

Note that the memory values used for the horizontal axis in Figure 7.2 don't include the

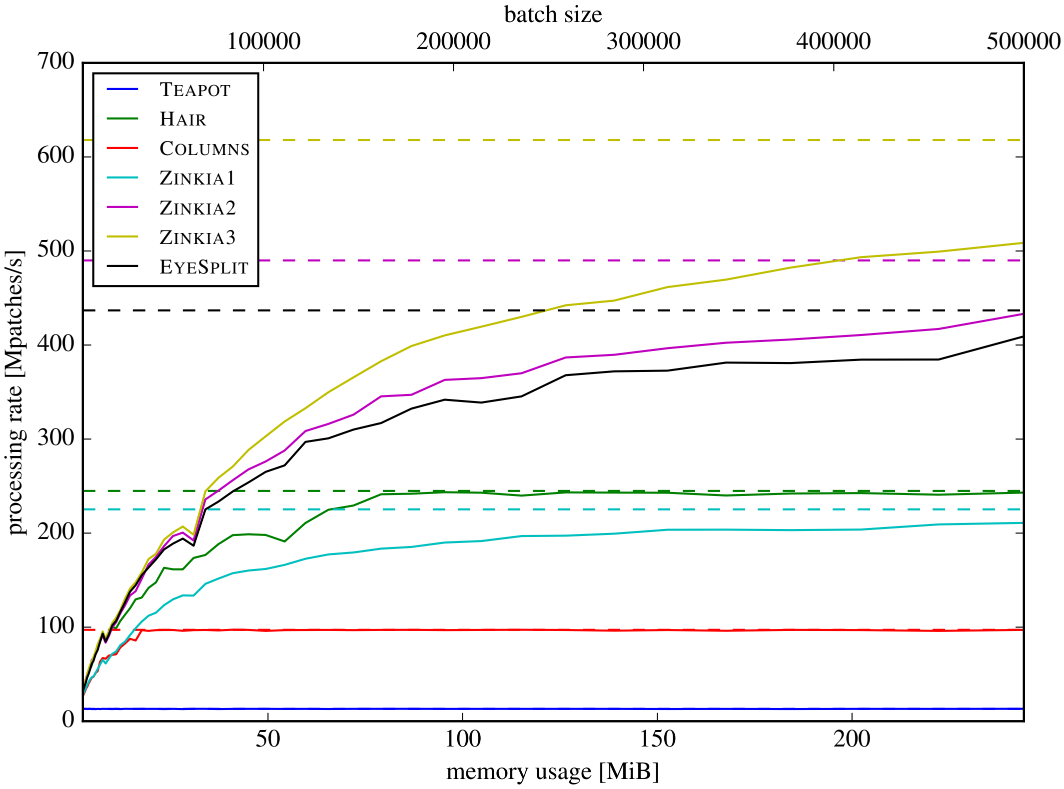


Figure 7.2: Subdivision performance for our test scenes depending on the amount of assigned memory and batch size. The X axis shows the amount of used memory on the bottom axis and the batch size on top. Smaller scenes very quickly level out, while larger scenes show asymptotic growth. The dashed horizontal lines represent the processing rate achievable by BREADTH and the upper bound for BOUNDED.

constant memory requirement for the initial number of patches. This is done to make the memory usage and batch-size axes align. If we didn't do this, the plot would be shifted on the x axis, with the ZINKIA plot being the only one with a clearly visible shift by about 20 MiB to the right. The constant offsets of the other scenes are relatively small with at most 0.27 MiB for COLUMNS.

Exact performance comparisons against previous implementations are difficult because of different rendering parameters, but our overall performance appears competitive modulo differences in hardware and rendering parameters:

- Patney and Owens [28] give times for the adaptive subdivision of TEAPOT (6.99 ms) and KILLEROO (3.46 ms). They perform fewer split operations (512×512 resolution with a 16-pixel bound) and use a significantly less powerful NVIDIA GeForce 8800 GTX for measurement. Under this configuration our subdivision times are 1.43 ms for TEAPOT and 0.30 ms for KILLEROO with BREADTH. The subdivision times for BOUNDED are essentially the same.

- Tzeng et al. [34] give overall frame render times including shading and rasterization for Teapot (51.81 ms), BigGuy (90.50 ms), and Killeroo (54.11). They render at resolution 800 × 800 and use a 16-pixel bound. Micropolis is considerably faster in this configuration (Teapot: 3.08 ms, BigGuy: 3.11 ms, Killeroo: 5.94 ms). However this is once again hard to compare since Tzeng et al.'s renderer uses complex transparency and 16× multisampling.

Work-group local Subdivision

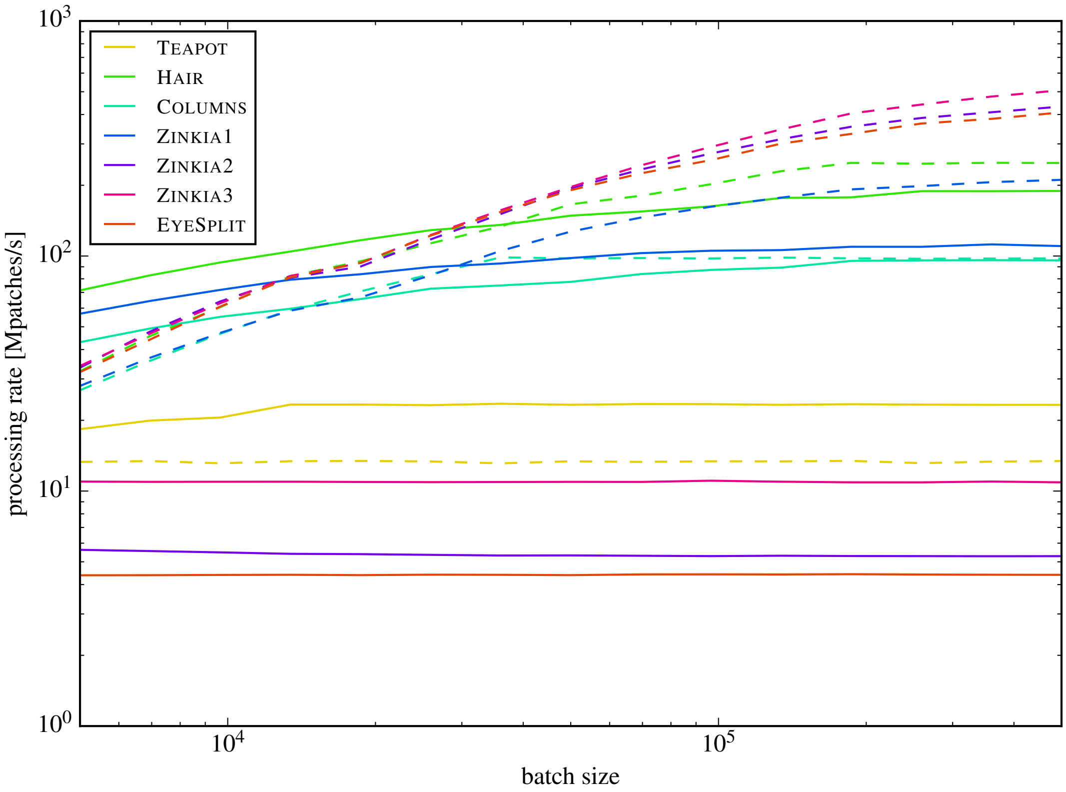


Figure 7.3: Performance comparison of the Bounded and Local depending on batch size. Solid lines represent Local and dashed lines represent Bounded. Local tends to outperform Bounded for small scenes and batch sizes, but for larger scenes Bounded quickly becomes faster.

Figure 7.3 shows the performance of the work-group local algorithm (Local) compared to that of Bounded. For small scenes like Teapot, Local performs considerably better, due to not having to return to the host for the first iterations. Local performs especially well for scenes with many small individual objects like Tree and Pillars. For larger scenes like Hair and

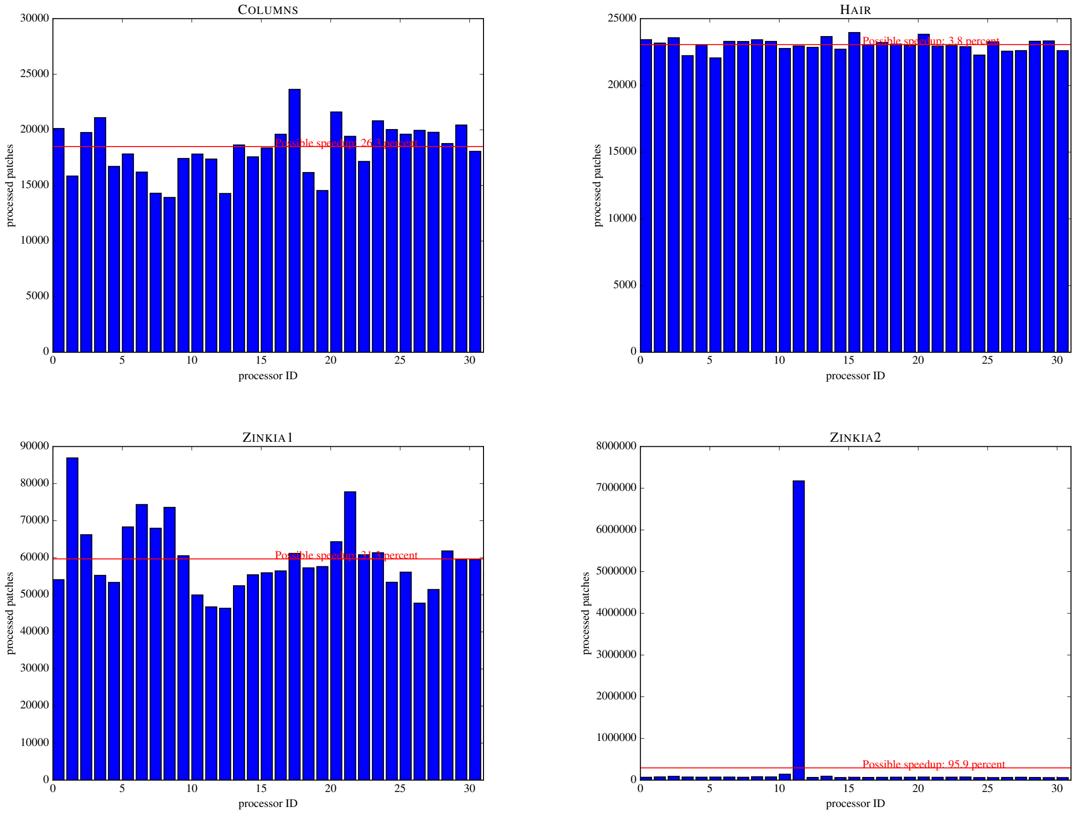


Figure 7.4: Illustration of the per work-group load of LOCAL for various test scenes. Ideally, all compute units should do the same amount of work. If one compute unit takes significantly longer, all other units have to wait. The red horizontal lines show the average load which is also the amount of work every processor would perform in an ideally balanced system.

COLUMNS, BOUNDED starts off worse, but tends to catch up to LOCAL, since less internal logic has to be performed and the work-load is implicitly balanced. BOUNDED severely outperforms LOCAL for all batch sizes for the ZINKIA scenes and especially for EYESPLIT. This is because our implementation of LOCAL evenly assigns the initial workload to the different work-groups without doing any load balancing during operation. In case one work-group takes significantly longer than the others, it will stall the entire kernel, which is the case for the mentioned scenes. Figure 7.4 illustrates this issue. Explicit load balancing between work-groups as described by Tzeng et. al. [34] would be necessary to alleviate this problem.

This should demonstrate that BOUNDED should be the method of choice. The only situation when LOCAL shows better time behavior is when host-device interactions dominate the total render time. This case can be seen for TEAPOT in Figure 7.5, where the GPU is idle for more than 25 percent of the frame. Kernel-side enqueue as supported in the new OpenCL 2.0 standard could be used to avoid this.

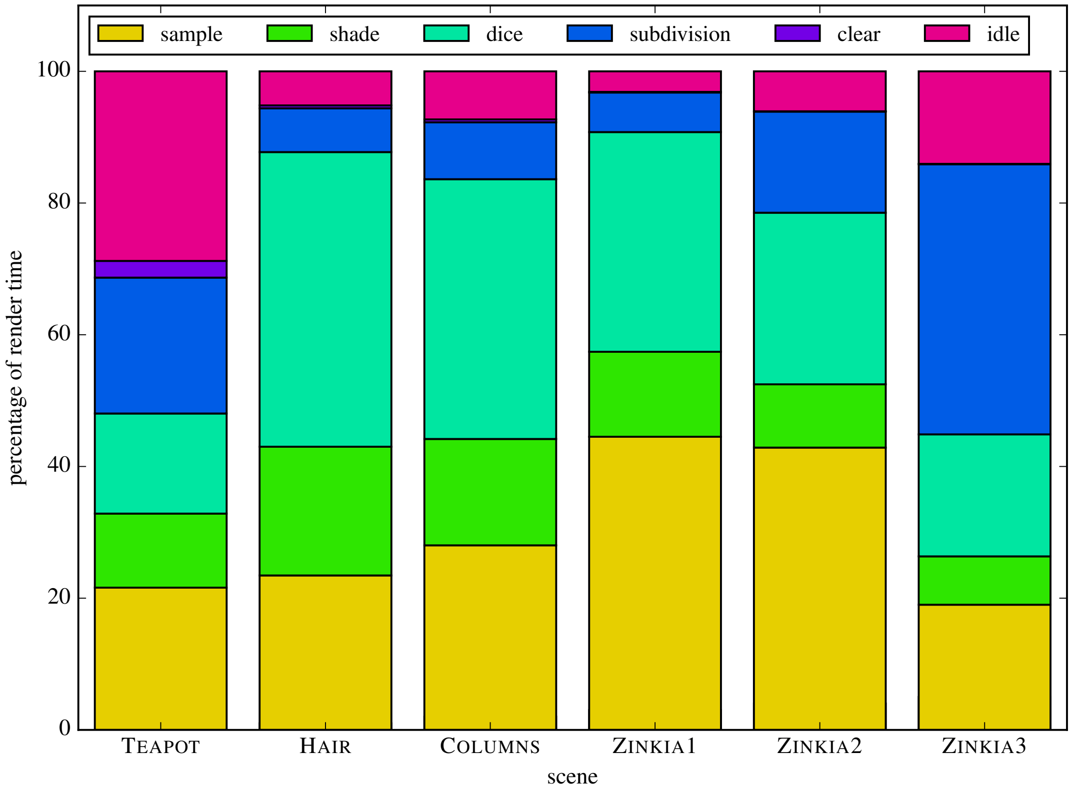


Figure 7.5: Relative amount of time spent for individual tasks when rendering various test scenes. Larger scenes are usually dominated by the surface evaluation and rasterization, while less complex scenes can spend a significant amount of time in subdivision. This also shows that there is a significant amount of time the GPU is idle since it's waiting for intervention from the CPU. ZINKIA3 is special insofar as it contains an eye-split so that lots of subdivided surfaces are generated that end up culled due to exceeding the maximum number of splits. The subdivision algorithm used was BOUNDED.

7.2 Rendering

The benchmarks in the previous section only measure the time spent subdividing and ignore overall rendering performance including dicing, shading, and rasterization. In this section we will give an overview on how much time is spent in the individual rendering stages. Figures 7.8 and 7.9 list kernel traces for the overall rendering process of a single frame for various scenes. The traces show timing information for both BOUNDED and LOCAL. BOUNDED performs adaptive subdivision in a succession of kernel calls that do bounding, apply prefix sums and then split, while LOCAL is implemented in a single large kernel.

What can be seen is that for reasonably sized scenes, only a small part of the time is spent in subdivision. This is also illustrated in Figures 7.5 and 7.6, which show the relative amount

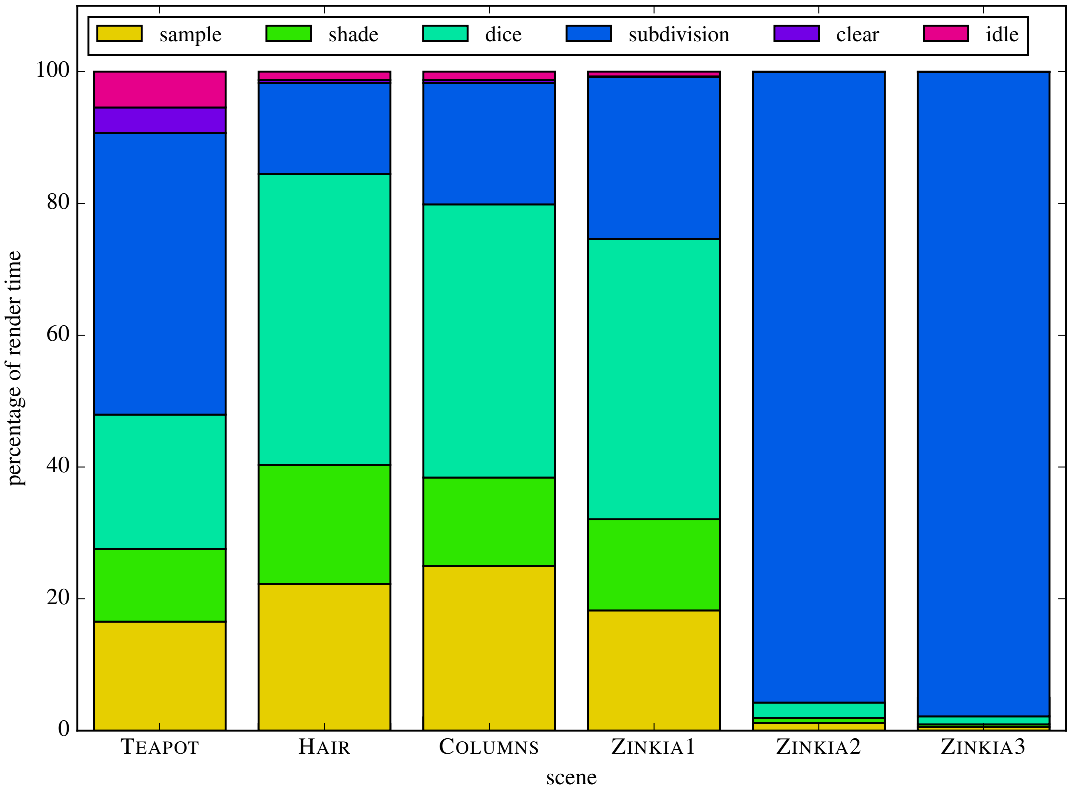


Figure 7.6: Relative amount of time spent for individual tasks when rendering various test scenes. The subdivision algorithm used was LOCAL.

of time spent in individual stages. The stages spent most time in are dicing and polygon sampling. Subdivision can take up a significant proportion of the frame render time for small and subdivision-heavy scenes like ZINKIA3, however. For BOUNDED, there is also a significant chunk of the overall time where the GPU has to idle since it has to interact with the CPU. LOCAL, by comparison, requires fewer iterations and can thereby reduce the number of necessary host interventions when processing. However, we can see that the missing load balancing of LOCAL makes it perform very poorly for scenes like ZINKIA1 and ZINKIA2.

One thing that can be noticed is that the later stages have a significantly more uniform performance for LOCAL than for BOUNDED. One reason for this is probably that the amount of processed surfaces remains constant with LOCAL. This would indicate that it might be desirable to have BOUNDED collect the output of several iterations and only send them to the dicing stage once certain number of surfaces have been generated. This is something not implemented at the moment.

Another difference between the traces for BOUNDED and LOCAL is the relative amount of time spent dicing and sampling. BOUNDED spends significantly more time in the sample kernel. The reason for this is due to locality and a performance limitation of our sampling kernel. Tile

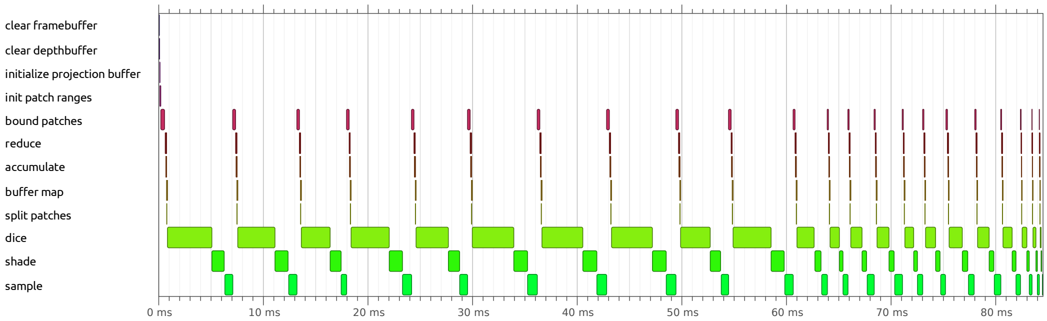
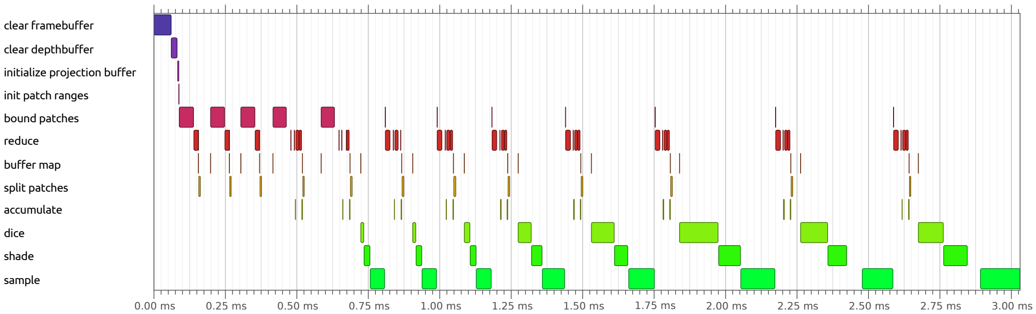


Figure 7.7: Kernel trace for the ZINKIA scene with BOUNDED when disabling tile-locking in the sample kernel

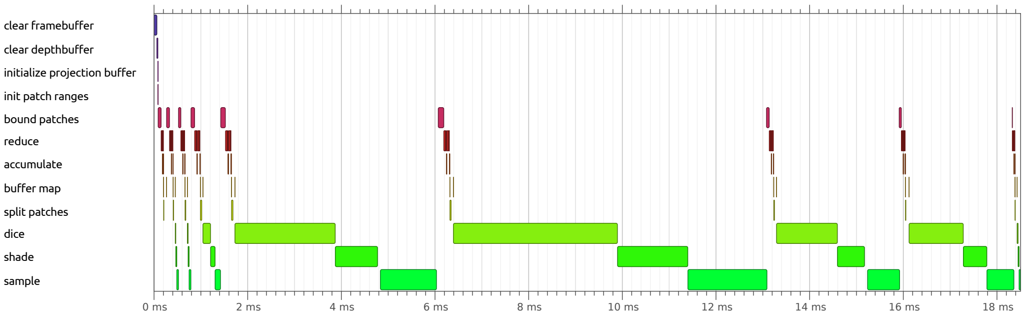
locking is used for applying the fragments generated in the kernel to the framebuffer. In case another work-group has acquired a the lock, the kernel has to wait. The output surfaces of LOCAL are less close to each other, since the overall scene is split up between individual work-groups that concurrently write to a single output buffer. This is visualized in Figure 7.11 and Figure **??**, where the surface color indicates its output time. High locality of output surfaces is a desirable property that should improve performance for concurrent read accesses as used when dicing or for texture mapping. But it also demonstrates a serious limitation of our sort-last approach and current implementation of the sample kernel.

Figure 7.7 shows the kernel trace for the ZINKIA scene and BOUNDED when commenting out locking of tiles in the sample kernel. While this will potentially result in render artifacts, the time spent in the sample kernel is significantly reduced. This indicates that further work will be necessary on the rasterization part of the renderer. However, we chose to put the focus of this thesis on efficient and robust surface subdivision and weren't able to further explore this due to time constraints.

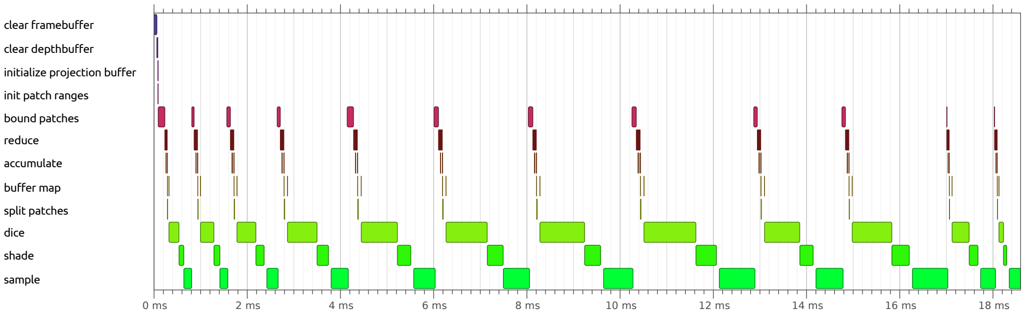
One thing to note is that we still use pixel-level locking within the innermost loop in Figure 7.7, but this does not pose much of a performance problem, since polygons within a grid tend to have only little overlap. This might be different for procedurally displaced surfaces, however.



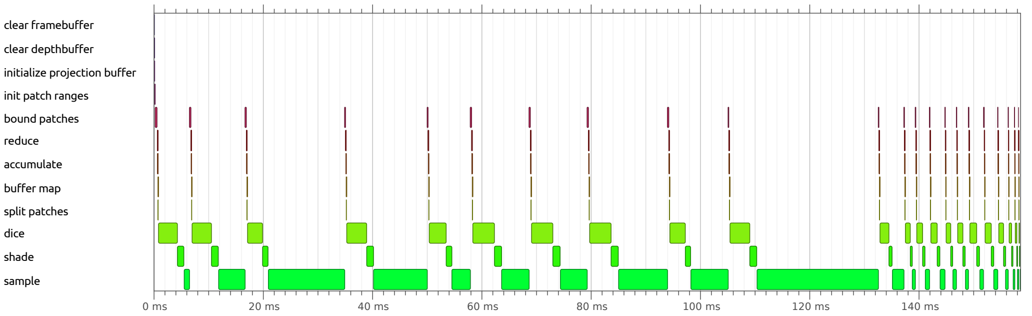
(a) Teapot



(b) Hair

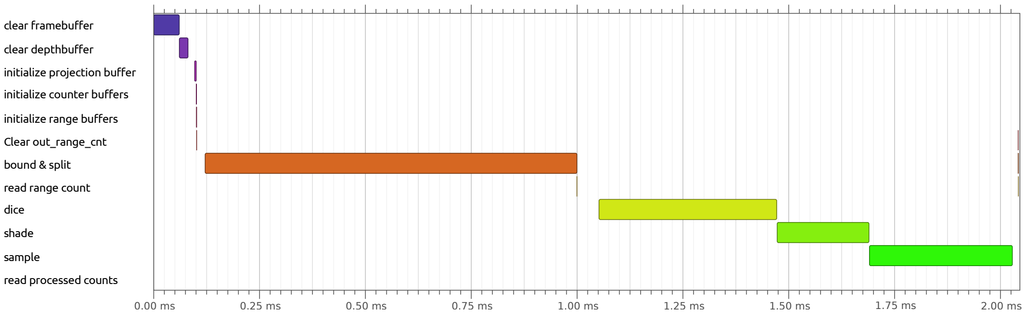


(c) Columns

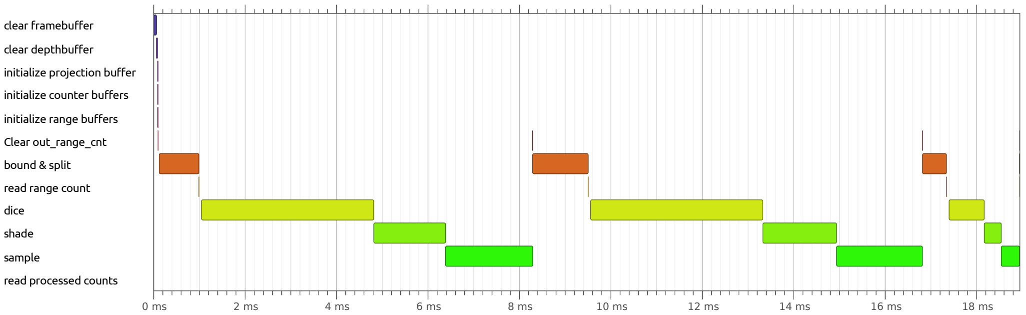


(d) Zinkia

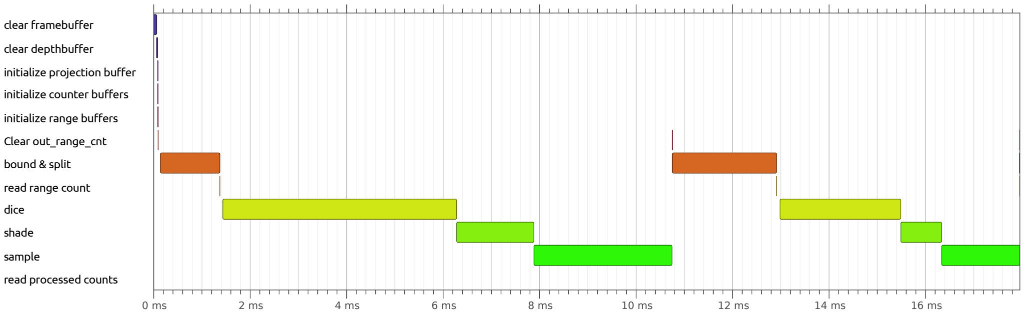
Figure 7.8: OpenCL timing traces for various test scenes using Bounded for subdivision.



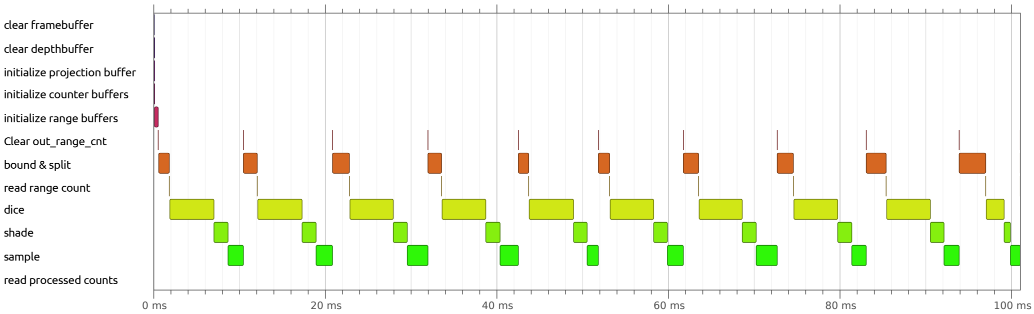
(a) Teapot



(b) Hair



(c) Columns



(d) Zinkia

Figure 7.9: OpenCL timing traces for various test scenes using Local for subdivision.

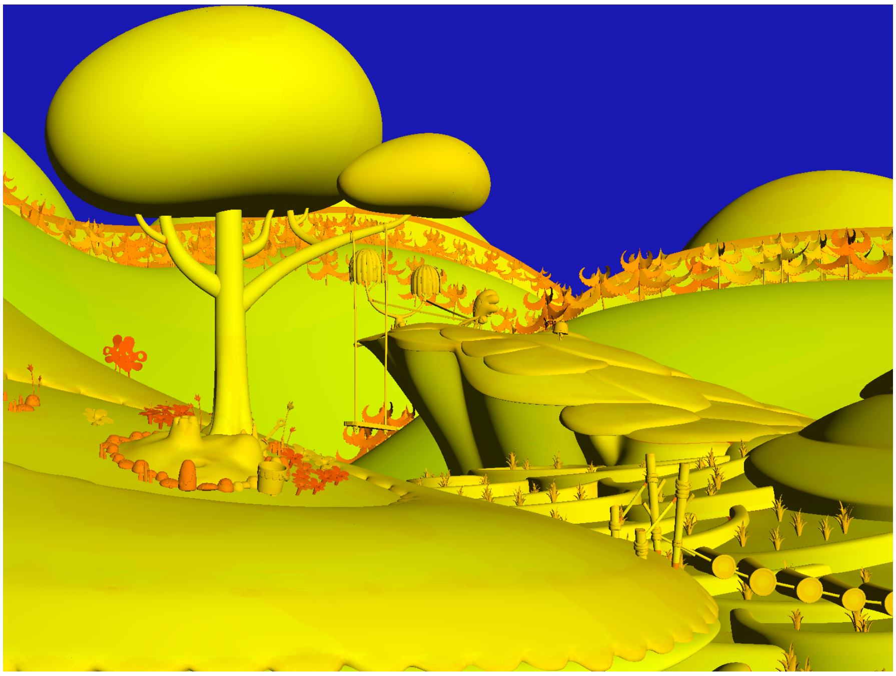


Figure 7.10: The ZINKIA scene subdivided using BOUNDED with the output order of surfaces indicated by color. Geometric locality is preserved. Zinkia Entertainment, S.A.

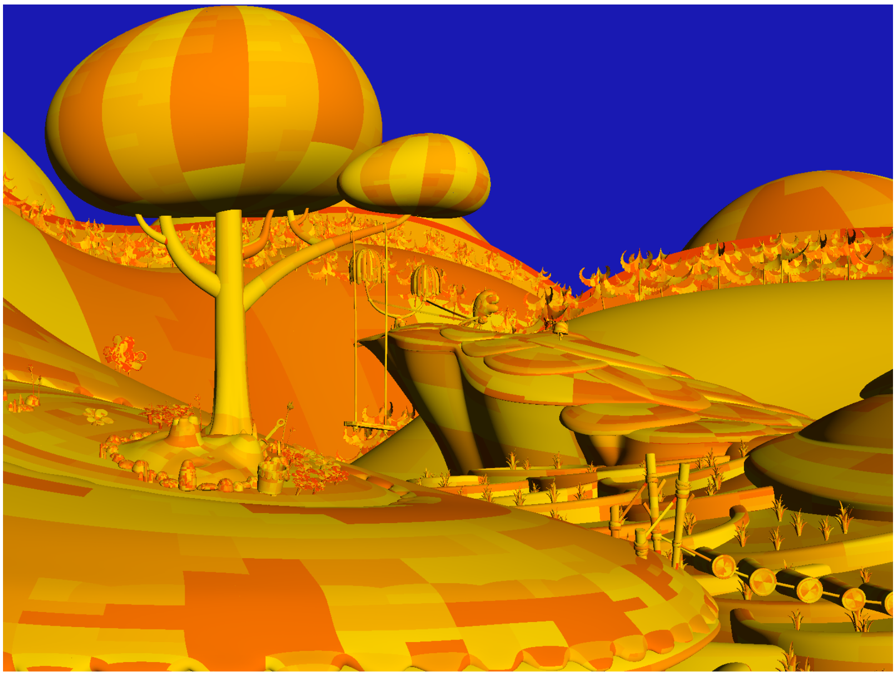


Figure 7.11: The ZINKIA scene subdivided using LOCAL with the output order of surfaces indicated by color. Geometric locality is not preserved. Zinkia Entertainment, S.A.

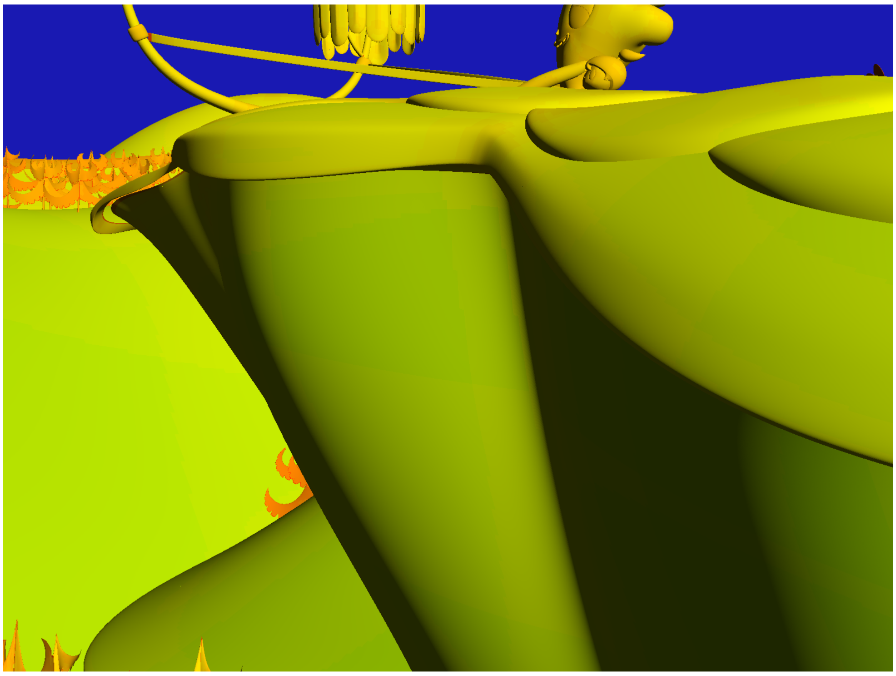


Figure 7.12: The ZINKIA1 scene. Surface colors indicate the output order. Zinkia Entertainment, S.A.

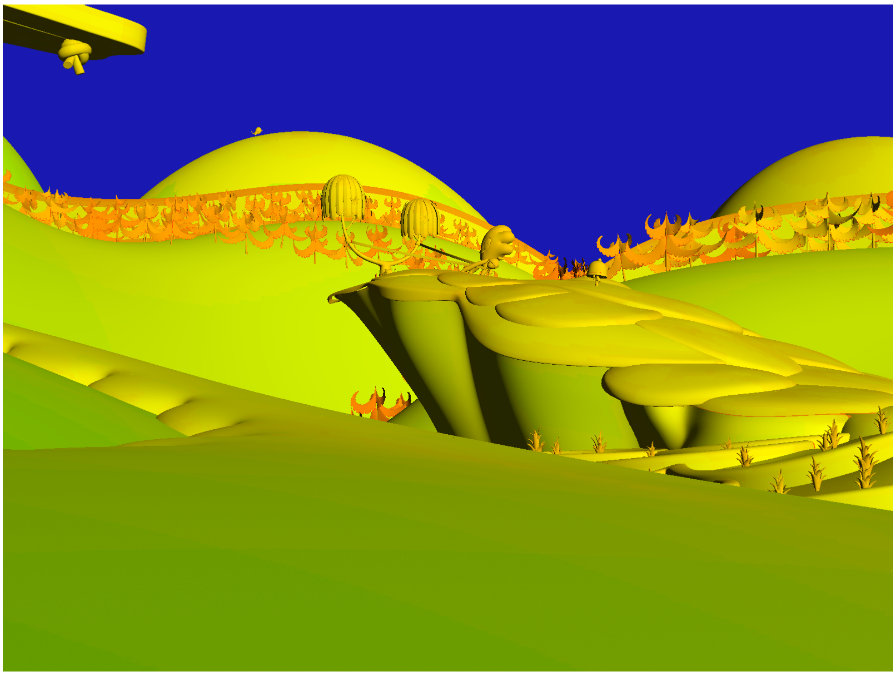


Figure 7.13: The ZINKIA2 scene. Surface colors indicate the output order. Zinkia Entertainment, S.A.

Figure 7.14: The ZINKIA3 scene. Surface colors indicate the output order. Zinkia Entertainment, S.A.

CHAPTER 8

Conclusion

This thesis has presented a hardware-accelerated implementation of the Reyes algorithm in OpenCL. Our major contribution is a method for implementing adaptive surface subdivision on the GPU with a bounded peak memory consumption. The output order of generated surfaces also preserves locality. We believe the memory advantages of our algorithm over previous GPU implementations of bound-and-split may make adaptive surface subdivision more tractable for real-time usage, in particular for constrained rendering environments like mobile platforms.

The performance of BOUNDED could be greatly improved by using device-side enqueue, as supported in version OpenCL 2.0. This is because a lot of the overhead of performing more iterations comes from the necessary host-device interactions. If this overhead were negligible, then even relatively small batch sizes should be able to fully utilize all available parallelism for a given graphics processor. Figure 7.5 shows the amount of time spent idling during rendering of a frame, most of which should be avoidable with kernel-side enqueue. However, at the time of writing AMD only released a preliminary driver supporting OpenCL 2.0, which is why we weren't able to fully explore this.

As we have seen in the previous chapter, bound-and-split only accounts for a small portion of the overall render time. It may therefore be feasible to set aside a small portion of a GPU's compute units just for this task, so that subdivision and rendering can be performed concurrently in a true pipeline.

Robust adaptive subdivision has many possible uses. Hanika et al. [12] present a method for ray-tracing polygons using a two-level approach with ray reordering. This method may be well-suited for implementation on the GPU using our described method for geometry generation. Integrating adaptive subdivision into a larger GPU graphics pipeline would also allow for interesting optimization possibilities like culling occluded surfaces during subdivision.

We have chosen a sort-last approach in conjuction with tile-level locking for our implementation of micropolyon rasterization. While this approach works reasonably well, we have found that lock congestion is a serious source of overhead in some of our configurations. Further work would be necessary to improve this inefficiency. Lock-free approaches for parallel fragment composition like the one described by Patney et. al. [29] could be explored for this.

Many alternative GPU-based software rasterizers instead opt for a sort-middle approach, where primitives are assigned to screen tiles and then a rasterization kernel is called for each tile. However, this requires further record keeping and may suffer from load imbalances when certain screen areas have a significantly higher depth complexity than others. Hardware rasterizers use special silicon to perform fragment merging and do not suffer from these inefficiencies. If this functionality could be exposed to compute kernels or if rasterization hardware simply added support for handling micropolygons efficiently then this would be the ideal solution to this problem.

The source code for *Micropolis*, the OpenCL Reyes renderer described in this thesis, can be found at `https://github.com/ginkgo/micropolis`.

Bibliography

[1] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan: Creating CGI for Motion Picture*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999.

[2] J. S. Brunhaver, K. Fatahalian, and P. Hanrahan. Hardware implementation of micropolygon rasterization with motion and defocus blur. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 1–9, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[3] Christopher A. Burns, Kayvon Fatahalian, and William R. Mark. A lazy object-space shading architecture with decoupled sampling. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 19–28, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[4] Edwin Catmull and James H. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10(6):350–355, November 1978.

[5] Christophe Riccio. Opengl mathematics, `http://glm.g-truc.net`. Accessed: 2014-11-03.

[6] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes image rendering architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, pages 95–102, July 1987.

[7] Tony DeRose, Michael Kass, and Tien Truong. Subdivision surfaces in character animation. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 85–94, New York, NY, USA, 1998. ACM.

[8] Christian Eisenacher and Charles Loop. Data-parallel micropolygon rasterization. In Stefan Seipel and Hendrik Lensch, editors, *Eurographics 2010 Annex: Short Papers*, May 2010.

[9] Christian Eisenacher, Quirin Meyer, and Charles Loop. Real-time view-dependent rendering of parametric surfaces. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, pages 137–143, 2009.

[10] Kayvon Fatahalian, Edward Luong, Solomon Boulos, Kurt Akeley, William R. Mark, and Pat Hanrahan. Data-parallel rasterization of micropolygons with defocus and motion blur. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 59–68, New York, NY, USA, 2009. ACM.

[11] Matthew Fisher, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, William R. Mark, and Pat Hanrahan. DiagSplit: Parallel, crack-free, adaptive tessellation for micropolygon rendering. *ACM Transactions on Graphics*, 28(5):150:1–150:10, December 2009.

[12] Johannes Hanika, Alexander Keller, and Hendrik P. A. Lensch. Two-level ray tracing with reordering for highly complex scenes. In *Proceedings of Graphics Interface 2010*, GI '10, pages 145–152, 2010.

[13] Qiming Hou, Xin Sun, Kun Zhou, Christian Lauterbach, and Dinesh Manocha. Memory-scalable GPU spatial hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics*, 17(4):466–474, April 2011.

[14] Qiming Hou, Kun Zhou, and Baining Guo. BSGP: Bulk-synchronous GPU programming. *ACM Trans. Graph.*, 27(3):19:1–19:12, August 2008.

[15] Kenton Varda. Cap'n proto, `https://kentonv.github.io/capnproto/`. Accessed: 2014-11-03.

[16] Khronos OpenCL Working Group. *OpenCL 1.2 API and C Language Specification*, November 2012.

[17] Denis Kovacs, Jason Mitchell, Shanon Drone, and Denis Zorin. Real-time creased approximate subdivision surfaces. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, pages 155–160, New York, NY, USA, 2009. ACM.

[18] Samuli Laine and Tero Karras. High-performance software rasterization on gpus. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 79–88, New York, NY, USA, 2011. ACM.

[19] Charles Loop and Christian Eisenacher. Real-time patch-based sort-middle rendering on massively parallel hardware. Technical Report MSR-TR-2009-83, Microsoft Research, May 2009.

[20] Charles Loop and Scott Schaefer. Approximating Catmull-Clark subdivision surfaces with bicubic patches. *ACM Transactions on Graphics*, 27(1):8:1–8:11, March 2008.

[21] Charles Loop, Scott Schaefer, Tianyun Ni, and Ignacio Castaño. Approximating subdivision surfaces with Gregory patches for hardware tessellation. *ACM Transactions on Graphics*, 28(5):151:1–151:9, December 2009.

[22] Jacob Munkberg, Jon Hasselgren, Robert Toth, and Tomas Akenine-Möller. Efficient bounding of displaced Bézier patches. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 153–162, 2010.

[23] Matthias Nießner and Charles Loop. Analytic displacement mapping using hardware tessellation. *ACM Transactions on Graphics*, 32(3):26:1–26:9, July 2013.

[24] Matthias Nießner, Charles Loop, Mark Meyer, and Tony Derose. Feature-adaptive GPU rendering of Catmull-Clark subdivision surfaces. *ACM Transactions on Graphics*, 31(1):6:1–6:11, February 2012.

[25] Matthias Nießner, Charles T. Loop, and Günther Greiner. Efficient evaluation of semi-smooth creases in Catmull-Clark subdivision surfaces. In *Eurographics (Short Papers)*, pages 41–44, 2012.

[26] John D. Owens, Brucek Khailany, Brian Towles, and William J. Dally. Comparing Reyes and opengl on a stream architecture. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '02, pages 47–56, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[27] Anjul Patney, Mohamed S. Ebeida, and John D. Owens. Parallel view-dependent tessellation of Catmull-Clark subdivision surfaces. In *Proceedings of the Conference on High Performance Graphics*, HPG '09, pages 99–108, 2009.

[28] Anjul Patney and John D. Owens. Real-time Reyes-style adaptive surface subdivision. *ACM Transactions on Graphics*, 27(5):143:1–143:8, December 2008.

[29] Anjul Patney, Stanley Tzeng, and John D. Owens. Fragment-parallel composite and filter. *Computer Graphics Forum (Proceedings of EGSR 2010)*, 29(4):1251–1258, June 2010.

[30] Daniel Sanchez, David Lo, Richard M. Yoo, Jeremy Sugerman, and Christos Kozyrakis. Dynamic fine-grain scheduling of pipeline parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 22–32, October 2011.

[31] Michael Schwarz and Marc Stamminger. Fast GPU-based adaptive tessellation with cuda. *Computer Graphics Forum*, 28(2):365–374, 2009.

[32] Jos Stam. Exact evaluation of catmull-clark subdivision surfaces at arbitrary parameter values. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 395–404, New York, NY, USA, 1998. ACM.

[33] Ian Stephenson. Implementing RenderMan on the Sony PS2. In *ACM SIGGRAPH 2003 Sketches &Amp; Applications*, SIGGRAPH '03, pages 1–1, New York, NY, USA, 2003. ACM.

[34] Stanley Tzeng, Anjul Patney, and John D. Owens. Task management for irregular-parallel workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 29–37, 2010.

[35] Kun Zhou, Qiming Hou, Zhong Ren, Minmin Gong, Xin Sun, and Baining Guo. RenderAnts: Interactive Reyes rendering on GPUs. *ACM Transactions on Graphics*, 28(5):155:1–155:11, December 2009.