Mark Dokter

# Applications of Massively Parallel Geometry Processing

**DOCTORAL THESIS**

to achieve the university degree of
Doktor der technischen Wissenschaften

submitted to

**Graz University of Technology**

Advisors
Prof. Dr. Dieter Schmalstieg
Ass. Prof. Dr. Markus Steinberger

Referee
Prof. Dr. Karol Myszkowski
Max Planck Institute for Informatics

Graz, Austria, March 2019

To my family

Ubi materia, ibi geometria

*Johannes Kepler (1571—1630)*

## Affidavit

*I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used.*

*The text document uploaded to TUGRAZonline is identical to the present doctoral thesis.*

| | | |
|---|---|---|
| Place | Date | Signature |

## Eidesstattliche Erklärung

*Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.*

*Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Dissertation identisch.*

| | | |
|---|---|---|
| Ort | Datum | Unterschrift |

# Abstract

Geometry is one of computer graphic's core concepts describing how to represent, store and process artificially generated imagery. In this thesis, we investigate how to make best use of a modern graphics processing unit to tackle various tasks involved in handling geometry in a graphics pipeline. We use sophisticated scheduling mechanisms to generate geometry on the fly on a large scale by evaluating shape grammars. We analyze the derivation process at a high level by introducing the operator graph, which not only permits to adjust execution patterns to improve efficiency, but also lets us mimic various styles of shape grammar derivation systems proposed in previous work to analyze and compare their performance within the same software framework. Furthermore, we developed a novel graphics primitive—the CPatch—for rasterization of two dimensional vector graphics. Like triangles in polygon meshes for 3D computer graphics, CPatches not only consisting of lines but also curves, can be thought of as equivalent for 2D vector graphics. A third aspect of geometry processing that this thesis contributes to deals with sampling for visibility. An object space shading approach for decoupled rendering systems needs to determine what will be visible in a scene from a certain view point to reduce the shading workload to this subset of the scene's geometry. This subset—the visible set—can be insufficient and cause disocclusion artifacts if either the view point at the time a frame is displayed differs too much from the view point at the time of shading, or the input primitives were simply not detected because their transformed geometry is too small and fell through the sampling grid. In either case, the artifacts, stemming from missing information, can be mitigated by enriching the visible set by additional primitives, forming a potentially visible set (PVS). Our proposed method is a solution to the latter problem of geometry that can not be sampled correctly

because of its size.

**Keywords.** computer graphics, virtual reality, GPU computing, scheduling, geometry processing, vector graphics, svg, decoupled rendering, rasterization, visibility sampling, CUDA, Vulkan

# Kurzfassung

Geometrie, ein zentrales Konzept in der Computergrafik, wird dazu verwendet, zu beschreiben, wie künstlich erzeugte Bilder repräsentiert, gespeichert und verarbeitet werden. Diese Arbeit untersucht die Anwendbarkeit moderner Grafikprozessoren (GPU) um verschiedene Aspekte der Geometrieverarbeitung in Grafikpipelines zu verbessern. Ausgeklügelte Ablaufkoordinationsmechanismen erlauben es, Figur-Grammatiken zur Laufzeit auszuwerten um große Mengen an Geometrie direkt zur Darstellung zu verwenden. Zur Analyse des Ableitungsprozesses führen wir den Operator Graph ein. Dieser erlaubt es nicht nur, Ausführungsmuster anzupassen um die Effizienz des Vorganges zu steigern, sondern auch verschiedene Figur-Grammatik Stile zu imitieren um deren Leistungsfähigkeit im gleichen Software System zu vergleichen. Des Weiteren haben wir den CPatch entwickelt—eine neuartige geometrische Grundform zur Rasterisierung zwei dimensionaler Vektorgrafik. Wie Dreiecke in Polygon Netzen zur Darstellung in der 3D Grafik, werden CPatches dazu verwendet zweidimensionale Vektorgrafiken anzuzeigen. Sie bestehen jedoch nicht nur aus Linien zur Begrenzung, sondern auch aus Kurven. Ein dritter Aspekt der Geometrieverarbeitung, den diese Dissertation behandelt, ist die Detektion der Sichtbarkeit von Dreiecken. Bei Anwendung von Object Space Shading Methoden in entkoppelten Render Systemen, müssen die sichtbaren Elemente eines bestimmten Blickwinkels einer Szene zuerst ermittelt werden um den Rechenaufwand in Grenzen zu halten und vorhandene Rechenleistung auf auf diesen sichtbaren Teil der Szene zu konzentrieren. Diese Menge der sichtbaren Elemente kann unter Umständen nicht ausreichen, um eine Szene fehlerfrei darzustellen. Wenn zum Beispiel der Blickwinkel zwischen Erzeugung und Anzeige der Bildinformation verschieden ist, oder Dreiecke nicht gerendert werden, weil sie zu klein sind um als sichtbar markiert zu werden,

entstehen Artefakte auf Grund der fehlenden Bildinformation. Die Anzahl dieser Artefakte kann vermindert werden indem man die Menge der sichtbaren Elemente um wahrscheinlich sichtbare erweitert. Die vorgestellte Methode kann zu kleine Dreiecke erkennen um eine potentiell sichtbare Menge an Dreiecken zu erzeugen.

# Acknowledgments

I would like to thank the people who supported me in one way or another during my time as a PhD student. First of all, Prof. Dieter Schmalstieg, who provided a great working environment with a lot of freedom to pursue research ideas and at the same time pushing me to not stray too far from the path to completion of the doctorate. He was always encouraging to pursue new ideas and took the time for fruitful discussions to put them into practice. Special thanks to to my co-supervisor, Ass. Prof. Markus Steinberger, who guided me through many challenges and provided advice and support swiftly with seemingly never ending patience. He not only provided help and solutions when hitting obstacles but also motivated me continuously with his inspiring enthusiasm and exemplary commitment. Furthermore, I am grateful for having Prof. Karol Myzkowski as the external referee to review my thesis.

I am proud to have worked with many brilliant people in the various projects and research groups throughout the years. I started out on medical visualization with Jan and Philip. We had a great time and met interesting people at several project meetings all over Europe. Before joining Markus for an extended research stay in Saarbrücken, my struggle in pushing the limits of GPU computing in computer graphics was shared by Pedro Boechat, who spent long hours at the office with me to get our first publication accepted at a major conference. My stay at the Max Planck Institute for Informatics (MPII) in Saarbrücken was made a pleasant one by a lot of people. A probably incomplete list includes Jozef, besides always being motivated for discussion and asking good questions, he made sure that we all got a fair share of workout and sore muscles. Renjie provided me with free WiFi and company on the way to work. Rhaleb being a good host when paying Saarbrücken a

short visit after my stay. Bertram who was never short of an anecdote to entertain us at lunch time. Bernhard, Timo, Yulia, Petr and Thomas made sure that there was a life besides work in Saarbrücken. Sabine and Ellen, who helped me through all the paper work and organizational matters around the MPII. Hans-Peter, who provided the opportunity to work at the MPII and a liberal working environment to pursue my research. Returning to Graz University of Technology, I was part in the research group led by Markus Steinberger, internally called *the GPU people*. My thanks for many interesting discussions and a great working experience go to Elena, Bernhard, Daniel, Karl, Martin and Matthias. One subset of the GPU people was the *vector streaming group* consisting of Jörg, Philip, Thomas, Bernhard and myself. Thank you for the pleasant and productive collaboration. Special thanks go to Mikey, who shared an office with me. Having a living encyclopedia of programming language and graphics API standards sitting in the room really helps a lot with life in computer science. Next I want to thank all of the administrative staff at the Institute for Computer Graphics and Vision (ICG). Especially Christina, Nicole, Andi, Daniel and Alex provided valuable support throughout the years to get services running, forms signed and organizational questions answered quickly. Furthermore, my gratitude for advice, inspiration, guidance or good colleagueship go to Clemens Arth, Lorenz Jäger, Prof. Vincent Lepetit and Prof. Franz Leberl.

Last but not least I want to thank my parents, Walter and Anita, my sister Nadja and my wife Julia, who supported and encouraged me in every possible way.

My work was also funded by the MPII, the Christian Doppler Laboratory for Semantic 3D Computer Vision and Qualcomm Inc.

# Contents

# 1 Introduction

## 1.1 Motivation

Approaching the end of the second decade of the twenty first century, graphics processing units have become a common piece of hardware from the smart phone in the pocket over laptop and desktop computers to compute clusters in data centers. The widespread adoption of these many-core devices gave rise to the development of parallel algorithms. With such powerful tools, we set out to improve methods which deal with one of the central topics in computer graphics that is geometry.

Geometry as we understand it in the context of this work is a collection of coordinates that describe positions, usually in two or three dimensional space. A tuple of three coordinates, for example, would describe a point in space that is at some distance to the origin and is called a vertex. Two vertices could describe a line, three a triangle—the most commonly used and most efficiently handled geometric primitive in computer graphics. But three vertices could also describe quadratic Bézier curve, for example—start, end and control point. So we can define the shape and position of things we want to display by their geometric properties and store them in some suitable format. From the stored description to the final image, several steps of processing are needed. Such processing systems are usually organized in a pipeline design involving transformation, projection, rasterization and shading stages.

**Graphics pipelines** implemented in current graphics application programming interface (API) consist of several programmable and fixed-function stages. Figure 1.2 shows an overview of the OpenGL® pipeline design. The stages in blue are user

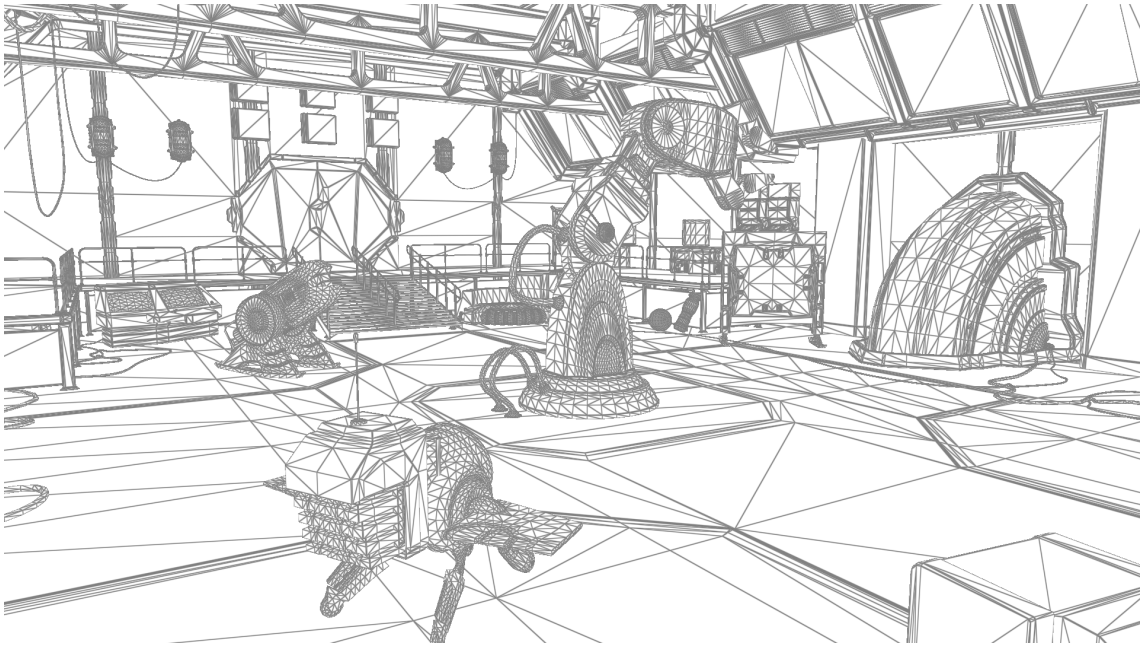


Figure 1.1: Visualization of geometric primitives, e.g., a wire-frame rendering with flat black and white coloring of the triangles and their borders.

programmable. Shader code is executed directly on the graphics processor. The first three of the programmable stages deal with geometry processing and the remaining one is for per pixel operations. The API is used to interface with the graphics hardware, where threads are assigned with a granularity appropriate for the various pipeline stages. The stages displayed in orange are assigned a fixed function. User control in these stages, if at all possible, works via API calls to configure the pipeline before drawing commences. A brief overview of these stages follows. The details of what these stages do in particular can be looked up in the full specification [43].

- **Vertex Assembly** Vertex data (coordinates) and additional attributes (color, normal, texture coordinates, etc) are pulled from input buffers.
- **Vertex Shader** User code is executed that usually performs transformation operations on the vertex coordinates, like applying model, view and projection matrices.
- **Tessellation Shader** This stage consists of three sub-stages. A programmable control shader, a fixed-function primitive generator and a programmable evaluation shader. The tessellation shader operates on patches of vertex data to do dynamic subdivision. This can be used for adjusting the geometric level of detail during rendering.

- **Geometry Shader** The optional geometry shader stage operates on primitives and can output zero or more new primitives. Programs often just pass through the input vertices unaltered and just calculate some per primitive values. This is also called a pass-through shader.
- **Primitive Assembly** This fixed-function stage does various post-processing, like establishing primitive order, on the geometry data, before it is converted to pixels in the rasterization stage.
- **Rasterization** This stage uses fixed-function rasterization units to determine the pixels covered by a projected rendering primitive. Some configuration can be done on recent versions of the graphics API, like configuring the use of conservative rasterization or switching it off completely. In case of a switched-off rasterization stage, the pipeline ends at the geometry stage.
- **Fragment Shader** Per fragment code is executed to determine the color value of the output pixel or if it will be discarded.
- **Per-Fragment Operations** Before pixel data is written to the output framebuffer, some (configurable) fixed-function operations will be carried out, like blending and depth testing.

Three quarters of this pipeline deal with geometry processing, which illustrates the importance this topic. Thus, efficient handling of geometric data and quick execution of involved algorithms is most desirable. Besides, the development history of this pipeline design shows that geometry processing has become more and more important throughout the years. The first user programmable geometry stage was the vertex shader, followed by geometry shaders and then tessellation shaders. Lately, NVIDIA has proposed mesh shaders [52], which enable more efficient geometry processing by supporting cooperative thread groups in the style of compute shaders to work on geometry and dynamically generate levels of detail for meshes on the graphics processing unit (GPU). Looking at these developments in graphics pipelines, one can expect that the capabilities to handle more and more geometry will increase further and that the complexity of geometry generated from shader programs directly on the GPU will become richer in detail, as developers learn to use the novel capabilities better. Some of these new methods of handling computer graphics tasks, like generating geometry directly on the GPU or separating the rendering pipeline into shading and display, are explored within this thesis.

**Geometry processing** is a large field that covers topics from subdividing surfaces over modeling techniques like constructive solid geometry (CSG) to smoothly deform-

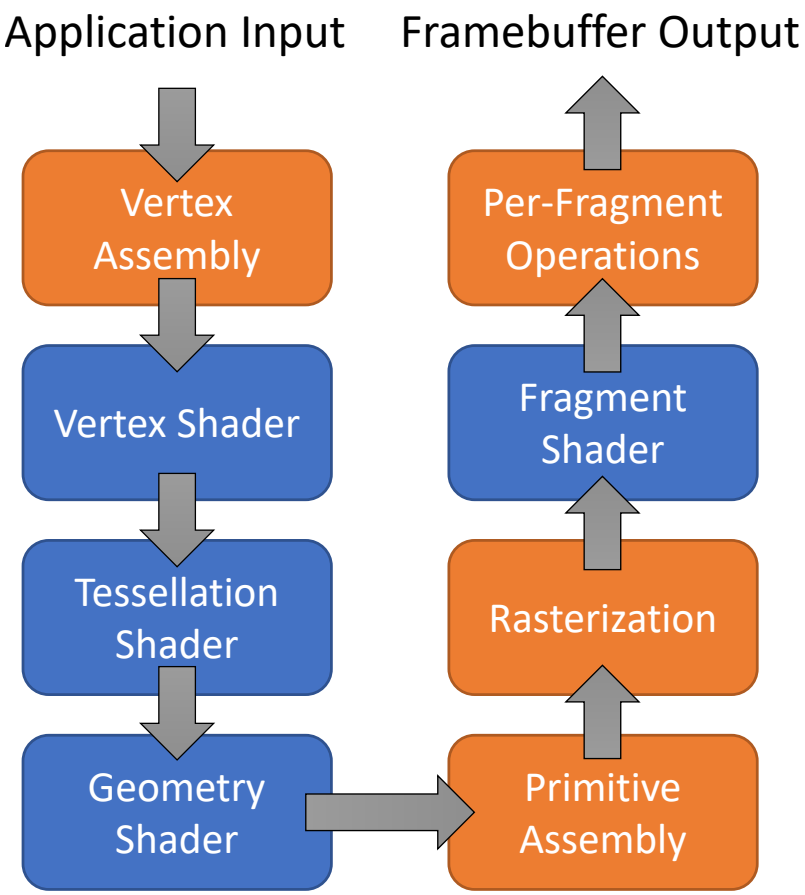



Figure 1.2: A graphics pipeline as it is used in modern graphics API like it is described in the OpenGL specification [43]. The blue stages are user programmable stages and the orange ones fixed funtion stages that can be configured to some extent before issuing drawing commands.

ing meshes, to name just a few. This thesis focuses on a few specific topics that are used to render computer-generated imagery in real-time. We try to avoid setting this focus too narrow to cover a good subset of important topics within the domain of geometry processing from generating models, alternative representations, rasterization, sampling, visibility and decoupled rendering pipelines. Many techniques in real-time graphics exist that increase image quality and realism by operating on textures. They can have a significant visual effect already with relatively low computational overhead or even have gained hardware support because they have been around for quite some time and belong to the standard feature set of any good 3D engine, like various filter or texture mapping effects. In some cases, however, a certain level of detail in geometric representation can not be avoided. An increase in the number of triangles to draw can quickly become a hazardous performance bottleneck. One

way of dealing with this is to adaptively reduce the quality of the used 3D models. Another would be to keep storage requirements low by either applying compression schemes or avoiding data transfers altogether and generate or refine the geometry on the fly. With current hardware being rather limited by memory bandwidth and latency rather than raw compute power, this is often a viable route to go. Therefore, our primary objective is to provide solutions to the various problems that increase efficiency by leveraging highly parallel hardware platforms.

**Programming models** on GPU hardware are not only vastly different from traditional CPU based single or multi-threaded paradigms, but have also evolved with more and more functional blocks of the graphics pipeline being opened up from fixed-function to being freely programmable. Furthermore, the trend of using shader programs for general purpose computing tasks has led the manufacturers of such devices to provide API to use them in a dedicated compute mode. The free programmability of GPU hardware opened up this platform for an even broader range of tasks than just graphics problems. But the massively parallel nature of this hardware makes it a non-trivial task to map algorithms to it. To ease this burden and to increase the efficiency of programs running on the GPU, sophisticated scheduling and careful resource management have to be considered to keep processor occupancy high and memory access wait times low. In this regard, previous work has shown the effectiveness of using software scheduling specifically tailored towards massively parallel architectures, which motivated research in applications of that technology leading to the results that are presented in this thesis.

## 1.2 Objectives

In our research, we consider applications that require geometry processing algorithms with the following properties.

- **G1** The geometry is both large and complex. Consequently, it can not be rasterized directly. For example, this encompasses vector graphics containing curved primitives, implicit modeling through grammars and poorly conditioned triangle sets with tiny slithery primitives.
- **G2** We aim for immediate rendering and, therefore, we want the result of the geometry processing on-the-fly.

The two requirements imply that geometry processing must have both high throughput and low latency. Even though offline geometry preprocessing can be done, the

value of such a solution would be significantly reduced. A preprocessing solution would be restricted by storage bounds and by inability to handle dynamic, animated, or user-generated data sets. From these general requirements, we define several more concrete research goals that meet the properties G1 and G2.

We postulate that we can improve upon the state of the art in various aspects of geometry processing in terms of performance and image quality and derive the following research objectives to be met by the work conducted prior to the writing of this thesis.

- **R1** Since previous work [71] has shown that with the help of appropriate scheduling, the performance of shape grammar derivation systems can be increased, an analysis of this workload's characteristics can further improve the mapping to GPU and therefore the efficiency of execution.
- **R2** Besides the procedural generation of geometry, other problems that show the same pattern of interpreting an input description to produce visual output, must be able to benefit from applying our task scheduling paradigm. To further specify this rather general objective, we choose descriptions of vector graphics illustrations to reinterpret them in a way to foster ideal mapping to GPU.
- **R3** Line equations are used in triangle rasterization to determine if a point is inside or outside of a given primitive. Curve equations in their implicit form can be used in the same way when various precautions are taken. This will not only enable to map the same principle from triangle to curve rasterization, but also allows us to represent curved outlines in compact fashion with a low memory footprint.
- **R4** In order to process just the right amount of geometry from a scene in a rendering pipeline, a good method of determining the visibility of primitives produces a visible set that is *complete enough* to avoid disocclusion artifacts but does not induce excessive overshading. A method that meets these requirements will be superior to simply increasing sampling density, which would be infeasible in terms of performance when doing real-time rendering.

Starting out with an analysis of procedural modeling tasks is a good way to focus on the abstraction of the algorithm to its fundamental operations and their interaction. Since we already know, that parallel execution of such a workload can be highly efficient we concentrate on the question why this is the case and how to tune the break down of the derivation into work items to achieve maximum efficiency. To this end, we postulate that a graph representation of a given shape grammar

will help us to fulfill **R1** and draw parallels to code generation and compilation optimizations. Furthermore, makes sense to not only generate geometry directly where it is consumed—on the GPU—to avoid expensive memory transfers, but also to execute as much of the generation pipeline as possible on the GPU without any round trip delays for central processing unit (CPU) intervention.

In a similar way we handle **R2**, if we appropriately prepare vector graphics illustrations and thus will be able to execute the rasterization process in a highly parallel fashion. By interpreting the contours as regions bounded by curves that do not intersect their enclosed area, each sub-region will be handled independently and in parallel. Again, a graph representation can be a suitable structure to analyze the task at hand. From that graph representation we can decompose the regions to be filled into patches bounded by curves which can be directly processed in a rasterizer that can evaluate their implicit form (**R3**).

So far we have covered creation, rasterization and representation of geometry as well as optimized instruction scheduling in our research goals. In our last objective **R4** we concentrate on the detection of geometry to improve not only visual quality but also the sampling process. We argue that it is not only of importance how to prepare the data and the rendering pipeline for efficient processing, but also to detect what geometry is necessary to feed to that pipeline to achieve high fidelity graphics. The most efficient rendering engine is of no use if the triangles it is supposed to shade are not detected to be visible. Similarly it is of no use to spam the system with work that will not be visible on screen. Consequently, potentially visible set (PVS) algorithms are of interest in the context of virtual reality (VR) and this thesis.

## 1.3 Contribution

The contribution to the various topics of geometry processing for real-time graphics applications achieved by this thesis and its preceding publications and how they relate to the objectives stated in section 1.2 will be laid out in the following list.

- We introduce the operator graph to analyze shape grammars and find an optimal schedule for the derivation process. This not only increases performance in general, but also makes procedural generation more efficient (**R1**) on massively parallel hardware by finding suitable work packages with the help of heuristics to search the vast space of possible solutions.
- Besides procedural generation of geometry, we show that problems of interpreting descriptions for generating graphical output can be tackled by reformulating

them in a suitable manner for processing with a GPU scheduling system (**R2**). We demonstrate this by implementing a vector graphics rasterization system that reinterprets drawing commands to form new rendering primitives that can be used to render simple as well as highly complex illustrations equally well.

- We propose an alternative representation (**R3**) to describe rendering primitives. What triangles are to conventional rasterization in 3D rendering pipelines, CPatches are to vector graphics rasterization. Similarly to using line equations we can use curves for an efficient description of a primitive's boundaries.
- In the advent of decoupled rendering systems to suit the demands of emerging technologies in VR we propose a novel method to sample the visibility of triangles. Specifically, our approach solves the problem missing geometry when sampling, because it would be too small to be detected. By employing conservative rasterization and carefully designed heuristics to filter excess triangles we provide a solution that can meet real-time requirements as opposed to what can be achieved by naively using supersampling (**R4**).

## 1.4 Organization

**First,** an introductory part contains a motivational statement that portrays the situation and the problem to be solved. Chapter 1 contains this as well as research objectives, a list of achieved contributions and organizational details about the thesis.

After successfully introducing and outlining the achievements of this thesis, we will take a little detour in Chapter 2 to discuss background information and previously conducted research in the fields covered by the individual subtopics. This shall give the reader pointers for further reading as well as help to distinguish the accomplished contribution.

**Second,** before starting out to the three in-depth chapters on geometry processing, in Chapter 3, we provide an outline of what is to come. The publications, which are also listed in Appendix **??** and a statement of collaboration is provided with a short description of each individual subtopic. The two topics, GPU scheduling and decoupled rendering are covered in more detail in the overview, to give the reader essential knowledge upfront. The detailed description of the work follows in subsequent chapters. The geometry generation in Chapter 4, representation and sampling in Chapter 5 and visibility sampling in Chapter 6. These three chapters closely resemble publications the author worked on prior to writing up the thesis.

With this arrangement, we try to draw the picture for geometry's life cycle from it's creation via procedural modeling, over storing it to memory in some representation to transforming it to visual output in rasterization and consuming primitives in sampling operations.

**Third,** Chapter 7 concludes the thesis with a summary and closing remarks on limitations and future work. Lastly, there is the bibliography listing references to other publications.

## 1.5 Publications and Statement of Collaborations

A brief statement of collaboration in the work that has been done and the publications produced will be mentioned here in the introductory chapter in chronological order in the style of title (in bold), authors (thesis author in bold) and venue/journal where the work was published. The author of this thesis is meant when "the author" is mentioned in the statement of contribution, not the main author of the publication.

### Whippletree: Task-based Scheduling of Dynamic Workloads on the GPU

Steinberger, M., Kenzel, M., Boechat de Almeida Germano, P., Kerbl, B., **Dokter, M.**, Schmalstieg, D.
*November 2014: ACM Transactions on Graphics (TOG)—Proceedings of ACM SIGGRAPH Asia 2014 Volume 33 Issue 6 Article 228*

The author contributed implementation work of the procedural modeling show case of the described method. The co-authors did the major part of implementation and the paper writing.

### Representing and Scheduling Procedural Generation using Operator Graphs

Boechat de Almeida Germano, P., **Dokter, M.**, Kenzel, M., Seidel, H-P., Schmalstieg, D., Steinberger, M.
*November 2016: ACM Transactions on Graphics (TOG)—Proceedings of ACM SIGGRAPH Asia 2016 Volume 35 Issue 6 Article 183*

The author shared the implementation effort with the paper's main author and carried out supportive work in paper writing. The co-authors also provided ideas,

expertise and useful discussion.

## Shading Atlas Streaming

Müller, J., Voglreiter, P., **Dokter, M.**, Neff, T., Majar, M., Steinberger, M., Schmalstieg, D.
*December 2018: ACM Transactions on Graphics (TOG)—Proceedings of ACM SIGGRAPH Asia 2018 Volume 37 Issue 6 Article 199*

The author contributed development effort to the software framework and conducted experiments to evaluate the described system against other state of the art methods. he co-authors contributed implementation work, paper writing, ideas and guidance.

## Hierarchical Rasterization of Curved Primitives for Vector Graphics Rendering on the GPU

**Dokter, M.**, Hladky, J., Parger, M., Schmalstieg, D., Seidel, H-P., Steinberger, M.
*May 2019: Computer Graphics Forum (Proceedings EUROGRAPHICS). To appear.*

The author spent the majority of the projects lifetime on implementation work. The writing was completed in cooperation with Markus Steinberger. The other co-authors contributed development effort, valuable comments and guidance.

## Real-Time Sub-Pixel Visibility for Decoupled Shading

**Dokter, M.**, Müller, J., Voglreiter, P., Neff, T., Steinberger, M., Schmalstieg, D.
*2019: pending submission IEEE Transactions on Visualization and Computer Graphics—TVCG*

The author implemented the described method and did most of the writing of the publication. The co-authors provided the shared effort in creating the rendering framework, guidance in overcoming development obstacles, expertise and experience in general.

# 2 Related Work

## Contents



The various topics of geometry processing, GPU programming and scheduling discussed in this thesis have been subject to previous research work, which will be considered on the following pages.

## 2.1 Procedural Generation

Procedural modeling has been the focus of research projects for decades, but only in recent years several ways of parallelizing generation of geometry using shape grammar descriptions have been proposed.

### 2.1.1 Computer aided procedural modeling

Stiny's original *shape grammars* [75] were one of the first approaches which applied a sequence of procedurally defined operations on a given set of shapes. Later on, Stiny refined this work to define *set grammars* [76]. While shape grammars consider operations on shapes, similar procedural generation processes can be defined on

character strings. These approaches, called *L-systems*, are inspired by plant growth patterns [60]. Similarly, Wonka et al. [88] found that facades can effectively be described by *split grammars*, a special case of shape grammars, which restrict the available operations to space subdivisions. Combining concepts from the original shape grammar work, L-systems, and split grammars, *CGA shape* [49] is able to describe complex buildings and cities. While the work on shape grammars and L-systems spans three decades, the idea behind productions generation remains unchanged. Given a shape or symbol, an operation is applied to that entity to yield new shapes/symbols.

One of the most important extensions to simple procedural generation is considering external influences, such as the environment [59], user-defined curves [61], external guidance [6], or vector fields [37]. In contrast to external influences, interactions between generated objects can be modeled, such as interconnecting different structures [31], resolving intersections of generated geometry [57], or querying neighboring shapes [49]. Treating shapes as first class citizens allows a variety of queries between generated shapes and even temporary objects [69]. Procedural modeling can also be extended to support more general terminal symbols [32] or more basic mesh editing functions [23]. While these extensions introduce additional dependencies into the generation, the very basic principle of procedural generation remains unchanged.

### 2.1.2 Parallel evaluation

The basis for computation on the GPU is how well an approach can be executed in parallel. L-systems and shape grammars, which form the foundation of many procedural generation systems, naturally provide a high degree of parallelism. In L-system derivation, every symbol can be worked on independently. Shape grammar evaluation follows a tree-like derivation, where different nodes in the tree can be worked on in parallel. While we are interested in using this parallelism for execution on the GPU, parallel CPU approaches are, of course, also possible [91].

### 2.1.3 Grammar evaluation using GPU shaders

Shape grammar evaluation with shaders started with the work by Lacz and Hart [34]. They derive their split grammar using vertex and pixel shaders, which are relaunched using a render-to-texture loop. They sort intermediate symbols between shader launches to provide the GPU with a homogeneous workload. Without this step, control flow splits into multiple branches, leading to thread divergence and slowing down the execution on the single instruction, multiple data (SIMD) units of the GPU.

As every symbol on the same level of the tree can be worked on in parallel, good performance should be achievable. However, the intermediate sorting steps dominate the execution time, significantly lowering the performance. A similar approach was proposed by Magdics [41] for L-system derivation, which uses multi-pass rendering and GPU stream output to collect and sort intermediate symbols. The probably most advanced shape grammar evaluation system using shaders is the approach by Marvie et al. [46]. They use a combination of vertex, geometry and fragment shaders to derive shape grammars similar to CGA shape. For their derivation approach, they use a fixed-size stack in the vertex shader. This approach reduces the available parallelism to the number of input axioms.

### 2.1.4 General purpose GPU languages

General purpose GPU languages have also been used for grammar derivations on graphics processors. One of the first approaches to use NVIDIA Compute Unified Device Architecture (CUDA) for L-system derivation was the system by Lipp et al. [39]. Their approach is similar to the traditional shader-based approaches. During one kernel launch, each thread interprets one symbol. Prefix sums are run between launches to collect symbols in memory. As symbols are not being sorted, their approach has severe problems with divergence and could not achieve speed-ups over an optimized CPU approach, when the grammar was complex.

Recently, Steinberger et al. [72] proposed an approach that can perform an entire shape grammar derivation within a single kernel launch. Intermediate symbols are stored in a queue in global graphics memory, while threads are running in a loop, pulling shapes from the queue and pushing newly generated shapes back into the queue. Their approach handles additional interdependencies in the evaluation, such as context sensitivity, by transforming them into redundant parallel evaluation or splitting the generation process into multiple stages.

### 2.1.5 Per-pixel grammar evaluation

Alternatively, per-pixel grammar evaluation creates parallelism for grammar derivation by evaluating the rule-sets for every pixel on screen. This guarantees that work is only completed for visible parts of the scene, and a sufficient amount of parallelism can always be found. However, this approach is only applicable, if a coarse spatial separation of geometry is already available. Furthermore, the grammars have to be reevaluated for every rendered frame. Thus, this approach has only been used for facade textures [22, 45]. While a significant amount of parallelism is guaranteed, it

also leads to a significant number of redundant derivations for neighboring pixels. In case different facade parts are hit by neighboring pixels, divergence can become an issue.

## 2.2 Vector Graphics Rasterization

Previous curve rasterization techniques can be roughly classified into three categories: (1) scanline filling methods, (2) 'stencil, then cover' approaches, and (3) alternative vector graphics representations, such as data structures supporting spatial queries.

### 2.2.1 Scanline methods

Early scanline algorithms focus on rendering triangles [89] and construct spans limited by pairs of edge-scanline intersections. To increase the efficiency of scanline algorithms, the intersections of a scanline with all edges can be computed before sorting and filling [51, 1].

CPU scanline algorithms for vector graphic rendering are found in contemporary curve rendering packages such as Skia [20] or Cairo [55], which are used if no appropriate GPU accelerated alternative is available. Manson and Schaefer [42] used pixel-sized scanlines to implement analytic shading and anti-aliasing filters. To increase performance, spans can be merged and clipped for hidden surface removal, as shown by Whitington [87].

While scanline approaches are usually designed for the GPU, Li et al. [36] recently showed that a GPU scanline algorithm can also be efficient. Their approach first builds an acceleration data structure of potential scanline-curve intersections and then evaluates them in parallel with simplified geometry. A final step rendered the generated spans using traditional OpenGL. The efficiency of this algorithm comes from the fact that not every filled pixel must be tested against the path. CPatch has the same advantageous property, while requiring only a single pass.

### 2.2.2 Stencil, then cover

Many GPU curve rendering approaches follow a 'stencil, then cover' approach, where a mask is first generated for a path (stencil) before filling, while blending happens in a second step (cover). Stencil generation goes back to the work of Loop and Blinn [40], which allows to efficiently determine on which side of a curve a sample lies. In combination with Jordan's theorem [16], fill rules can be computed in a discrete

manner, which allows complex path stencils to be generated by rendering multiple overlapping triangles with implicit curve descriptions [30, 29]. While hardware support makes this method fast, it still requires a per-path multi-pass algorithm that potentially touches many samples which are not part of the final stencil. Note that the scanline approach by Li et al. [36] can be classified as "stencil, then cover" as well.

Several extensions exist: For example, the 'stencil, then cover' method used in Adobe Illustrator [5] extends color schemes and blending modes. Tile-based rendering of stencils [92] runs efficiently on mobile devices.

Like these approaches, our method classifies half-spaces, but it directly renders paths from CPatches, rather than using a separate cover pass. In that sense, our approach is closer to the original method of Loop and Blinn [40]. However, their method only supported a single curve per primitive, which makes it prohibitively complicated to construct complex shapes, like thin parallel curves. Our approach supports multiple curves and draws further efficiency from hierarchical rasterization.

### 2.2.3 Alternative representations

Various alternative representations of vector graphics have been proposed. Motivated by rendering vector graphics on top of surfaces, vector texture methods try to encode sharp features in regularly sampled textures. Feature curves [56] encode distances to quadratic Bézier curves and can thus render a limited number of sharp curves intersecting at one location. Precise vector textures [63] encode the distances to monotonic curve segments in the texture. As long as the distance to the evaluated curves is not larger than the curve's curvature, the method delivers error-free results. Vector solid textures [84] use radial basis functions as primitive to construct sharp features. All the above representations allow highly flexible display transformation and are efficient to render using texture hardware. However, they cannot represent arbitrary vector graphics due to their limitation to the sample grid of the underlying texture or curves.

Nehab and Hoppe [50] use an adaptive lattice structure to describe vector graphics with appropriate detail where needed. They support distance evaluation in the lattice cells to linear, radial and quadratic curves. Cubic curves are not supported. While their lattice generation is carried out on the CPU, the rendering is performed on the GPU. Their lattice structure reduces the number of curves that need to be tested for each fragment, but still requires that all pixels within a potentially large cell be tested against all curves of that cell.

Shortcut trees [19] allow efficient indexing into vector graphics. They can be built on the GPU and support cubic curves by monotonizing curve segments. While their tree representation is elegant and relatively fast to build, the resulting rendering performance can compete with hardware-supported 'stencil, then cover' strategies only at very high resolutions.

Diffusion curves [54, 18, 78] let a designer construct path outlines that implicitly control the color of the interior through a simulated diffusion process. This approach lends itself to parallel solving, but remains very computationally demanding overall.

Our method does not build an auxiliary data structure, but converts the vector graphics data entirely into a set of new primitives supporting an object-order approach, rather than being constrained to image-order.

## 2.3 Visibility

Our method of sub-pixel visibility sampling, described in Chapter 6 tackles the problem of determining triangle visibility in a new way using conservative rasterization and solves a problem of decoupled shading. What other research has been done in visibility, decoupled shading and conservative rasterization will be briefly discussed in the following subsections.

### 2.3.1 Visibility Algorithms

Visibility is an important aspect of computer graphics pipelines as it is crucial to focus computation on what eventually will be displayed on screen. [7] give overview of the many methods that have been proposed to address challenges in this field. Anti aliasing in the context of visibility determination is needed to fight aliasing at sharp edges of a scene's geometry. This is done by increasing the sampling per pixel which increases not only memory footprint but also run time considerably. Therefore it is desirable to reduce the amount of sampling to a required minimum for example by decoupling the visibility sampling to gather scene depth from expensive shading [85].

### 2.3.2 Decoupled Shading

Decoupled rendering [64] describes the idea that shader evaluation and image generation are run as separate passes, for example, in a deferred rendering framework. Various image-based models can be employed to connect the two passes. Usually,

the image generation uses color and depth buffers and, possibly, proxy geometry [44, 15, 8, 35, 66]; others try to do the shading in object space and reuse the shaded primitives for rendering the final output.

While the image based rendering (IBR) methods work from fully rendered frames, object space shading methods need to decide upfront what to shade and therefore need to decide what might be visible when the output image is rendered. A rendering cache can, for example, rely on spatio-temporal coherence [64]. Many object space shading methods use complex software implementations that are too slow for real time use [3]; [10]; [11]; [9] and would require dedicated hardware acceleration to work with feasible speed. Some implementations use visibility determination to enable efficient implementations on modern graphics hardware, which makes the technique fit for real time VR rendering [38]; [4]; [24].

# 3 Overview

## Contents



In this chapter, a brief overview of the work to be described in this thesis will be given. The core areas in this thesis, namely procedural generation, rasterization of curved primitives representation, and visibility sampling of geometry, will each be discussed in a full chapter. Two topics that concern enabling technology and system architecture will be covered in a more in-depth description in this overview chapter to familiarize the reader with concepts of GPU scheduling and decoupled rendering.

## 3.1 GPU Scheduling

When trying to increase the performance of an algorithm by parallel execution, not only concurrency issues ought to be considered, but also how to orchestrate the collaboration of individual tasks on different levels of granularity supported by the underlying platform. The framework used in this thesis will be briefly summarized in this section.

### 3.1.1 The Whippletree Scheduling Framework

Our framework is built around the concept of a persistent threads *megakernel*. This is a style of compute kernel execution, where all functionality is provided as subroutines

of a single kernel function. Threads run in a loop continuously to draw work items from a queue. Execution stops once all queues are empty.

This implementation, the Whippletree Megakernel (WMK), supports multi-programming of different tasks on top of CUDA. It respects the constraints of different task types, while ensuring high performance. Furthermore, we also propose easy-to-use mechanisms for load balancing, data-locality-aware queuing, as well as configurable scheduling strategies.

Parallel execution on the GPU is organized into a three-level hierarchy (see Table 3.1) At the lowest level, small groups of cores operate in a SIMD fashion. While it is possible for control flow within the same SIMD group to take different branches, execution of these branches has to be serialized, leading to a condition known as *thread divergence*. At the intermediate level, multiple SIMD groups are organized into a *streaming multiprocessor (SM)*. Each SM contains a small amount of fast local shared memory accessible to all SIMD groups on the SM. At the top level, the GPU itself consists of multiple SM units and is connected to global graphics memory. Two programming models are prevalent on this hardware: shading languages and compute languages.

The main contribution of Whippletree is its fine-grained resource scheduling and its ability to exploit sparse, scattered parallelism. Complex branching or recursive graphics pipelines as well as algorithms processing hierarchical data structures can be scheduled such that SM units are efficiently filled with coherent workloads. Moreover, the Whippletree model allows full multiple instructions, multiple data (MIMD) task-parallelism without problematic interleaving of multiple kernels. Tasks can be created dynamically and scheduling considers data locality.

As WMK inherits some traits from persistent megakernels, blocks of threads (worker-blocks) are launched to exactly fill up all multiprocessors. These worker-blocks execute a loop drawing tasks from work queues. Procedures are implemented

| task type | thread count | feature set |
|---|---|---|
| level-0 | $2..M$ | warp-level |
| level-1 | $M..B$ | block-level |
| level-2 | 1 | global |

Table 3.1: Overview of the three task types supported by our programming model in decreasing order of the associated feature set. $M$ denotes the warp size of the device and $B$ the maximum block size.

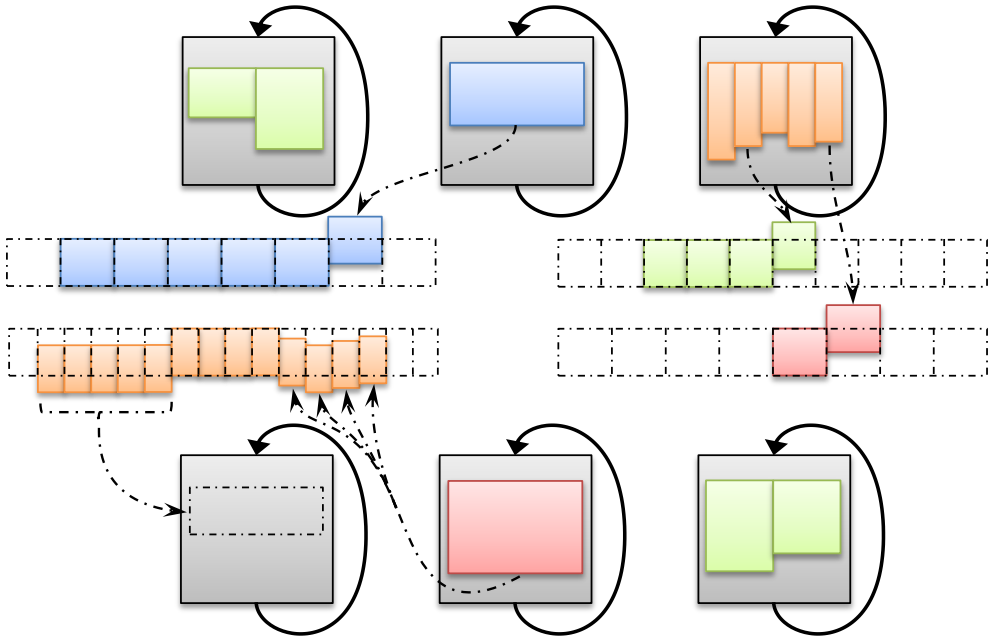


Figure 3.1: Whippletree consists of worker-blocks, continuously drawing tasks from queues. We use one queue per function, which is essential for a divergence-free execution of different task types. During execution, new tasks of any kind can be created.

as branches in the main loop. When new tasks are generated, they are inserted into the queues. In this way, concurrent execution of multiple tasks, *e.g.*, multiple stages of a pipeline, is supported. Load balancing between worker-blocks is achieved through the work queues.

With Whippletree, we have a remarkably versatile framework at hand that allows for scheduling tasks at various granularities of parallelism that map the underlying hardware. We use this software scheduling method to implement our work on operator graph analysis, described in Chapter 4, and develop our hierarchical rasterization scheme for vector graphics rendering, as described in Chapter 5

## 3.2 Procedural Generation

In chapter 4 we present the findings of our research on shape grammar derivation on the GPU. While we already used the Whippletree framework to evaluate shape grammars in the paper focusing on the scheduling framework [73], we investigate potential space for optimizing this process even further by using a more general

approach of analyzing the input in a graph representation.

## 3.3 Representation and Rasterization of Vector Graphics

In Chapter 5 we show how to reinterpret common vector graphics descriptions to generate new data structures named CPatches. Similarly to triangles, which are represented by three lines, a CPatch represents a shape by a set of Bézier curves.

As with triangles, where the line equations are used to determine on which side of the line a point is to determine the inside and the outside of that primitive, the implicit form of Bézier curves is used to evaluate the filling of a shape that is described by a CPatch.

We developed a novel method of rasterizing vector graphics that can process the CPatches. Our hierarchical rasterization scheme is very well suited for execution on GPU hardware. We use the Whippletree framework again to achieve remarkably fast rendering times that beat state of the art methods in many cases.

## 3.4 Visibility Sampling for Decoupled Rendering

Another discipline where we saw potential for improvement is in sampling of geometry for visibility, very small geometry, to be specific. When triangles become so small that can not be seen from the current view point, they might not be shaded in the current frame. In a decoupled rendering scenario this might be a problem, when a subsequent frame is to be rendered from the shading output of the current frame, and triangles that have become visible in the new view point can not be displayed due to the lack of shading information.

In Shading Atlas Streaming (SAS), the server GPU fills an atlas with shaded pixels corresponding to just the visible triangles. The client GPU performs a final geometry pass, but samples the shading information from the atlas rather than invoking expensive fragment shaders.

To understand the design constraints of a remote rendering system using object-space shading, we begin with an overview of the server-client pipeline ( Figure 3.2). The end-to-end latency for a round trip from client to server and back can be broken down as follows:

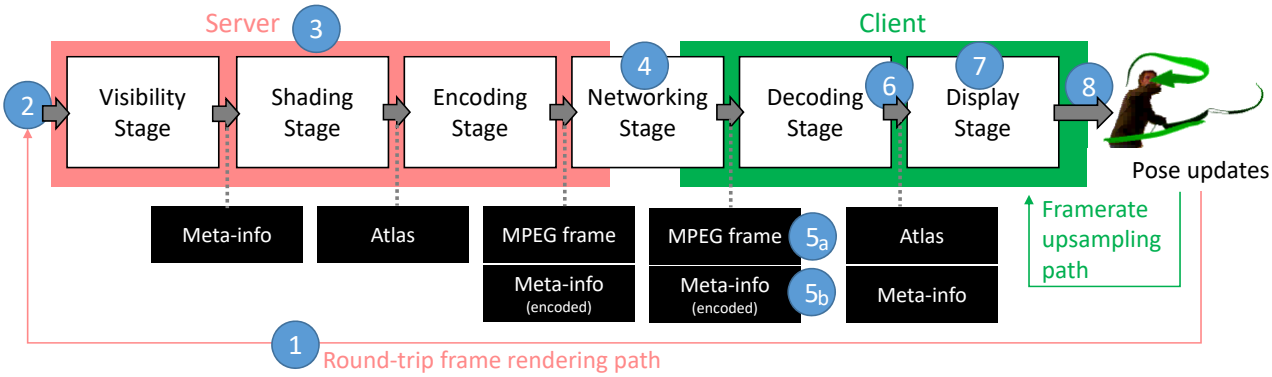



Figure 3.2: Our pipeline is split across a server and a client part, with the shading atlas as the central data structure connecting the two. Camera pose updates generated by the user are sent upstream, on a slow (networked) path to the server for rendering new shading into the atlas, and on a fast (direct) path to the client to render new images to the display.

1. The client sends the current view matrix to the server. This duration is owed to network latency. It can be changed with faster networking technology.
2. The message waits at the server until a new frame starts rendering. This duration depends on how frequent the client sends pose updates, since the most recently received pose is used for the next frame.
3. The server renders the atlas frame. This duration corresponds to the processing times for visibility, shading and encoding stages. It depends on server CPU and GPU performance.
4. Atlas frame and corresponding meta-information are sent to the client. This duration is owed to network latency.
5. The client decodes (5a) the atlas frame and (5b) the meta-information. This duration corresponds to the maximum of client's processing time for the two decoding tasks, since they are carried out by different hardware units. The duration depends on client CPU and GPU performance.
6. The received data waits at the client until a new frame starts rendering. This duration depends on the client frame rate and varies between 0 and the client frame time.
7. The client renders the final frame. This duration corresponds to the client's processing times for the display stage. It depends on the client's CPU and GPUperformance.

8. The final frame is displayed. The most recently received server frame is used to render additional frames at the client (*frame rate upsampling*). Thus, this duration increases with every upsampled frame, until a new atlas frame arrives.

The SAS architecture forms the basis for the work on sub-pixel visibility presented in Chapter 6. However, to focus on the problem of determining the visibility of triangles, the networking components the original implementation have been removed.

For simplicity, the architecture, for doing experiments in visibility sampling, has been reduced to a visibility stage, shading stage and a display stage. Since there is no need for a network layer, we implemented just a split rendering pipeline where the server part consists of visibility and shading and the client part, running in the same program, handles the display part asynchronously.

In contrast to the work on operator graphs and CPatches, the implementation of this method was not done in CUDA and Whippletree, but in Vulkan.

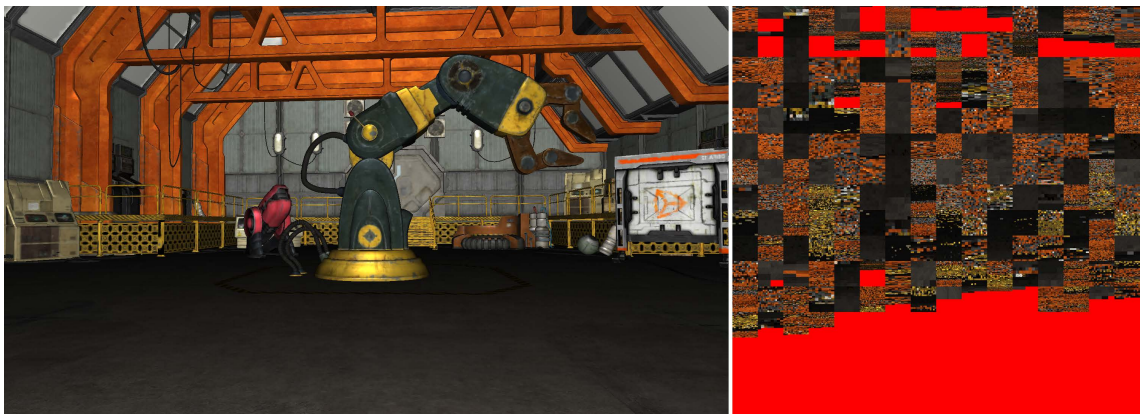


Figure 3.3: Game scenes (top row) with corresponding shading atlas (bottom row). The shading atlas contains all the shading information of the visible surfaces corresponding to the rendered scenes. The object-space parametrization is created fully dynamically. From the shading atlas, novel views at close viewpoints can be rendered for framerate upsampling and warping. The shading atlas is temporally coherent and lends itself to efficient MPEG compression and streaming.

# 4 Procedural Generation

## Contents



In recent years, content creation for virtual worlds has become increasingly limited by human effort, rather than technology. Manually crafting every detail of vast virtual worlds for games and feature films is tedious and time-consuming. Thus, it is not surprising that procedural generation is becoming more widely adopted in the digital content creation industry, shifting part of the labor from the designer to an automated system. Using a procedural approach, complex models can be created from small procedural programs or rule sets. A simple program written by an expert can generate a large number of plausible variants of a model type, *e.g.*, buildings for an entire city.

Procedural generation methods are present in all phases of a content authoring pipeline. In the design phase, tools like automatic object placement and style transfer [21] may evaluate procedural programs hundreds of times to match high level modeling goals. Similarly, Metropolis procedural modeling techniques [80] may execute thousands of parameter sets to tune a model towards a target function. In the deployment phase, a program might be evaluated millions of times, as the user moves through a continuously generated procedural world [71]. Analogously, during the rendering of a movie, procedural models might be evaluated on-the-fly for every

frame, because reevaluation is more cost-efficient than keeping models around. While the aforementioned applications use different procedural generation methods, they all require the evaluation to run as fast as possible to increase usability, frame rates or production time.

To speed up the evaluation process, procedural generation has been brought to the GPU before [39, 46, 72]. However, previous approaches to procedural generation on the GPU only focused on specific methods. We believe there is a need for systematic parallelization of general procedural methods, which ensures that the generation process is always efficient without having to rely on manual fine-tuning.

Figure 4.1: Using the operator graph, we optimize the procedural generation of an entire city containing 50 000 buildings and 10 million triangles. The original shape grammar takes more than five minutes to generate on the CPU. Generation on the GPU according to previous work takes 630 ms. Applying our optimization framework, we reduce the generation time to 109 ms by only changing the way we schedule the underlying operations on the GPU.

To this aim, we describe a unified formal model for procedural generation systems, which can be used to describe a variety of processes: the *operator graph*. Its practical importance lies in its ability to analyze different ways to execute the generation process on the GPU. Our work makes the following contributions:

- We introduce the *operator graph*, which is a common representation for a variety of procedural generation methods and is independent of a particular way a designer might write a program or describe a generation process.
- We establish a general *scheduling* method for running procedural generations on the GPU and evaluate how static or dynamic scheduling of the operator graph influences load balancing and data locality.
- We show that each schedule corresponds to a *partition* of the operator graph and demonstrate how existing GPU approaches can be reduced to a specific partitioning of the operator graph.
- We show how to *optimize* the scheduling of a procedural generation system by finding the operator graph partition that achieves the best performance on a given hardware. As the number of partitions is growing exponentially with the size of the graph, we propose a set of heuristics to constrain the search space to a manageable size.

To validate our results and show that the operator graph can be used to parallelize different types of procedural generation systems, we analyze graphs of varying sizes and with different characteristics. We fully analyze the space of possible partitions for a number of small graphs verifying the proposed heuristics. For large graphs, we show that the operator graph partitioning has considerable influence on the execution time. We show that an optimized schedule is up to 14× faster than hand crafted solution. We conclude that optimizing the scheduling based on the operator graph leads to the currently fastest procedural generation evaluation on the GPU.

## 4.1 Operator graph representation

While a wide variety of procedural generation methods for different application domains has been proposed, their execution model can most often be represented with a rather simple graph. Consider the following methods: L-systems [60], which were developed to model plant growth, define expansions on symbol strings. Shape grammars, like CGA shape [49], which are well suited to model buildings, define spatial relationships between shapes. Stack-based generation languages, such as GML [23], which are well suited to model detailed man-made objects, define a list of fine-grained modeling operations on polyhedra.

The common factor among them is that they all describe sequences of operations applied to objects. This observation creates a link to data flow programming [83] or

the stream processing abstraction [79], in which programs are defined as directed graphs. Graph-based abstractions have been used for procedural generation before. For example, commercial products like Houdini by Side Effects Software or Autodesk Maya use data-flow networks. Directed acyclic graphs have been used for simple shape grammars [58]. However, these graphs do not support more complex generation methods or offer the information needed for efficient scheduling. To this aim, we introduce the *operator graph* as an intermediate representation that captures the requirements a procedural approach imposes on a scheduler. In this section, we describe the basics of the operator graph. A detailed account how it can be used to represent the aforementioned procedural modeling methods is given in 4.3.

**Graph definition** We model the procedural generation as a directed multi-graph $G = (V, E, D)$, whereas $V = \{v_k\}$ describes the vertices of the graph, $E = \{e_j\}$ corresponds to directed edges modeling the flow of objects in the graph, and $D = \{d_j\}$ is an additional set of directed edges introducing dependencies in the graph. Similar to data flow programming and stream processing, the nodes in the graph describe operations that are applied to objects that travel along the edges $E$ of the graph. Dependencies $D$ introduce restrictions on the order of the operations. Alongside the multi-graph, sets of supported operations $R$ and object types $O$ are needed to describe the procedural generation. Both $R$ and $O$ may differ strongly between generation methods and implementations of those methods.

An **edge** $e = (v_s, v_d, O_e, m_e)$ is an ordered quadruple, connecting a source vertex $v_s \in V$ to a destination vertex $v_d \in V$, and defines a path an object can take through the graph, *i.e.*, there is a possibility for $v_s$ to output an object that will be input to $v_d$. $O_e \in O$ defines the set of objects that can travel along the edge, *i.e.*, objects that may be input to $v_d$. $m_e$ is called the multiplicity of the edge, describing the number of objects that will move over an edge concurrently after a single invocation of the operation associated with node $v_s$. A multiplicity set to a static number indicates that a specific number of objects will always be generated by the node; / indicates that either 0 or 1 object will be generated, and * indicates that an arbitrary number of objects can be generated (including 0). The source vertex for an edge shall be given by $s(e) = v_s$, the destination vertex, by $d(e) = v_d$, and its multiplicity, by $m(e) = m_e$.

A **vertex** $v = (r, (p_1, \cdots, p_n))$ is described by an operation $r \in R$ and a list of parameters $(p_1, \cdots, p_n)$, which influence the operation. Whenever an object moves over an incoming edge to $v$, the operation $r$ is invoked on that object with the node's parameters; we say the object has been consumed by the node. Every invocation

might output any number of objects; we say the objects are produced by the node. The types of objects consumable by an operation $r \in R$ and the types of producible objects can be defined by a left-total, binary relation $IO_r \subseteq O \times O$. The operation defines the number of outgoing edges of the node; for each possible output, an edge must exist in the graph.

**Examples** In a shape grammar, a *translation* operation in $\mathbb{R}^3$ moves an object along a direction given by three parameters. It consumes and produces a single object in $\mathbb{R}^3$. A translation node has one incoming edge and one outgoing edge, each being limited to the same set of objects. In a *string rewriting* system, an operation rewrites characters defined in the alphabet $\Sigma$ according to matching rules $R$. It consumes a character and outputs as many characters as there are parameters. Both incoming and outgoing edges are limited to single characters. The outgoing edge multiplicity is equal to the number of parameters. A *triangulation* operation consumes any flat polygon, takes no parameters and triangulates the interior of the polygon. The associated node has a single input edge, which is limited to polygons, and a single outgoing edge with multiplicity $^*$, which allows for triangles only. A *random path* operator chooses randomly between different outgoing edges with the parameters describing the likelihood for each option. A random path node consumes any object type and produces objects of the same type. For every likelihood given as parameter, an outgoing edge with $m = /$ exists.

A **dependence edge** $d = (v_s, v_d)$ is an ordered pair, connecting a source vertex $v_s \in V$ to a destination vertex $v_d \in V$, and defines a secondary graph on top of $V$. Dependencies model side effects in the operator graph that cannot be captured by $E$. Such side effects describe influences the operation from the source vertex $v_s$ can have on the operation of $v_d$. For example, the generation of a wall ($v_s$) may limit an operation $v_d$ that controls the growth of a tree. Dependency edges can be seen as additional parameters to the operation of $v_d$, which are set up dynamically, as other objects move through the graph. If a dependency edge exists between nodes, the system executing the operator graph has to make sure that there is no possibility that any objects being present somewhere in the graph might still go through $v_s$, before executing objects waiting at $v_d$. In the example above, this means that the generation of trees has to wait, until all walls have been generated.

For a vertex $v$, the incoming edges shall be given by $in(v) = \{e \in E | d(e) = v\}$, and the outgoing edges, by $out(v) = \{e \in E | s(e) = v\}$. A vertex with $in(v) = \varnothing$ is called source node, and a vertex with $out(v) = \varnothing$ is called terminal node. Source nodes correspond to operations that start the procedural generation by introducing

initial objects into the graph. Terminal nodes correspond to operations that end the generation process, by discarding objects or outputting objects as part of the generated model.

**Fractal Example** Consider the recursive generation of the Menger sponge, shown in Figure 4.2, with the operator graph in Figure 4.3. The generation can be completed by using axis-aligned boxes with a recursion counter: The required objects can be given by the tuple $o = (\mathbf{s}, \mathbf{t}, c)$, The generation starts at the blue source node with a single outgoing edge with a multiplicity $*$, indicating that any number of initial boxes might come from the source node. Each box is split along the $X$, $Y$, and $Z$-axis, creating three sub-boxes each, leading to a grid of $3 \times 3$ equally sized boxes. Seven of these boxes are discarded, and the remaining twenty boxes are split anew, unless a fixed number of recursions have been carried out. The recursive nature of the generation is captured by a cycle in the graph. The graph does not contain additional dependencies.

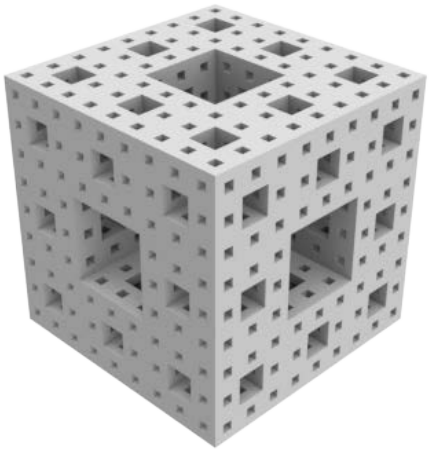


Figure 4.2: The Menger Sponge generated by a recursive generation program stopped at level 3.

**L-System Example** An L-system is defined over an alphabet $S$, axioms $a \in S$ and production rules $P$, which take a character $s \in S$ and map it to a string $(s_1, \ldots, s_n)$ of characters $s_i \in S$, see Lindenmayers's original algae example in Figure 4.4 (a). The result of an L-system evaluation is a string of characters after a certain number of iterations. A system for L-system evaluation can be described as follows. An object is modeled by a character, its position on the string and a rewrite counter: $O = S \times \mathbb{N}^2$. Only a single parameterized rewrite operation is needed, which produces one object (character) for each parameter. It also increases the recursion count and updates the

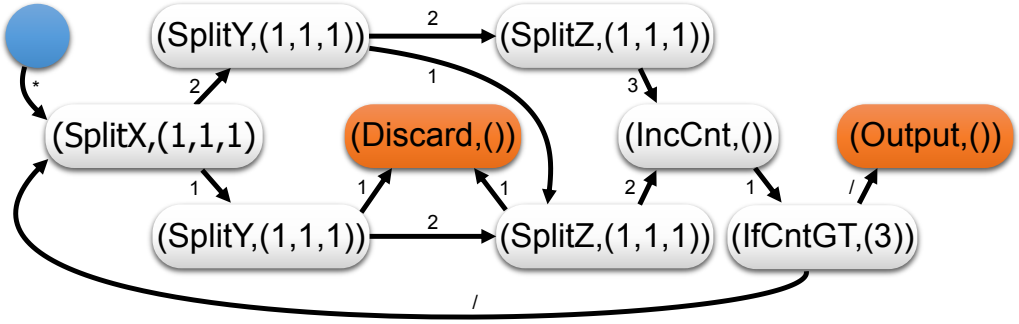
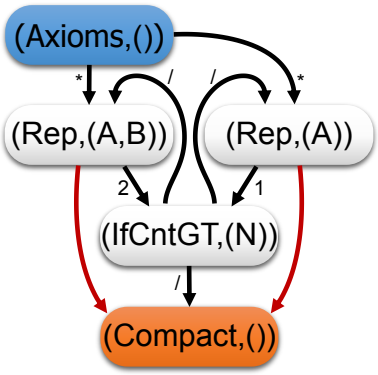



Figure 4.3: The operator graph for the Menger Sponge. The blue node is the source; orange nodes are terminals. Each node is annotated with its operation and parameters. For each edge, its multiplicity is given: * indicates a variable number of objects can be produced, / stands for 0 or 1 produced object. No dependencies are present.

```
Alphabet: A, B
Axiom: A
Production rules:
1: (A -> AB)
2: (B -> A)
```

(a) L-system




(b) Operator Graph

Figure 4.4: Simple algae L-system and its translation to an operator graph. Red edges indicate dependency edges.

string position. As the string position depends on the operations that are carried out for all characters to the left, we compute a conservative string positions and leave empty spaces in the string, *i.e.*, the operator multiplies the position of the input with the maximum number of characters that are produced by any operator. After the desired number of rewrites, a compaction operator removes the empty spaces and forms the output string. To stop the generation, a conditional operation is required, see Figure 4.4 (b).

Every production rule, the recursion check, and the compaction step translate into a node in the operator graph. As the compaction requires all other objects to exist first, a dependency edge is introduced in the graph. This dependency is

a typical example of a side effect. The operator graph does not only contain the original L-system, but describes a full system that enables the interpretation of the L-system. For example, if one would want to describe the parallel L-system derivation by Lipp et al. [39], one would introduce a prefix sum node (replacing our compaction step) within the recursive cycle. Note that the axioms themselves are not part of the operator graph, and thus it describes all possible algae generations from any combinations of starting symbols and not only a static scene.

## 4.2 Operator graph scheduling

Previous approaches proposed very specific ways of running procedural generation on the GPU, trying to include different aspects of best practices in GPU programming. Foremost, this includes (1) supplying enough parallelism to fully utilize the GPU by load balancing between the execution cores, (2) avoiding thread divergence, and (3) avoiding costly memory transactions. While previous approaches try to achieve these goals in very different ways, we generalize these techniques using the concept of the operator graph.

From an operator graph view, a system for procedural generation on the GPU provides a user with the definitions of supported operations $R$ and objects $O$. The user specifies the generation process in a textual or graphical way. This step corresponds to setting up the operator graph and handing it to the generation system. The system must be able to manage objects that are produced throughout the generation process, storing them in one of the memory spaces available on the GPU. For every operation supported by the system, a GPU implementation must exist. Additionally, the run-time system must provide the parameters to the operations and consider the dependencies between graph nodes. The decisions about which objects should end up in which memory space and which operations should be executed when and on which processing cores can have a significant influence on the performance. Inspired by the naming in Halide [65], we call the sum of these decisions the *schedule* of the generation.

### 4.2.1 Scheduling strategies

In terms of the operator graph, the procedural generation is completed, when all objects have moved through the entire graph. The time spent on the generation can be divided into time spent on individual nodes and edges of the graph. The cost of a node corresponds to the time spent on the execution of operations. Operations are executed quickly, if (1) a sufficient number of them can be executed in parallel, (2)

no divergence occurs, *i.e.*, all GPU cores are active, and (3) operations executing on the same SIMD cores require the same low-level instructions. The cost of an edge can be seen as time needed for scheduling, including assigning cores to operations as well as loading and storing of objects. Ideally, scheduling decisions take as little time as possible and lead to divergence-free parallel execution.

**Dynamic scheduling** However, in real systems, these are opposing goals. To create divergence-free parallel execution, a scheduler needs to collect objects that are to be executed by the same operation. As new objects can be generated during any other operation, on any multiprocessor, there is no way around global, device-wide communication. Such communication is only possible via slow global GPU memory. A global sorting or grouping mechanism for objects that are to be processed by the same operation is ultimately necessary. Thus, objects must be transferred to and from global memory between the execution of operations. In the worst case, the sorting or grouping mechanisms involve device-wide synchronization. The combination of these steps can be very costly in comparison to the execution of a single operation, possibly increasing the cost of edges way beyond the cost of nodes. However, the cost of nodes will be low, as operations are executed divergence-free, and load balancing occurs over the entire device. As these scheduling decisions determine when and where operations are executed during run-time, we call them *dynamic*.

**Static scheduling** The most time-efficient way of scheduling is to avoid any decision making during run-time. The only possibility to achieve such a *static* scheduling decision on current hardware is to continue using the resources already allocated for a previous operation, fusing an operation with its successor, *i.e.* an object produced by a thread is stored within the registers allocated to this thread and the subsequent operation is executed by the same thread. If a node with multiple outgoing edges should be processed by a single thread and scheduling is completely static, the execution of all child nodes can only be serialized. This does not allow for load balancing and reduces the amount of available parallelism. At the same time, the chances of divergence rise, if the number of produced objects depends on the input object. However, there is essentially no overhead associated with static scheduling. Note that other scheduling systems, especially, if they work on entire kernels, call this kind of static scheduling *kernel fusion*.

Static and dynamic scheduling decisions represent extremes along a continuum of schedules. For example, instead of going to global memory for a dynamic scheduling decision, one can also involve local shared memory, combining information from all threads running on the same multiprocessor. This will generate a faster dynamic

scheduling decision with less overhead and fewer options in terms of load balancing and divergence avoidance. However, there is usually a pivotal point on this continuum that separates dynamic scheduling decisions, involving decisions made during runtime, from static scheduling decisions, involving only resource reuse and a predefined execution order. In the interest of brevity, we distinguish only dynamic and static scheduling decisions in the following discussions.

### 4.2.2 Graph partitioning

While scheduling decisions can either be dynamic or static, there is no need to make scheduling decisions uniformly throughout the generation process. It is rather possible to decide for each operator graph edge if it should involve a dynamic or static scheduling decision, adjusting the schedule according to a desired execution pattern. Given that the commands executed for individual operations do not change, the schedule is actually (for a given GPU) the only factor influencing the execution time of a production. Distinguishing between dynamic and static scheduling decisions, it is possible to describe a schedule as a partitioning of the operator graph. Edges within a single component of the partition involve static scheduling decisions only. Edges between different components denote dynamic scheduling decisions. We define a *schedule* $S$ as a partitioning of $V$ into non-empty subsets. We call each component $S_i \in S$ an *execution group*. Note that, by definition, $S$ itself is also an execution group. Due to the way scheduling decision can be made, certain conditions must hold for each execution group for the schedule to be valid.

**Condition 1** There can only be a single node $v_r \in S_i$, for which incoming edges have source nodes outside $S_i$. This condition makes sure that each component has a single node $v_r$ from which the execution of the component starts. All other nodes in the component can be executed in a statically defined order.

**Condition 2** There must be a path from the starting node $v_r$ to each other node $v_n \in S_i$ of the execution group that only contains nodes of $S_i$. This condition guarantees that a static schedule involving all nodes in $S_i$ can be defined. All edges that are used in the above definition shall involve static scheduling decision and shall be called *static edges*.

**Condition 3** An execution group is not allowed to contain cycles, besides cycles that include $v_r$. This conditions prohibits recursions within a component, which might lead to unpredictable time and resource requirements within statically scheduled nodes. If a cycle is formed with $v_r$, the incoming edge at $v_r$ is made dynamic, and the cycle can be supported via dynamic scheduling.

**Condition 4** Dependency edges are only allowed between different execution groups and not within an execution group. This condition is necessary, as context sensitivity is usually only supported between dynamic scheduling decision, *i.e.*, the scheduler must be able to check if all sources of the dependency have been executed, before the dependent node can be executed.

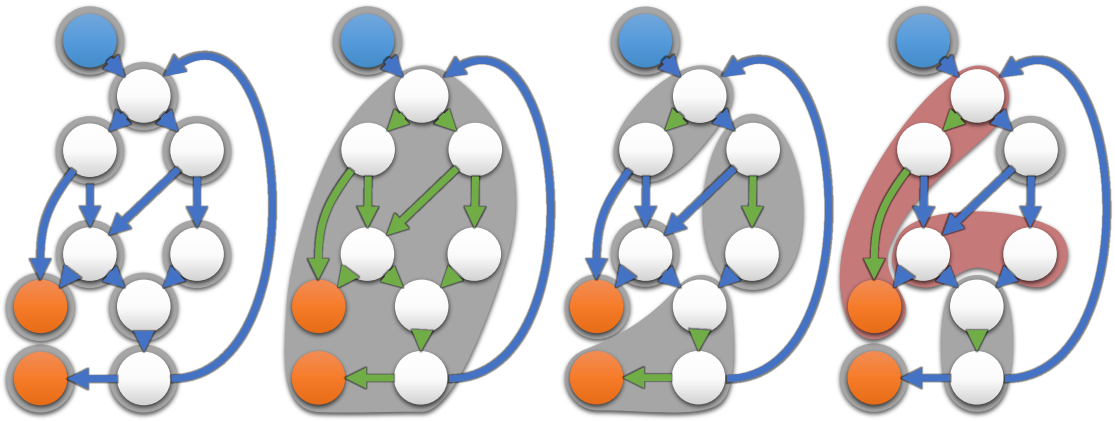



Figure 4.5: Simplified Menger Sponge operator graph partitions. Blue edges are dynamic, green edges static, execution groups are outlined in gray. (a) Turning every node into an execution group, yields completely dynamic scheduling. (b) A large execution group with mostly static edges. (c) Rules turned into execution groups. (d) Invalid partitioning (red) having multiple nodes with dynamic incoming edges or disconnected nodes.

**Examples** We revisit the Menger sponge, as shown in Figure 4.5. If all edges are made dynamic (a), a large number of small execution groups is generated, possibly leading to large scheduling overhead. As the graph contains a single cycle, nearly the entire graph can be turned into a single execution group (b). The edge creating the cycle becomes a dynamic edge. The first node in a large execution group is the only one with incoming edges from outside of the execution group. A schedule like this only draws parallelism from the recursion and the source node. A mixture of dynamic and static edges creates execution groups of different sizes (c). A setup like this might be a good compromise between dynamic and static scheduling. Not all possible partitions are valid; unconnected nodes or multiple nodes with incoming edges outside of the execution group yield an invalid schedule (d).

Operator graph scheduling allows modeling previous work as different partitioning schemes, independent of the low-level details of the implementation: Sequential rewriting and shape grammar evaluation algorithms, such as the approaches by Lacz

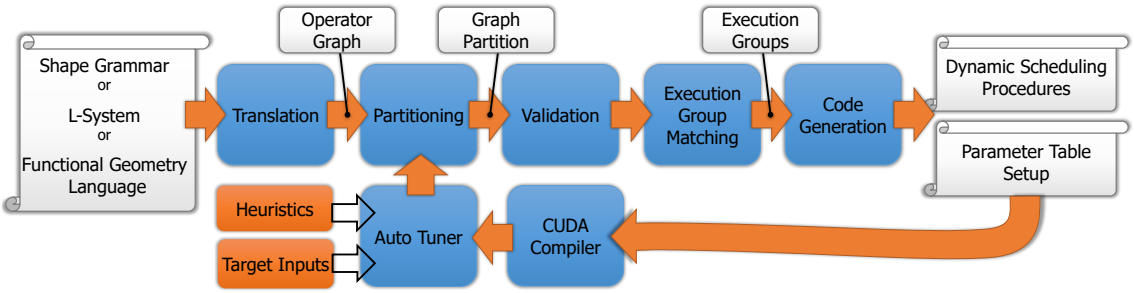



Figure 4.6: Using the operator graph as central concept, our compile pipeline consists of seven major steps: A procedural generation written in one of the supported languages is translated into an operator graph; after selecting one partition, it is validated, and execution groups are formed, while taking operator homomorphism into account. Finally, code for a dynamic scheduler is generated alongside the parameter table. After compilation to GPU code, an auto-tuner records statistics for possible input parameters and steers the selection of different partitions.

et al. [34] and Lipp et al. [39], yield execution groups $S_i \in S$ of single nodes, $|S_i| = 1$. Thus, they make scheduling decisions after every single operation. While they get good load balancing and a high degree of available parallelism, their scheduling overhead is large. GPU shape grammars [46] put the entire graph into a single execution group $S = V$. Thus, they can only draw parallelism from the axioms and face problems with divergence, when generating different buildings. PGA [72] partitions the graph according to the rules written by the designers. Thus, their performance heavily depends on the way the rule sets have been written and can suffer the same problems as the other approaches.

### 4.2.3 Execution group matching

There is another factor influencing performance that can be described using the operator graph. Dynamic scheduling combines objects that are to be executed by the same execution group to generate divergence-free parallel execution. If the number of objects for one execution group is too low, the execution will suffer under-utilization or divergence. To mitigate this issue, we propose execution group matching in combination with dynamically loading parameters.

To describe execution group matching, we define *operator homomorphism*, which applies to operator graphs that are compatible in terms of structure and operations and, thus, can be described by the same parameterized execution group: Let $G_1 = (V_1, E_1, D_1)$ and $G_2 = (V_2, E_2, D_2)$ be two different operator graphs with $e_1 \in E_1$, $e_2 \in E_2$ and $d_1 \in D_1$, $d_2 \in D_2$. These operator graphs are *operator homomorphic*,

iff there is a bijective homomorphism $f_h : V_1 \to V_2$, which fulfills the following conditions:

$$\forall e_1 \, \exists e_2 : f_h(s(e_1)) = s(e_2) \wedge f_h(d(e_1)) = d(e_2) \tag{4.1}$$
$$\exists e_1 \, \forall e_2 : f_h(s(e_1)) = s(e_2) \wedge f_h(d(e_1)) = d(e_2) \tag{4.2}$$
$$\forall d_1 \, \exists d_2 : f_h(s(d_1)) = s(d_2) \wedge f_h(d(d_1)) = d(d_2) \tag{4.3}$$
$$\exists d_1 \, \forall d_2 : f_h(s(d_1)) = s(d_2) \wedge f_h(d(d_1)) = d(d_2) \tag{4.4}$$
$$f_h\left((r_1, p_1)\right) = (r_2, p_2) \longrightarrow r_1 = r_2, \tag{4.5}$$

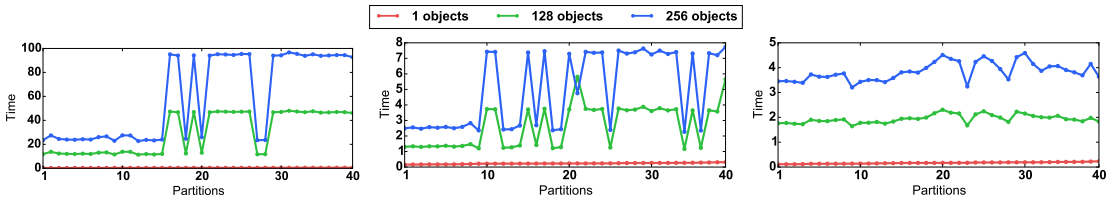
*i.e.*, if there are matching operations in all nodes in both graphs and edges that connect those nodes in the same way.

If the graphs of two execution groups are operator homomorphic, they can be described by a single parameterized piece of code, constructed from the operations used in either graph. This means that all possible ways through two operator homomorphic graphs can be described by the same code. This concept is not only applicable to two execution groups, but can be extended to multiple execution groups, combining all of them. The implication is that more objects can be combined for this parameterized execution group and thus executed more efficiently. However, as the parameters of the individual operations might differ, the scheduling system must provide the parameters dynamically to the executing objects.

In addition to easing the process of finding a sufficient number of objects for efficient execution during dynamic scheduling, a homomorphic execution group is represented using a single GPU function instead of multiple. This might have an additional positive effect on the cost of dynamic scheduling. Depending on the implementation of dynamic scheduling mechanism, fewer execution groups might reduce the number of grouping structures or queues and thus reduce the time spend on searching through this structure. Therefore, execution group matching can potentially reduce the overhead of dynamic scheduling.

## 4.3 Operator graph equivalence to conventional representations

Recall that a variety of methods and systems for procedural generation can be described using an operator graph. In this section, we provide additional detail on how the most prominent methods for procedural generation can be cast into operator graph form.




(a) GTX 630 (b) GTX 780 Ti (c) GTX 980

Figure 4.7: Performance results in ms for all 40 possible schedules for the Menger Sponge test case with different object counts (sorted by performance on the GTX 630). Even for such a small rule set, the performance difference between schedules can vary from a factor of two to five. Note how the relative performance of the schedules changes between object counts and GPU.

### 4.3.1 L-systems extensions

The operator graph can also be used to describe extensions for L-systems. Stochastic L-systems allow to specify multiple production rules for a single symbol, each being chosen with a certain probability. This behavior can be integrated into the previously described system by adding an operation that chooses one of its outgoing edges at random. Context-sensitive L-systems adjust the production to the characters before and after the input. This behavior can be modeled with dependency edges, making sure that all symbols for a certain iteration are processed, before executing the rules for the next iteration. Another extension to L-systems adds parameters to characters. Using our definitions, this can be achieved by extending the object type to $O = S \times \mathbb{N}^2 \times \Re^n$, with $n$ being the number of parameters that should be associated with each character.

### 4.3.2 Shape Grammars

From a procedural generation point of view, shape grammars, like CGA shape [49], are similar to L-systems. A designer sets up production rules, which are associated with symbols to define the sequence of operations to be applied. However, the underlying objects are shapes ($O$), and the production rules themselves can be sequences (or trees) of operations. These operations can usually be chosen from a predefined set of parameterized operations ($R$). As it has been shown before that rule sets of simple, context-free shape grammars can be described by a direct acyclic graphs [58], it is not surprising that our operator graph can be used for shape grammars. However, our operator graph can model more complex shape grammar operations, like context sensitivity or recursive productions.

For example, a rule set written in CGA shape to create the Menger Sponge (see Figures 4.2 and 4.3) could look as follows:

```
1: A -> Split(X){1r: B,
2:     1r: Split(Y){1r: C, 1r: Discard, 1r: C},
3:     1r: B }
4: B -> Split(Y){1r: D, 1r: C, 1r: D}
5: C -> Split(Z){1r: E, 1r: Discard, 1r: E}
6: D -> Split(Z){1r: E, 1r: E, 1r : E}
7: E -> IncRec(){IfRecGreater(4){Term, A}}
```

This rule set defines five rules, using the *Split*, *Discard*, *IncRec*, and *IfRecGreater* operations. $1r$ represents a relative size parameter. Translating a rule set into an operator graph is straight forward. Linking rules with symbols corresponds to setting up edges between nodes. Also, the nesting of operators within a rule (line 1 and 7 in the example) translates to edges. The operations, including the parameters, are captured by nodes, translating the rule set above into the operator graph shown in Figure 4.3. Note that the operator graph is to a certain degree independent of the way a designer chooses to group operators to rules. If the designer split the nested rules in line 1 and 7 into multiple rules, each containing a single operation, the resulting operator graph would still be the same.

Shape grammars often use random values and randomized rule selection to introduce stochastic variation into the generation process. Random parameters are simply added as parameters in the operator graph and can be defined by any random variable distribution. Probabilistic rule selection can be added as another node. Finally, context-sensitivity can either be set up directly between nodes (as described before), or between entire subsets of nodes. CGA shape assigns priorities to rules, which translates into execution phases, *i.e.*, executing all rules with highest priority before rules of the next lower priority. To model this behavior with dependency edges, we add dependencies between all nodes of a higher priority to the next lower priority, forcing all nodes of a certain priority level to be executed before nodes of the next phase.

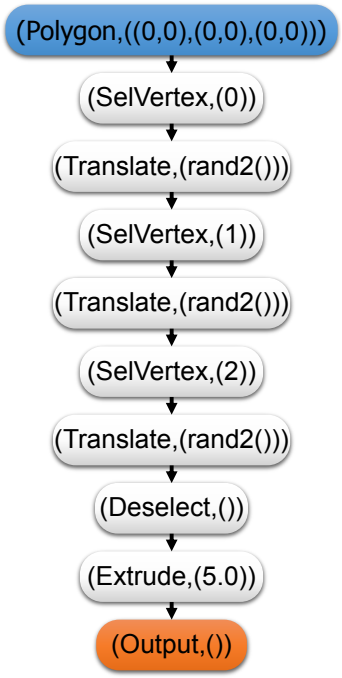
### 4.3.3 Stack-based Generation

Stack-based generation languages, like GML [23], also work on simple objects and offer a user a set of operations that can be applied to objects. As found by Havemann [23], the similarity between GML and a generalized data flow network is striking. They can actually describe the same set of problems. By "flattening" out the stack

(constructing the inputs and outputs of operations from the stack), it is possible to generate a graph of operations. However, there is a distinct difference to our operator graph. A single operation in a stack-based language can pop any number of objects from the stack. In the operator graph, only a single object (and a set of parameters) can be consumed by an operation. We do not allow for a join node that would combine multiple input objects. This limitation keeps objects independent of each other and allows for efficient parallel execution.

```
01: (0,0)
02: vertex
03: (0,0)
04: vertex
05: (0,0)
06: rand2()
07: translate
08: rand2()
09: translate
10: rand2()
11: translate
12: 3
13: polygon
14: 5.0
15: extrude
```

(a) Stack-based Program




(b) Operator Graph

Figure 4.8: A program written in a stack-based modeling language can be flattened to a data flow graph. As operations used in the operator graph only consume a single object at the time, operations which would require multiple inputs are replaced by a single object and selectors for the sub objects. Note that such a strategy is hardly ever needed in practice.

However, most operations in this type of procedural generation target a single input object. Additional inputs usually correspond to parameters controlling the operation itself, *e.g.*, the length of an extrude or the direction of a translation. Among the few exceptions that take multiple input objects are operations that combine low-level objects. For example, a variable number of vertices may be combined to a polygon. Each of those vertices can come from a chain of operations. While this is usually not the case, the operator graph can support such more complex setups, too.

For example, instead of modeling the chains of operations that produce three vertices as input to a polygon generation, we can start with a polygon of three vertices and alter one after the other, as shown in Figure 4.8. In essence, we serialize operations from previously parallel paths. In this way, even operations which pop multiple complex objects from the stack can be represented by an operator graph.

## 4.4 Compiler Pipeline

To complete our approach, we describe a state-of-the-art procedural generation runtime system and an auto-tuner for the selection of schedules. These components close the loop of our compile pipeline, see Figure 4.6. As dynamic scheduler, we use the task scheduling framework *Whippletree* [73]. Whippletree works around the definition of procedures and tasks. Procedures are function-like entities that take tasks as input. Tasks are collected in queues in global GPU memory. When there are enough tasks available for parallel execution on a multiprocessor, Whippletree executes them together, increasing the chances for a divergence-free execution. In our terminology, tasks correspond to objects moving through the operator graph, and procedures are operations or execution groups. Whippletree makes dynamic scheduling decisions as to when and on which multiprocessor these execution groups should be executed. Note that Whippletree has been used for shape grammar scheduling before [72]. However, we do not use Whippletree's shape grammar implementation, but rather generate a schedule directly from an operator graph and our own operation definitions.

### 4.4.1 Procedural generation system

We implemented a limited number of operations, which are similar to the shape grammar operations available in CGA shape, as well as operations for L-systems and functional languages for procedural generation. To use Whippletree for dynamic scheduling, we define a Whippletree procedure for every execution group in the operator graph. The procedure is built from operations used in the execution group, together with the parameters specified. These parameters can either be predefined values or drawn from random distributions. Within an execution group, we serialize the execution of all contributing operations, implementing static scheduling decision. Serializing the graph in a depth-first manner, we make sure that the number of temporary objects that need to be kept in registers is low. Whenever a dynamic edge is reached, we hand the object over to the Whippletree scheduler, which transfers it into its own grouping mechanism and load-balances its execution across the entire

GPU.

We require a way to supply dynamic parameters to operator-homomorphic execution groups. For this purpose, we use a parameter table stored in GPU memory. This table is used to look up the parameters of the involved operations based on a unique identifier that we store alongside each object. During execution, each thread looks up the parameters associated with its object. Using the texture cache for parameter loads reduces the overhead of these additional memory transactions. As a subset of the parameters might be identical for all operator homomorphic execution groups, we use the look-up table only for those parameters which actually differ. The remaining parameters are statically included during compilation.

Our implementation also supports context sensitivity, *i.e.*, dependency edges. Dependencies always involve two operations. The first operation (source) adds an object into a spatial data structure and adds a customizable id to that object. The second operation (destination) allows to query an object against the objects stored in the spatial data structure. If there is an overlap between the query and the objects of a given id in the data structure, a different outgoing edge is chosen than if there is no overlap. Scheduling needs to make sure that all source operations are completed, before the destination nodes are executed. Whippletree does not allow to set up these dependencies directly. However, we can separate the execution in multiple phases with global synchronization barriers in between by executing a sequence of Whippletree programs, one for each phase. Whenever there are dependencies in an operator graph, we make sure that source and destination nodes end up in different phases, while keeping the number of phases minimal.

### 4.4.2 Scheduling optimizer

The fastest schedule for a given operator graph might depend on the GPU architecture and the number of objects generated by the source node. Depending on the number of cores, a GPU requires more or less parallel workload to work efficiently. A larger number of initial objects provides more parallelism, and, thus, less dynamic scheduling is necessary to create enough parallel workload. When trying to find the best schedule, these factors must be considered.

The number of different graph partitions (and different schedules) for an operator graph with $|E|$ edges is $2^{|E|}$ (every edge can either be dynamic or static). While a large number of these schedules might achieve similar performance, the difference between a good schedule and a bad schedule can—according to our experiments—be up to two orders of magnitude for large operator graphs. Even for small graphs, the

best performing schedule varies between object counts and GPU type, as shown in the example in Figure 4.7. This large variance shows that, for modern a GPU, it is difficult to predict what makes a schedule good or bad. Thus, it is essential to provide means to find the best schedule for any given operator graph, object count and GPU.

With the goal of finding the most efficient procedural generation system, we search for the schedule that minimizes execution time. To this aim, we implemented an auto-tuner that searches for the best schedule for a given operator graph. It is intended to compute good schedules for use cases such as video games, movie production or inverse procedural modeling. As a baseline, it uses an exhaustive search algorithm, generating and evaluating all valid schedules for a given set of source objects.

Our compiler pipeline starts with a rule set written in a syntax similar to CGA shape or an L-system and internally translates it into an operator graph. To perform this translation, the compiler requires a definition of supported object types, operations, and input-output relations (cf. $O$, $R$, and $IO$ in section 4.1). We use C++ class specifications for object types and C++ source code for the operations and input-output relations. Internally, our compiler represents all possible partitions as a bit sequence (one bit per edge). After a partition is selected, it is checked for validity according to the conditions given in section 4.2.2. If it is valid, we search for operator homomorphic execution groups and identify common parameters. Finally, the compiler generates Whippletree procedure code for all execution groups, including the CUDA/C++ code for the involved operations. It also generates the parameter table for the procedures and inserts the corresponding load instructions into the CUDA/C++ code. The generated code is compiled, and the auto-tuner evaluates its performance for a given set of target input objects.

Even after restricting the search space to valid partitions, it is still not practical to perform an exhaustive search through the remaining number of possible schedules for larger graphs. Thus, we introduce a set of heuristics based on parallel programming principles translated to choices in the operator graph. These heuristics work by setting edges as static or dynamic, according to certain local characteristics of the operator graph. Naturally, fixing an edge (to either static or dynamic) reduces the search space in half. Although heuristics significantly reduce the search space, there is no guarantee that the best schedule is found. However, we have collected strong evidence that the presented heuristics work well for a wide variety of cases.

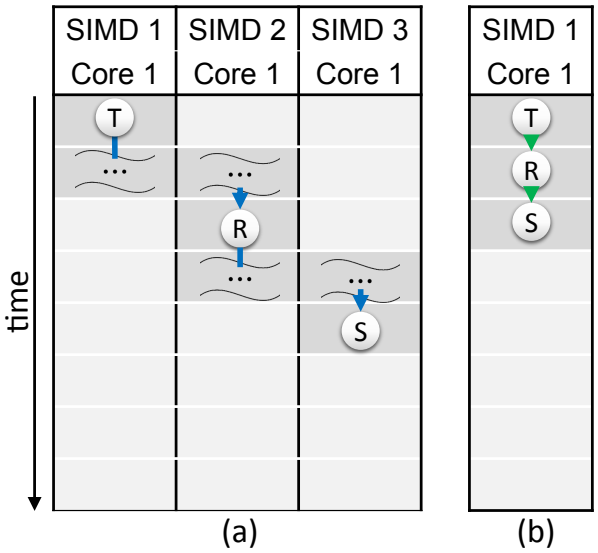



Figure 4.9: H1 (a) → (b) removes dynamic scheduling if there is only a single object generated.

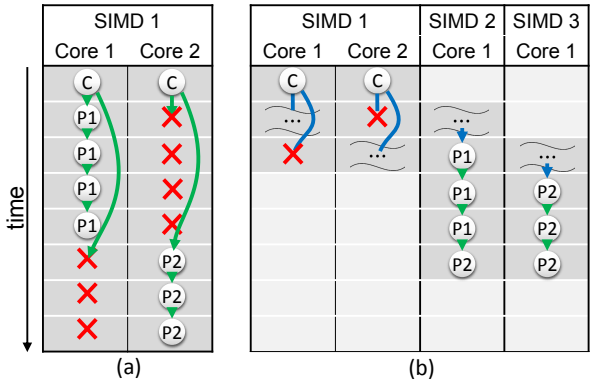



Figure 4.10: H2 (a) → (b) removes divergence (red crosses) by adding dynamic scheduling under conditional nodes.

### 4.4.3 Sequence fusion heuristic

Nodes with a single outgoing edge of multiplicity $m = 1$ do not introduce any parallelism. If no parallelism is introduced, there is no gain for load balancing, and a dynamic edge would only increase scheduling overhead. Thus, the sequence fusion heuristic H1 sets all edges in $E_{h,1}$ to static:

$$E_{h,1} = \{e | m(e) = 1 \wedge |out(src(e))| = 1\}.$$

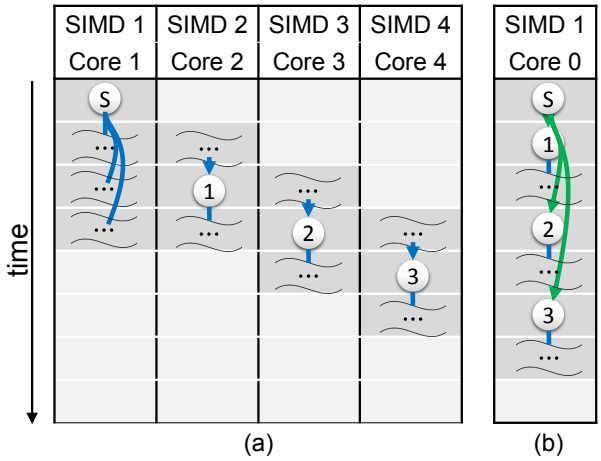



Figure 4.11: H3 (a) → (b) makes sure small execution groups are not split apart and thus reduces dynamic scheduling overhead.

For example, consider the sequence of a *Translate*, *Rotate* and *Scale* operation, as shown in Figure 4.9. For simplicity, assume that all operations as well as a dynamic scheduling decisions take equally long. The operations carried out on different SIMD units and cores are represented as cells of the table. Light cells indicate available cores, and dark cells indicate that the resource is used. Using static edges between those operations will result in a single thread executing all three operations, with virtually no cost for scheduling. Dynamic edges, in this case, might result in the operations to be executed with different threads. However, no more than a single thread will be active with those operations at any point in time. Additionally, the overhead for two dynamic scheduling decisions increases the overall execution time. Resource usage (darker cells) is reduced from seven to three, and the execution time is reduced from five to three by applying the heuristic. Intuitively, static edges should lead to a better performance in this case.

Note that applying this heuristic might link two dependent operations to each other with a sequence of static edges. For example, imagine a dependency edge between the *Translate* and *Scale* operation in the previous example. One of the static edges needs to be turned into a dynamic edge to yield a valid schedule. In our implementation, this is taken care of by the optimizer, which validates each application of heuristics. It would reject the static edge, when the heuristic is applied to the last edge in the sequence, as it would connect the dependencies to each other.

### 4.4.4 Divergence avoidance heuristic

Edges with multiplicity $m = /$ have a high change of introducing divergence. Consider an *If* operation or an operation that chooses one of the outgoing edges at random. For those operations, different objects have a high chance of choosing different edges for execution. If these edges are made static, thread divergence will occur. Thus, the divergence avoidance heuristic H2 sets all edges in $E_{h,2}$ to dynamic:

$$E_{h,2} = \{e | m(e) = /\}.$$

For example, consider a conditional operation executed on two cores of the same SIMD unit, as shown in Figure 4.10. Each conditional branch is followed by a sequence of operations. If static scheduling is used, and objects choose different edges, half of the cores will not run code for the entire execution. Using dynamic edges, however, will only introduce a small amount of divergence, when starting the dynamic scheduling. The execution of the following operations can then be carried out in parallel on different SIMD units, avoiding divergence and speeding up the execution (eight time units vs six time units). Also, the overall resource usage (darker cells) is reduced from 16 to 15. Intuitively, dynamic edges will allow to reduce divergence, introduce parallelism and speed up the execution in this case.

### 4.4.5 Execution group size heuristic

In contrast to edges that might or might not be taken, there are those nodes that have edges with a fixed multiplicity, *i.e.*, there will always be a fixed number of objects created by an operation. Examples include *Subdivide* or *ComponentSplit*. If the outgoing edges of such a node are set to static, combining a node with its children into the same execution group will prevent divergence, but potential parallelism will be lost. At first sight, it may appear that these outgoing edges are good candidates for dynamic edges to increase parallelism. However, one has to consider the overhead of dynamic scheduling. This is particularly true, when the edges following the successors are dynamic, too, creating sequences of dynamic scheduling decisions. In this case, the scheduling overhead will outweigh the gains of parallel execution. Thus, the execution group size heuristic H3 enforces a minimum execution group size by greedily setting edges with a constant multiplicity to static until a certain execution group size $t$ is reached.

$$\forall S_i \subseteq S : |S_i| \geq t \tag{4.6}$$

For example, consider a *Subdivide* operation creating three objects, each followed by a single node and a dynamic scheduling operation, as shown in Figure 4.11. If the edges are set to dynamic, three dynamic scheduling operations are carried out, and the following nodes are distributed among different cores. In this way, parallelism is generated, but with a high amount of scheduling overhead. However, if the edges are set to static, all nodes are executed on the same core, and the scheduling overhead is greatly reduced. While the overall execution is taking slightly longer (one time unit), the consumed resources (darker cells) are nearly halved from thirteen to seven. As long as there is a sufficient amount of parallelism available, the reduction in resource usage will lead to a faster generation process. In our experiments, we used a minimum execution group size $t$ of 5.

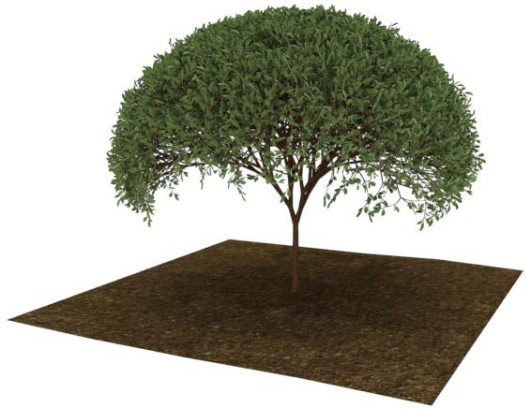
## 4.5 Results

To evaluate our approach, we used test cases from shape grammars, L-systems, and Monte Carlo procedural modeling approaches as shown in Figure 4.12 and Figure 4.1. The test cases include a variety of sizes and include results for different numbers of initial object counts. As test system, we used an Intel i7 4820K with 16GB RAM and an NVIDIA GTX 780Ti.

### 4.5.1 Evaluation of the heuristics

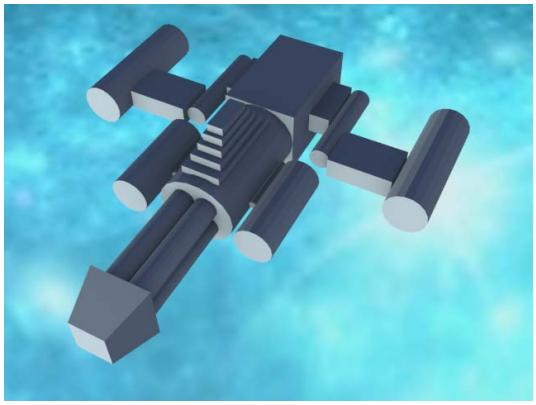
To evaluate the heuristics on test data sets of a non-trivial size, we use the following approach: First, we identify all edges in a given operator graph for which a heuristic does not apply and randomly set each of those edges to be either dynamic or static. Second, we apply the heuristic to the other edges and run the schedule for a given set of input objects, recording their performance. Third, we invert those edges to the opposite of the heuristic and run the schedule again on the same input set. We repeat the entire process, until we arrive at a predefined number of samples. For the pairs of samples, we run a paired Student's t-test to check if there is a difference between the performance of the schedules with activated heuristic and inverted heuristic. To determine the number of samples for achieving a 5% error margin and a confidence interval of 95%, we created 384 pairs, following the sample size guidelines [33].

In each of the test cases, we make sure that the pattern occurs on which the heuristic is based. We used *Suburban House* for H1, as it is rather small and offers multiple edges that match the heuristic, *Commercial* for H2, as it chooses randomly from different window styles, and *Balcony* for H3, as it has many nodes with only few outgoing edges. The t-test results for all three evaluations are shown in Table 4.1.



(a) 3D Tree 12,2

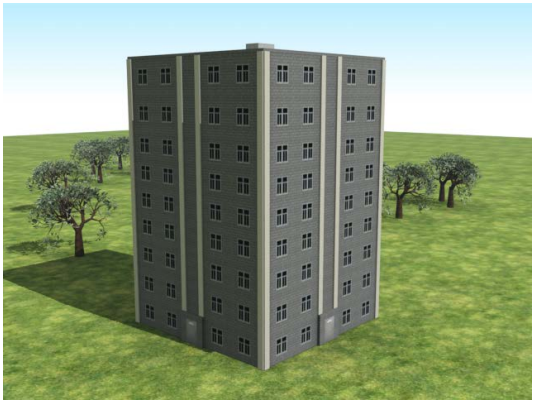
(b) MC Skyscrapers



(c) Spaceship

(d) Suburban House

(e) Balcony



(f) Commercial

Figure 4.12: The evaluation test cases include L-systems (a), Monte Carlo procedural modeling (b and c), and shape grammars (d-f).

| Test | Obj | $t$ | $p$ | $H$ | $\neg H$ | perc |
|---|---|---|---|---|---|---|
| H1 15/76 edges | 1 | -8.36 | ¡.001 | 0.58 (0.65) | 0.61 (0.68) | .81 |
| | 64 | -51.38 | ¡.001 | 1.13 (0.72) | 1.36 (0.72) | .99 |
| | 128 | -74.36 | ¡.001 | 1.73 (0.70) | 2.16 (0.69) | .99 |
| | 256 | -94.54 | ¡.001 | 2.95 (0.73) | 3.81 (0.72) | 1.00 |
| H2 6/150 edges | 1 | -18.20 | ¡.001 | 0.53 (0.09) | 0.58 (0.08) | .87 |
| | 64 | -30.76 | ¡.001 | 0.84 (0.16) | 0.94 (0.15) | .94 |
| | 128 | -34.85 | ¡.001 | 1.16 (0.29) | 1.29 (0.28) | .95 |
| | 256 | -35.86 | ¡.001 | 1.79 (0.56) | 2.01 (0.53) | .96 |
| H3 43/104 edges | 1 | -2.87 | .004 | 0.51 (0.16) | 0.54 (0.04) | .89 |
| | 64 | -27.87 | ¡.001 | 0.65 (0.17) | 0.90 (0.07) | .98 |
| | 128 | -55.96 | ¡.001 | 0.78 (0.16) | 1.33 (0.13) | 1.00 |
| | 256 | -94.96 | ¡.001 | 1.09 (0.16) | 2.19 (0.25) | 1.00 |

Table 4.1: The heuristics affected 15, 6 and 43 edges out of the 76, 150 and 104 edges in the respective operator graphs. All three heuristics had statistically significant influences on the performance ($p$ value and $t$ for the paired Student's t-test). $H$ is the mean generation time (with standard deviation) in ms with heuristic on, while $\neg H$ shows the same results with heuristics off. For small object counts (Obj), up to 89% (perc) of pairs benefited from the heuristic. For larger object counts, the heuristics work even better.

As can be seen, all heuristics had a statistically significant influence on performance for all axiom counts. Altering 19.7%, 4%, and 41% of edges boosted performance up to 30%, 12%, 100%, on average, for H1, H2, and H3, respectively, indicating that the chosen edges actually have a large impact on performance. The heuristics seem to work especially well for larger object counts. Overall, we argue that all three heuristics seem to be a good starting point for optimization, reducing the search space significantly.

### 4.5.2 Runtime performance

To relate our approach to previous work, we ran the *Tree 4,3* test case from GPU Shape Grammars [46], the *Tree 8,3*, *Overview* and *Skyscrapers* test cases from PGA [72] as well as the *MC Spaceship* and *MC Skyscrapers* from Stochastically-Ordered Sequential Monte Carlo [67]. We include their performance numbers for CPU and GPU approaches (adjusted for hardware differences). Additionally, we compare how the scheduling strategies followed by previous work introduced in our scheduler perform compared to the best schedule found by our auto-tuner. Rewriting

| Scene | Edges | Obj | Term | CPU | GPU | SGL | RW | DGN | $OPT_S$ | | $OPT_{S,M}$ | | $OPT_H$ | | $OPT_{H,M}$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Overview[1] | 9 | 38000 | 0.91M | 996 | 48.24 | **1.56** | 5.24 | 1.75 | 1.62 | 1m | **1.56** | 1m | 1.76 | 1m | 1.74 | 1m |
| Simple House | 11 | 4096 | 1.48M | | | 3.23 | 6.36 | 3.59 | 2.86 | 3m | **2.81** | 3m | 2.93 | 1m | 2.91 | 1m |
| Menger Sponge 3 | 17 | 256 | 2.05M | | | 7.28 | 2.87 | 2.87 | 2.28 | 10m | **2.27** | 10m | 2.30 | 2m | 2.31 | 2m |
| 3D Tree 12,2 | 35 | 64 | 0.65M | | | 1.29 | 3.89 | 1.41 | 3.27 | 5h | 2.99 | 5h | **1.26** | 2h | 1.28 | 2h |
| 3D Tree 4,3[2] | 35 | 1 | 283 | | 10.06 | 0.10 | 0.50 | 0.14 | 0.21 | 4h | 0.21 | 4h | 0.09 | 57m | **0.09** | 50m |
| 3D Tree 8,3[1] | 35 | 1 | 23K | | 3.71 | 0.20 | 1.34 | 0.36 | 0.32 | 4h | 0.35 | 4h | 0.19 | 1h | **0.19** | 1h |
| MC Spaceship[3] | 51 | 16384 | 0.40M | 4996 | | **1.48** | 5.47 | 2.77 | 1.96 | 6h | 1.91 | 5h | 1.55 | 4h | **1.48** | 4h |
| Skyscrapers[1] | 61 | 529 | 1.84M | 516 | 22.51 | 12.10 | 13.99 | 6.81 | **4.15** | 7h | 4.23 | 7h | 4.64 | 7h | 4.52 | 7h |
| MC Skyscrapers[3] | 63 | 512 | 1.02M | 818 | | 65.6 | 6.11 | 5.61 | **4.00** | 7h | 4.11 | 7h | 4.80 | 7h | 4.57 | 7h |
| Suburban House | 76 | 256 | 0.60M | | | 14.54 | 5.75 | 4.08 | 2.74 | 11h | 2.45 | 10h | 1.95 | 8h | **1.87** | 8h |
| Balcony | 104 | 256 | 0.18M | | | 4.99 | 3.57 | 2.16 | 1.18 | 7h | 1.16 | 7h | **0.75** | 9h | 0.76 | 8h |
| Commercial | 150 | 256 | 0.14M | | | 7.58 | 3.20 | 2.68 | 1.06 | 7h | 1.06 | 7h | 1.04 | 6h | **0.96** | 6h |

[1] from [72], [2] from [46], [3] from [67]

Table 4.2: Evaluation results in ms for test cases with different number of edges in the operator graph (Edges), objects counts (Obj), and number of terminals (Term) for various partitioning schemes. Previous implementations are included in terms of CPU and GPU generation time (performance extrapolated from original work using GPU floating-point operations per second (FLOPS) ratio). While a single execution group (SGL) performs well for small graphs, it is not well suited for large graphs. The rewriting approach (RW) behaves in the opposite way. The approach using the rules as execution groups (DGN) forms a trade-off between the two. Our optimizer using a random search ($OPT_S$), heuristic search ($OPT_H$), and with execution group matching ($OPT_{S,M}$, $OPT_{H,M}$) always find a better or equally good schedule. Time used for auto-tuning is additionally given; for the first three test cases all possible schedules can be tested. For the following test cases we sampled a maximum of 1000 schedules.

approaches ($RW$), like the ones by Lacz et al. [34] and Lipp et al. [39], correspond to execution groups of size 1 and all parameters being stored in the parameter table. GPU Shape Grammars [46] are represented by a schedule of a single execution group with all parameters being static ($SGL$). To represent PGA [72], we used execution groups according to the rules specified in the shape grammar rule set and again provide all parameters as statics. For unbiased evaluation, we let an external expert on shape grammars write those rule sets for us ($DGN$). We ran our optimizer in four different setups: random search ($OPT_S$), search with heuristics ($OPT_H$), and both variants with execution group matching ($OPT_{\cdot,M}$).

The performance results are shown in Table 4.2. SGL performs well for small operator graphs and large object counts, as there is a sufficient parallelism available, and scheduling overhead is reduced to a minimum. Additionally, there is a low chance of divergence for such small operator graphs. RW introduces the most parallelism, but also has the highest scheduling overhead. Thus, it performs better, when there are few initial objects for which parallelism must be generated quickly. DGN leads to more balance between scheduling overhead and the ability to generate parallelism. It works well for test cases which are above a certain size, outperforming both other approaches. In case of smaller test cases, DGN is, however, outperformed by SGL. Looking at all three approaches, we can see that there is no single strategy that always performs best. For the smaller test cases, the auto-tuner can search the entire space. Thus, $\text{OPT}_S$ slightly outperforms $\text{OPT}_H$ as it tests every single schedule. For the larger test cases, a fully random search is less likely to pick good schedules; thus, the heuristics-guided search most often achieves better results in less time. Also, execution group matching usually achieves slightly better results, while, at the same time it reduces compile time. Overall, OPT always picks the best option, boosting performance by up to 14.4× compared to SGL, 7.1× compared to RW, and 2.8× compared to DGN. On average, OPT is better by 3.9×, 3.6×, and 1.8×, respectively. The larger the operator graph gets, the larger the difference becomes. This points towards the fact that efficient scheduling for complex problems cannot easily be done by hand. Also, as procedural worlds are growing in size, the gains of an auto scheduler grows.

As the operator graph can be applied to different procedural approaches, we used it for L-system generation, shape grammars and Monte Carlo procedural modeling. GPU Shape Grammar's 3D Tree 4,3 is constructed by their approach in 40 ms. Adjusting for GPU differences (time multiplied by peak FLOPS ratio) results in about 10 ms on our GPU. The same test case is completed by $\text{OPT}_H, M$ in 0.09 ms. The comparisons to PGA (Overview, 3D Tree 8,3 and Skyscrapers) show that OPT

finds schedules that are 5 to 30 times faster than their GPU approach (adjusted for GPU differences) and up to 100 times faster than their CPU approach. Bringing Monte Carlo procedural modeling to the GPU, an optimized schedule can speed up the generation process multiple hundred times compared to the CPU implementation by Ritchie et al. Considering all results, it is safe to assume that our compiler finds efficient schedules. The comparison to previous work suggests that the combination of our compiler with an efficient dynamic scheduler yields the currently fastest procedural generation system for the GPU.

However, finding the best schedule for a given operator graph comes with a cost. For every single schedule tested, we have to generate the partition from the operator graph, generate source code, compile it, and run a representative set of procedural generations. The partition and source code generation takes less than a second, the compilation takes between 20 seconds and 2 minutes, depending on the operator graph size. Thus, the optimization for large test cases can take multiple hours. As the same schedule can be used for different initial objects, the target scenario for our approach is optimizing a generation that is used heavily in a production system for different inputs. For example, we used our approach to optimize the generation for an entire randomly generated city as seen in Figure 4.1.

## 4.6 Summary

While previous approaches to procedural generation on the GPU employed specialized data structures and techniques, we looked at the problem from a high-level perspective and introduced the concept of the operator graph to handle a variety of procedural generation methods. The operator graph concisely describes the generation process for any input object. Given the operator graph, we draw a connection between a partition of this graph and how a system can schedule the generation on the GPU. An optimizer can search the space of all partitions to find the schedule that performs the generation within the shortest time. While an exhaustive search of this exponential space is only possible for small graphs, we proposed three heuristics, which all increase search performance. Finally, we showed that our tool chain can significantly speed up the generation of a variety of common procedural generation systems. Our optimizer is able to increase performance over hand crafted solutions by up to fourteen times only by changing the schedule.

# 5 Vector Graphics Rasterization

## Contents



Vector graphics precede raster graphics as a representation of digital content, yet, remain relevant today, since a resolution-independent representation allows artifact-free display on everything from a tiny smartwatch to a huge wall-size display. Consequently, vector graphics are ubiquitous in all kinds of data visualization, including font rendering, user interfaces, web pages, diagrams, charts, maps, games, and artistic illustrations.

However, vector graphics representations have not radically departed from the seminal work of Warnock and Wyatt [86]. Vector graphics are typically defined as a collection of paths, where each path is defined by a number of curves. Curves are commonly defined as straight lines, quadratic or cubic Bézier curves, or circular segments. A closed path separates an interior and exterior; the interior and the path itself can be filled using a variety of styles and patterns.

Unfortunately, efficient rendering of vector graphics at high resolutions still forms a challenging task for computer graphics. Rendering on the CPU does not scale well to high resolutions and super-sampling. Consequently, parallel vector graphics rendering has been actively researched over the last decades [40, 29, 19, 5, 36]. However, there is still no parallel approach for vector graphics rendering which comes close to the elegance and efficiency of triangle rasterization.

Recent approaches typically use two steps: *stencil, then cover* [40, 29, 36]. A first step determines which (sub-)pixels are inside a patch. A second step evaluates the actual shading of the marked pixels. The stencil generation is the costly step of the two, either involving a large number of overlapping triangles to modify the stencil [40, 29] or scanline-curve intersections to generate bit masks [36].

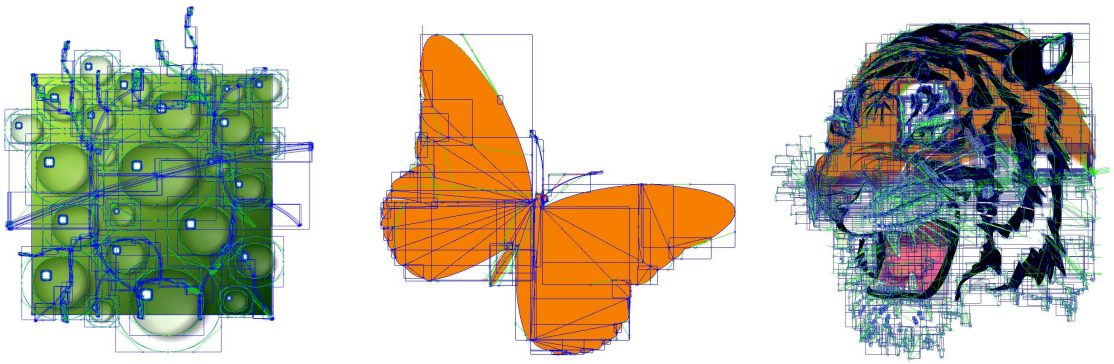
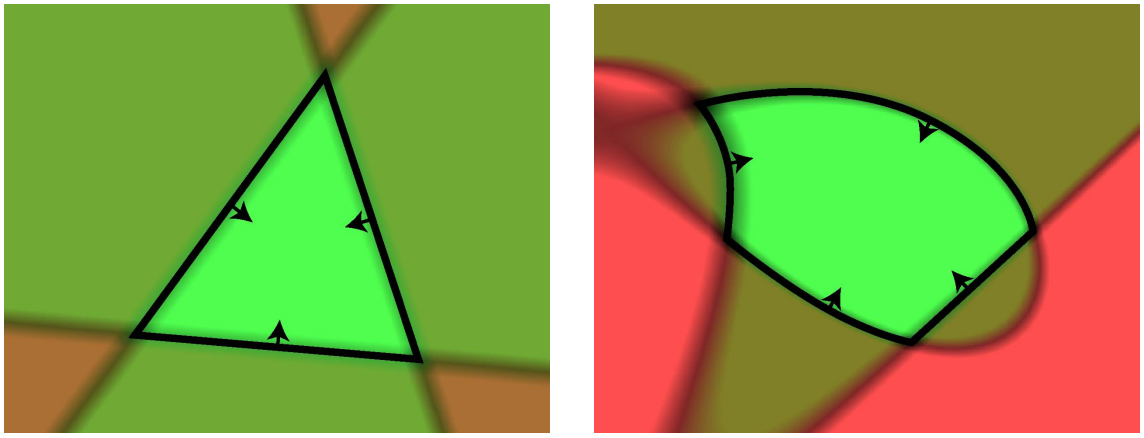


Figure 5.1: Three simple vector graphics constructed from curved patches (CPatches). All CPatches are indicated with their bounding box in blue. For efficient rasterization, auxiliary curve are added during patch cutting. Patch outlines and auxiliary curves are shown in blue and green. CPatches are rendered by our hierarchical rasterizer completely in parallel on the GPU, leading to superior performance and flexibility compared to previous work.

We propose a novel approach for vector graphics rendering in a single parallel rendering pass. Inspired by polygon rasterization, we propose a new primitive, the curved patch (*CPatch*), which is limited by a number of curves, each dividing the space into a positive and a negative half-space. The union of all positive half-spaces defines the inside of a CPatch, similar to the use of edge equations in polygon rasterization. Consequently, a CPatch can be seen as a generalization of a polygon. Even though representing the interior of a path might require multiple CPatches (Figure 5.1), all CPatches can be processed in parallel, leading to a very efficient algorithm. Hence, we make the following contributions:

- We introduce CPatches and their mathematical description.



(a) Triangle (b) CPatch

Figure 5.2: (a) Rasterization of a triangle classifies samples as inside a triangle, if all edge equations classify them as inside (green). (b) CPatches are constructed in the same spirit, with implicit curve equations classifying parts as inside.

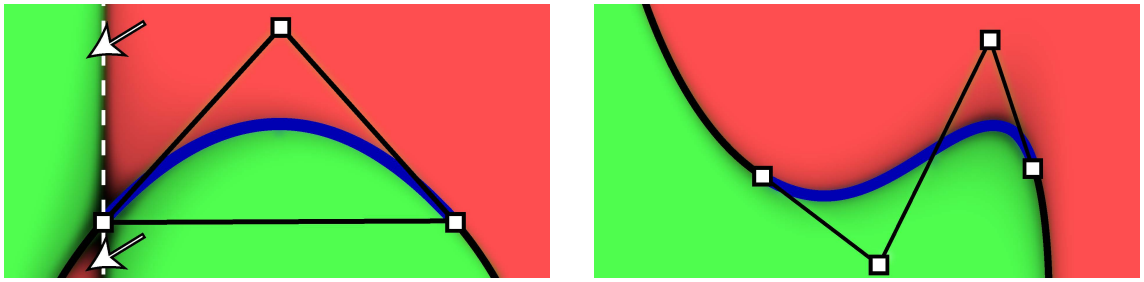
- We derive a parallel, hierarchical rasterization approach that is efficient to evaluate and very fast on current GPU hardware.
- We show how CPatches can efficiently be constructed and how arbitrary vector graphics can be translated into a collection of CPatches.

An evaluation of our approach on modern GPU hardware indicates that it outperforms previous GPU solutions by a factor of 1.17× to 1.80× on average.

## 5.1 CPatch: A novel curved primitive

Our approach is based on CPatches—primitives limited by cubic curves (see Figure 5.1 for examples). In principle, a primitive with curved boundaries can be treated in the same way as a polygon (Figure 5.2b). For a polygon, inserting into all line equations lets one determine whether a sample is inside (as shown in Figure 5.2a). Salmon [68] as well as Loop and Blinn [40] show how to translate quadratic and cubic Bézier curves into implicit form to determine on which side of a curve a sample lies: Depending on the type of the curve, three parameters $k$, $l$, $m$ are computed for each control point. A linear interpolation of these parameters and evaluation of a simple cubic function

$$f_c(x, y) = k^3(x, y) - l(x, y) \cdot m(x, y)$$



(a) Quadratic (b) Cubic

Figure 5.3: (a) The implicit form for a quadratic Bézier curve shows a sharp edge (white dashed line) outside the control polygon, which inverts the function (arrows). (b) The implicit form of a cubic follows the curve extension (black).

yields positive values for one side of the curve and negatives for the other. As the factors only need to be interpolated linearly, the approach is well suited for GPU execution. For the classification of curves into 'serpentine', 'cusp', and 'loop' and the complete table of interpolation factors, see Loop and Blinn [40].

However, the implicit function can only be used for this half-space classification within the convex hull of the curve's control points. When extending a curve to $\pm\infty$, it may reach inside the CPatch and lead to an incorrect classification of sample points. One could avoid this problem by limiting patches to the convex hull of all curves, but at the cost of limiting the supported patch types to the single-curve approach of Loop and Blinn [40]. Representing thin curved objects, such as font characters, would lead to an excessive number of patches. Instead, we split a patch into two when a curve extension reaches into the patch.

Figure 5.3a shows how, outside the convex bounds for a quadratic curve, one side of the implicit function continues along the extension of the curve (black extension to the right), while the other one changes abruptly (inverting at the white dashed line). In contrast, the extension of an implicit function for a cubic curve essentially follows the curve's extension when running through the parameter from $-\infty$ to $\infty$, as shown in Figure 5.3b. This behavior is preferable, as it is more predictable, and the locations of the sign change in the implicit form can be reconstructed from the explicit formulation. Therefore, we elevate all quadratic curves to cubics [17] and limit our discussions to the cubic case in the remainder of this thesis.

Note that, similar to triangles, CPatches only describe the interior of a primitive and not the shading of the boundaries. Thus, similar to previous work, we do not consider line shading as part of our approach. However, lines can be described by

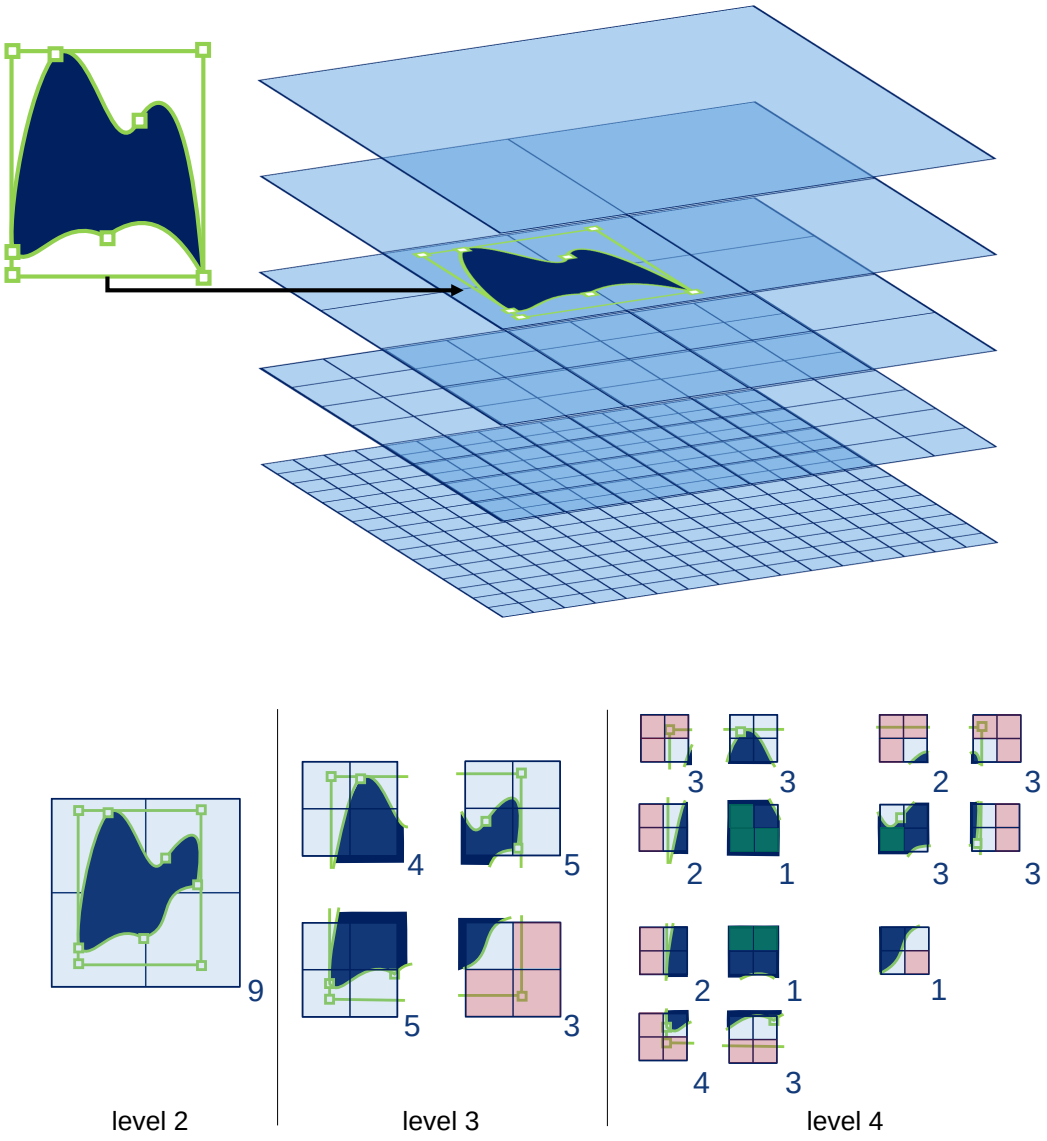



Figure 5.4: (left) Our hierarchical tiling starts by choosing the most fitting hierarchy level for the CPatch. (right) We process the patch down to the lowest hierarchy level. Sub-tiles are classified as completely outside (red overlay) or completely inside (green overlay), if the patch does not require any more testing. For tiles that are classified completely inside, the enclosing curve can be removed from further sub-tile testing (blue numbers indicate the number of actives curves).

CPatches. For solid strokes, CPatches are easy to derive as two 'parallel' curves in combination with two end curves, which are easy and efficient to rasterize.

## 5.2 Hierarchical rasterization of CPatches

Constructing vector graphics from a collection of primitives has multiple advantages: First, all primitives can be treated completely in parallel without any constraints imposed by a multi-pass approach, such as *stencil, then cover*. Second, rendering can be implemented as a streaming pipeline, which keeps resource requirements low. Third, we can establish primitive order to address issues like a correct blending order.

### 5.2.1 CPatch representation

Before detailing our hierarchical rasterization approach, we need to give an exact definition of a CPatch. We limit CPatches to consist of a predefined maximum number of curves (four to eight curves have proven to work well in our experiments) inside a given bounding box (represented as four lines). We allow curves to be either straight lines or cubics; quadratic curves are elevated to cubics. For straight lines, we encode the line equations in $k$, $l$, $m$ form, such that $l$ is the signed normal distance to the line, while $k = 0$ and $m = 1$ everywhere. While this approach slightly increases the evaluations for straight lines, it offers the advantage of a uniform treatment with only slightly increased computations. Note that we treat circular segments separately, as discussed at the end of the section.

To represent curves, we interpolate $k$, $l$, and $m$ over the entire space of the patch using homogeneous rasterization [53]. We can define $k$, $l$, and $m$ for three arbitrary points in space—any three control points are good choices—and store them in vector form:

$$\mathbf{k} = [k_0, k_1, k_2]^T, \quad \mathbf{l} = [l_0, l_1, l_2]^T, \quad \mathbf{m} = [m_0, m_1, m_2]^T.$$

Furthermore, we store the transformation matrix $\mathbf{M}$ that captures the location of the interpolation points in space, at which $[x_0, y_0]^T$ is the location where $k = k_0$, $l = l_0$, and $m = m_0$:

$$\mathbf{M} = \begin{bmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ 1 & 1 & 1 \end{bmatrix}^{-1}.$$

For any sample point $\mathbf{s} = [x, y, 1]^T$, we can interpolate $k, l, m$:

$$\mathbf{u} = \mathbf{M} \cdot \mathbf{s}, \quad k_s = \mathbf{k}^T \cdot \mathbf{u}, \quad l_s = \mathbf{l}^T \cdot \mathbf{u}, \quad m_s = \mathbf{m}^T \cdot \mathbf{u}.$$

Transformations can be applied by multiplying $\mathbf{M}$ with any $3 \times 3$ transformation matrix. While we only consider 2D operations here, it is straight forward to extend our homogeneous rasterization to 3D, as long as patches remain planar. Similarly to the interpolation of $k$, $l$, and $m$, other parameters, like texture coordinates or color gradients, can be stored along a patch.

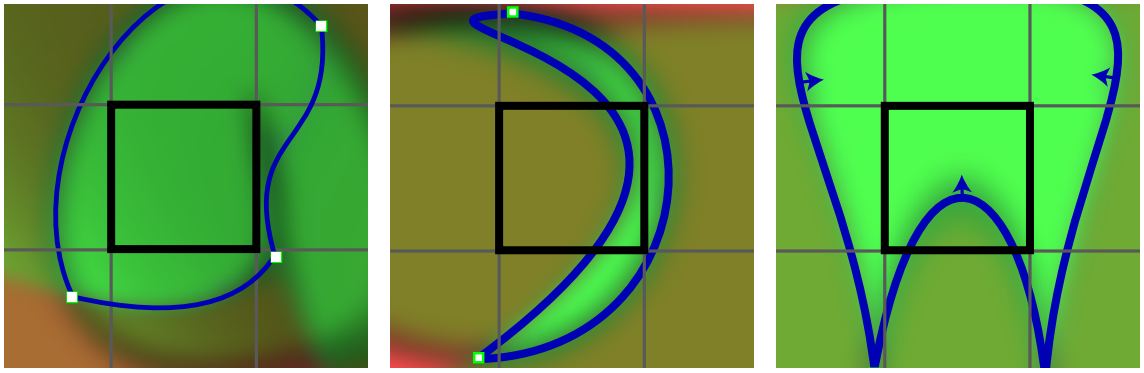
Commonly, a curve will be shared by multiple CPatches, *e.g.*, to construct a larger complex shape. Therefore, we propose an indirect storage format, similar to indexed triangle meshes. We store each curve separately ($\mathbf{k}$, $\mathbf{l}$, $\mathbf{m}$, $\mathbf{M}$), and represent a CPatch as a constant-size array of references to curves, padded with null pointers if necessary. Moreover, the CPatch stores a primitive id to look up additional shading parameters.

### 5.2.2 Tiled rasterization

A naive rasterization of CPatches would evaluate all curve equations for all pixels and fill those that lie in the intersection of all half-spaces. The main cost of such an approach is in the curve equation evaluation, which we would like to reduce as much as possible. Large homogeneous regions, which have the same classification, should be determined without visiting individual pixels. This consideration suggests a divide-and-conquer approach. We would like to concentrate on the regions close to curve boundaries, while the interior area can be filled in a single step.

**Hierarchy** Our hierarchical tiling approach is illustrated in Figure 5.4): Starting from the bounding rectangle of the patch, we determine the first level in the hierarchy where a patch should be tested. From there, the hierarchical rasterization removes irrelevant curves, while proceeding through the levels. All tiles of a level are processed in parallel. When the lowest level is reached, a fine rasterization determines the pixel fill state.

In the inner loop of this algorithm, we must determine whether a curve equation is uniformly positive or negative with respect to a given tile. Unfortunately, this test is complicated by the fact that boundaries are not lines, but implicit curves. Hence, testing the corners of a tile is not sufficient, as there is no guarantee that the curve does not change orientation between sample locations, as shown in Figure 5.5c. Furthermore, there is no efficient closed form solution to determine whether an entire tile is on one side of the curve, as this would require inserting two bounded linear functions into a cubic equation, leading to a higher order polynomial.



(a) Inside tile (b) Intersections (c) Double crossing

Figure 5.5: Our tile rasterizer relies on the fact that curves reach to infinity and thus determining sign changes along the tile boundary is sufficient to identify sign changes within a tile.

**Tile evaluation** For an efficient alternative solution to the problem, we rely on two facts. Since straight lines and cubic curves extend to infinity, it suffices to ensure that the implicit curves do not change sign along any tile boundary (Figure 5.5).

For this purpose, we rely on the intermediate value theorem. By determining the extremal values of a curve equation on the tile boundary, we determine whether there is a sign change.

We evaluate the curve equations at the tile corners and then look for the location of extrema in-between by constructing the interpolation factors along an edge: Let $\mathbf{c}_0$ and $\mathbf{c}_1$ be two corners of the tile edge. We compute the interpolation factors of $k$, $l$, $m$ in a $3 \times 2$ matrix $\mathbf{I}$:

$$\mathbf{I} = \begin{bmatrix} \mathbf{k}^T \\ \mathbf{l}^T \\ \mathbf{m}^T \end{bmatrix} \cdot \mathbf{M} \cdot \begin{bmatrix} c_{0,x} & c_{1,x} - c_{0,x} \\ c_{0,y} & c_{1,y} - c_{0,y} \\ 1 & 0 \end{bmatrix}.$$

Using $\mathbf{I}$, we can evaluate the curve equation anywhere on the edge by multiplying with $\begin{bmatrix} 1 & i \end{bmatrix}^T$, where $i$ is the relative location between $\mathbf{c}_0$ and $\mathbf{c}_1$. The general equation for evaluating the curve,

$$\begin{aligned} \mathbf{f}_{klm}(i) &= \mathbf{I} \cdot \begin{bmatrix} 1 & i \end{bmatrix}^T \\ f_c(i) &= f_k(i)^3 - f_l(i) \cdot f_m(i) \\ &= (I_{k0} + i \cdot I_{k1})^3 - (I_{l0} + i \cdot I_{l1}) \cdot (I_{m0} + i \cdot I_{m1}), \end{aligned}$$

has the derivative

$$f_c'(i) = 3 \cdot (I_{k0} + i \cdot I_{k1})^2 \cdot I_{k1} - I_{l0} f_{m1} - I_{m0} I_{l1} - 2 i f_{l1} I_{m1}.$$

We set $f_c'(i) = 0$ and directly solve the quadratic equation in $i$. If the found extrema lie within the tile border bounds ($0 < i < 1$), we evaluate the curve equation at these locations, again using $\mathbf{I}$, and determine the minimum and maximum along each tile border.

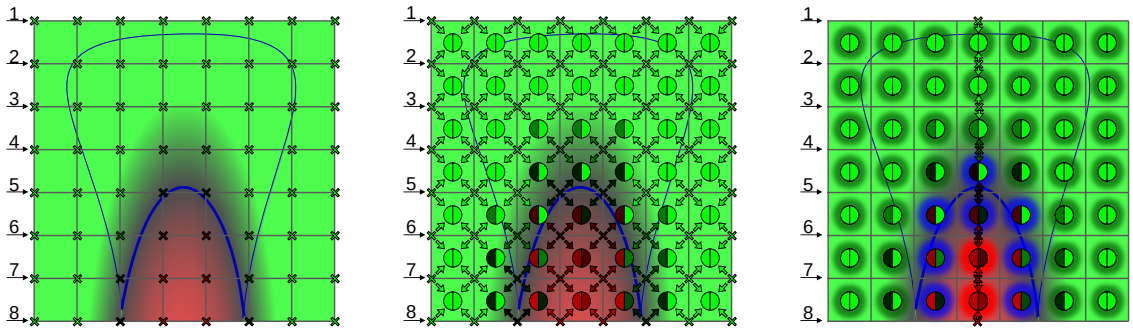



Figure 5.6: Operating on an entire tile sub-grid with eight threads, we reduce the overall number of operations. Example for the strong blue curve: (1) parallel corner evaluation, (2) min/max update, shown as circles, (3) row extrema computations and update. After column extrema computation (not shown), most tiles can be classified as either inside the curve (green glow) or outside (red glow), while only eight tiles still need to test for the curve (blue glow). Note that the center tile is only classified correctly due to the extrema.

**Parallel evaluation** Performing the above steps individually for all tiles would be inefficient, as the same computations would be repeated many times. Thus, we perform the evaluation on a sub-grid of tiles at once. Multiple threads can be employed for this evaluation, as shown in Figure 5.6 and Algorithm 1: We determine $\mathbf{I}$ for all rows of the grid using one thread per row. In step (1) (line 3–4), each thread evaluates the curve equation for all corners in its row. In step (2) (line 5), it applies the result to the surrounding tiles, updating their min/max. In step (3) (line 6–8), we determine the extrema for each row and update the min/max only for the touched tiles. Finally, we switch to columns and perform the min/max updates as well (line 9–13). This scheme reuses $\mathbf{I}$ for both the corner evaluation and the extrema computation, performing all computations only once for multiple tiles.

While iterating over all curves that define a patch, we only add those to the tiles that can still influence it (line 18). In particular, if a curve completely marks a tile

**Algorithm 1:** Parallel Tile Rasterization

```
1  for all curves ∈ CPatch do
2      for all grid rows r in parallel do
3          compute c_{r,0}, c_{r,n} and I to get f_{c_r}(i) for the curve
4          for i ∈ [0,1] with increase 1/(n−1) do
5              evalute f_{c_r}(i) and MinMax to surrounding tiles
6          compute e_i from f'_{c_r}(i) = 0
7          for all 0 < e_i < 1 do
8              evalute f_{c_r}(e_i) and MinMax to surrounding tiles
9      for all grid columns c in parallel do
10         compute c_{c,0}, c_{c,n} and I to get f_{c_c}(i)
11         compute e_i from f'_{c_c}(i) = 0
12         for all 0 < e_i < 1 do
13             evalute f_{c_c}(e_i) and MinMax to surrounding tiles
14     for all tiles in parallel do
15         if Max < 0 then
16             discard tile
17         else if Min ≤ 0 and Max ≥ 0 then
18             add curve to tile
19 for all non-discarded tiles in parallel do
20     if tile has no curves or final level is reached then
21         forward to fine raster
22     else
23         forward with curves to next level rasterization
```

as outside, we discard the tile (line 16). After completing the step for one patch, we have created a per-tile curve list, *i.e.*, a new CPatch structure for each tile to pass down the hierarchy (line 23). A tile with an empty curve list that is not marked as outside can be passed on to the fine rasterization stage immediately (line 21).

**Fine rasterizer** The final rasterization stage (the fine rasterizer) is called for a final tile and only needs to evaluate the remaining curve equations for all (sub-)pixels. The fine rasterizer operates in parallel over all pixels and can make use of efficient on-chip memory on the GPU. If multi-sampling is desired, a coverage bitmask is forwarded to the final shading stage.

**Circular curves** Half-space classification for circular curves only requires the center and the radius; tile-circle intersection is simply derived from line-circle tests. The only difference to the infinitely extending curves discussed above is that a circle can be completely inside a tile, a situation that is trivial to detect. Taking all this into account, we employ the same parallel tile test as for curves.

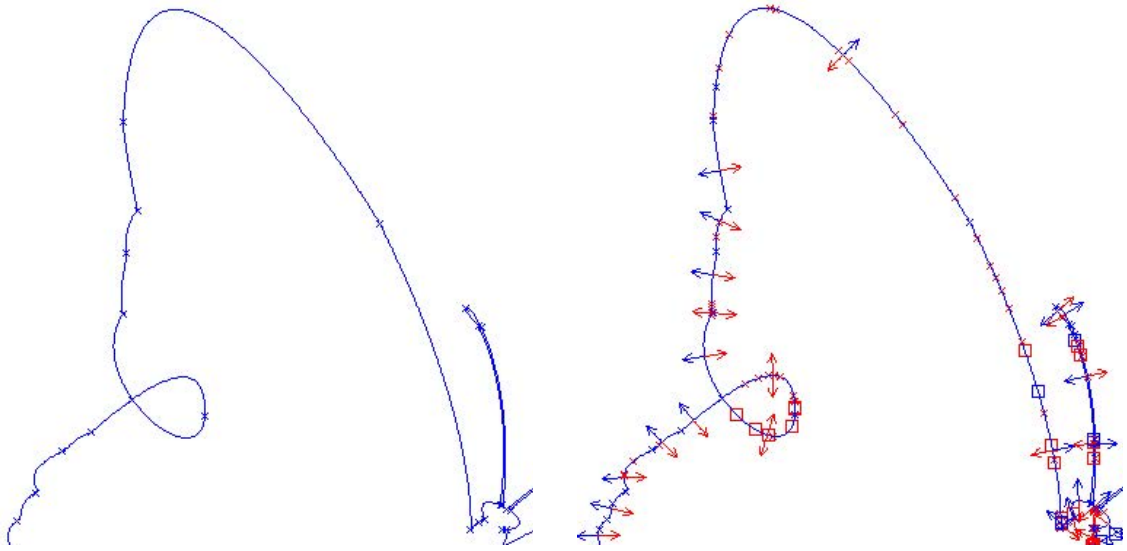
### 5.2.3 GPU software rasterizer

To show the benefits of our proposed scheme, we discuss an implementation operating on the GPU in compute mode. To take advantage of the many-core architecture of the GPU, we want to perform as many operations in parallel as possible. This problem is complicated by different entry points into the tile hierarchy, the varying number of hierarchy levels to traverse, and the varying amount of parallelism per patch, mainly owed to its size. Moreover, blending needs to respect the depth order of patches.

To take best advantage of the parallelism of the problem, we rely on Whippletree [74], a task-scheduling framework based on CUDA. We use two task types: the tile rasterizer and the fine rasterizer. For the tile rasterizer, we generate multiple instances supporting a range of sub-grids, from which we choose the one most fitting the CPatch. We use grids sizes of $1 \times 7$, $7 \times 1$ and $7 \times 7$, each using eight threads, which allows Whippletree to fill up a warp (32 threads executing on a SIMD core) with four tasks.

The input data to the tile rasterizer includes the level, and the id of the tile to be rasterized. As the execution of tile rasterizer on different levels is identical, Whippletree can combine tasks for different levels for efficient computation. For example, four tiles of size $1 \times 7$ taken from different levels can be combined for one warp. The fine rasterizer uses a grid of $8 \times 4$ threads, each responsible for a single pixel. For sub-pixel coverage, we use a bitmask for multi-sampling, while super-sampling treats all sub pixels individually. Using one thread for all sub-pixel samples achieves better performance than using one thread per sub-pixel sample.

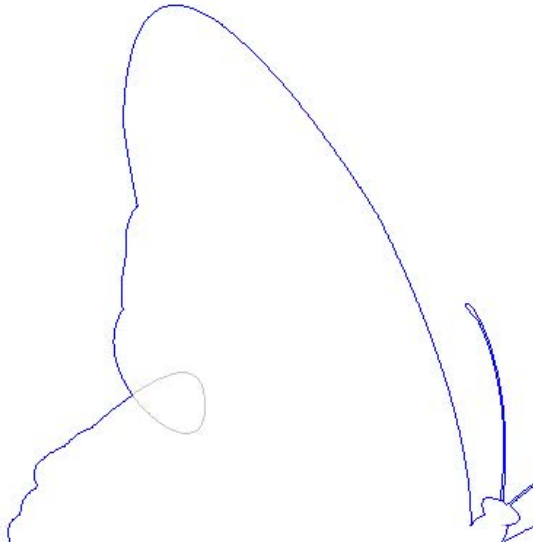
**Tile store** Finally, we need to resolve blend order. For order independent transparency in conventional polygon rendering, a common approach is constructing per-fragment linked lists [90]. We could employ a similar approach for CPatch blending, storing samples that lie inside patches in dynamically constructed linked lists. This approach would require many lists and all samples would need to be generated and stored, before consuming any of the data.



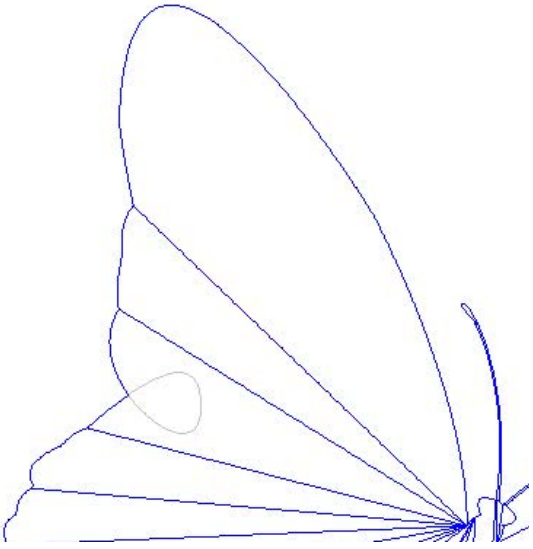
(a) Graph Flatten



(b) Fill Scoring

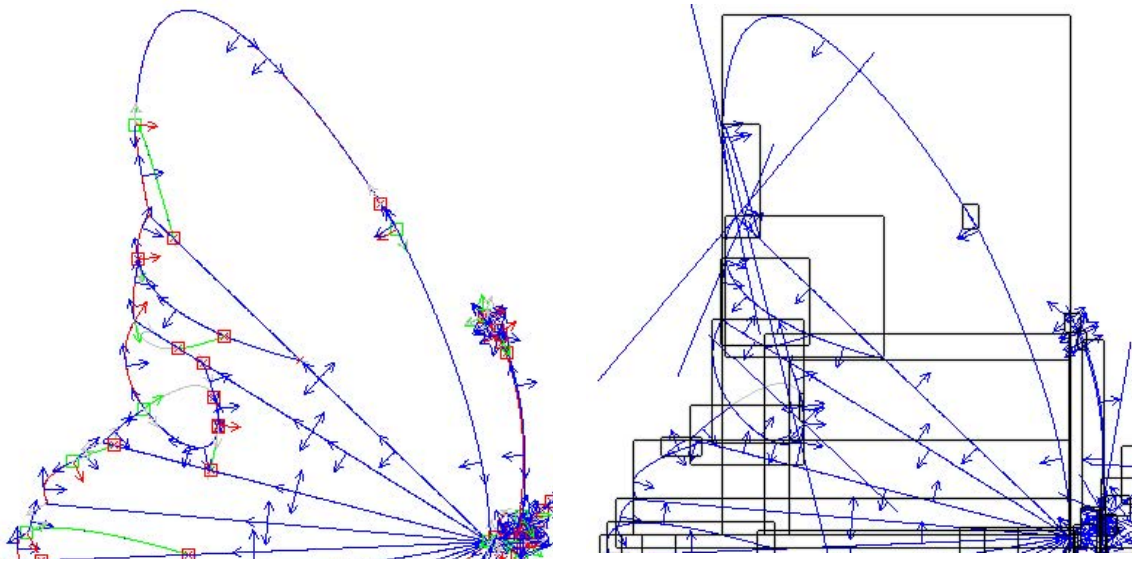


(c) Cycle Extraction



(d) Patch Cutting

Figure 5.7: Our six-stage conversion pipeline for arbitrary vector graphics to a CPatch representation: (a) A flat graph is constructed by computing all curve intersections. (b) By shooting two rays for each edge, the fill score is determined. (c) After removing edges with identical fill score on both sides, complete cycles are extracted. (d) Cycles with many curves are cut down to smaller patches.



(e) Self-intersections (f) Extension Correction

Figure 5.7: (e) Self-intersecting curves are handled by splitting patches along the self-intersections. (f) Additional straight lines are added to cut away wrongly filled outside areas.

Thus, instead of storing lists for each fragment, our tile-based rasterizer can be extended to create lists for the final tiles. This strategy allows to delay the execution of the fine rasterizer to a second pass operating on sorted tile lists. Each list entry needs to store the CPatch data, *i.e.*, the remaining curve references and the primitive id. This design implies a trade-off: While the number of lists is reduced significantly in comparison to per-pixel lists, the amount of data stored per entry is larger. Nonetheless, the resulting memory requirement is usually lower. Moreover, sorting becomes less expensive, as its cost is proportional to the number of lists.

Upon closer inspection, this approach closely resembles triangle rasterization on NVIDIA hardware: The hardware pipeline assigns primitives to tiles for final rasterization [26]; processing is carried out with a parallel sorting step before final rasterization [62]. However, our approach can still not be classified as a full streaming solution, since it temporarily stores all output data and performs a complete sort. A full streaming approach could reduce sorting cost further, but would require a more complex implementation.

Note that we evaluate shading only in the final pass, which inherits properties of deferred shading. We operate on the sorted lists from front to back and stop list traversal as soon as full opacity is reached. This not only reduces the shading and

blending cost, but also the rasterization cost. In case advanced blend modes are needed that do not support front-to-back processing, the process can be reversed.

## 5.3 Converting vector graphics to CPatches

In the last section, we have described a hierarchical rasterizer for CPatches. For a complete pipeline, it is left to show that general vector graphics can be represented as CPatches. To this end, we present a simple conversion pipeline. Our current implementation takes an scalable vector graphics (SVG) image as input, and converts all its path elements to a CPatch representation. Strokes must be converted to filled paths in a preprocessing step. The converter supports lines, quadratic Bézier curves, cubic Bézier curves and circular arcs, with non-zero and even-odd fill rules. The six stages of our pipeline are outlined in Figure 5.7, including examples of each stage.

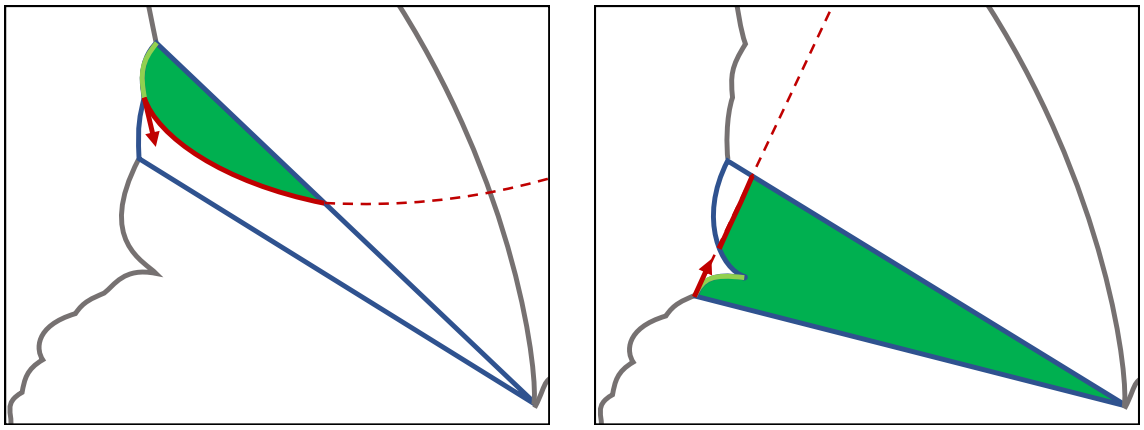
**Graph flattening** SVG paths can have arbitrary cycles and overlaps; intersections of curves are not explicitly captured in the SVG. To simplify later processing, we build a flattened graph for each path: We iteratively add curves to the existing path representation, until the complete path is captured by the graph. When adding a new curve, we determine matching nodes in the graph (end points of curves) and perform curve-curve intersection testing with all existing curves via Bézier clipping [70]. For each intersection, we add a new node to the graph and break open the curves at the intersection. This process results in a flattened graph for each path, where nodes capture all intersections of curves, and edges represent segments of the original curves connecting to the nodes.

**Fill scoring** For each flattened graph, we determine where the path should be filled. While fill scores are typically defined for each sample in the drawing, we only determine the fill score for each graph edge to either side of the curve. To this end, we shoot a ray normal to the edge at the half-way point of the curve. For each ray, we determine the fill score by computing the intersections with all curves and applying the fill score rule accordingly (non-zero or even-odd). The result of the fill score test is stored with each edge. If both sides of the edge yield the same fill score, we simply remove the edge, as it is not relevant for the drawing. Note that this computation is very light-weight, as we only determine the fill score twice per edge and not per sample in the drawing.

**Cycle extraction** As CPatches should represent primitives, we extract cycles in the graph at an early point. In this way, we can later ignore interaction between

loosely connected sub-patches. To perform the extraction, we start with a random edge and walk along the graph. At each node, we choose the curve with the smallest outgoing angle to the incoming edge, considering which side of the edge should be filled. For this angle computation, we compute the derivative of the involved curves at the node. When we encounter the starting edge again, we have extracted a full cycle.

In this way, we separate each path into multiple independent cycles. The outlined approach works well even if cycles are touching. However, nested cycles need additional treatment, as both the outer and the inner cycle are required to construct a CPatch representation. From each inner cycle we shoot a ray to find the first outer cycle and split ring-like paths into two separate touching cycles, as shown in Figure 5.7c. Note that there could be multiple inner cycles. In this case, we first connect the inner cycles and then make the connection to the outermost cycle to avoid 'cutting' inner cycles.



(a) (b)

Figure 5.8: Self-intersections of patches arise, when a curve reaches back into the patch and wrongly classifies parts as outside the patch, which can happen when a curve (light green) extension points inward the patch (a), or comes back (b). Cutting the patch in two resolves the problem.

**Patch cutting** While cycles, per definition, already form a patch, they might consist of a large number of curves. As we limit the number of curves for efficient rendering, we cut cycles that exceed the limit, using an algorithm inspired by ear clipping [47].

Although this heuristic is rather simple, it worked for all drawings we tested. In some cases, a large number of additional nodes are inserted when straight lines

```
1  while Patch has more than MaxCurves curves do
2      for N ← MaxCurves − 1 to 1 do
3          for all edges in patch do
4              mark edge and next N-1 edges
5              connect end-points of marked edges with line
6              if line does not intersect any edge then
7                  split off marked edges and make new patch
8                  add line to original patch
9                  break
10     if Patch still has more than MaxCurves curves then
11         split longest edge in the middle and add new node
```

cannot be placed in the interior. Curved cuts could be an option to avoid these additional nodes.

**Self-intersection cutting** Self-intersecting curves are one of the major challenges for CPatch generation, since they lead to incorrect half-space classifications, as shown in Figure 5.8. We distinguish two cases, a curve which extends into a patch from its starting node (Figure 5.8a) and a curves that returns into the patch after leaving it (Figure 5.8b).

We handle self-intersections by cutting the patch in two. We iterate over all curves of the patch and test the derivative at the end nodes to determine whether the curve points inward. Then, we compute all intersections of the curve extension with the patch. We simply extend the curve to a large multiple of the original length using the De Casteljau algorithm [14] and again perform Bézier clipping to check for intersection. According to our experiments, the alternative of solving for intersections using the implicit curve is more time consuming and less accurate.

After finding all intersections, we sort them and split the patch in two (Figure 5.8). After creating the two new patches, we continue the process for both newly generated patches. For efficiency reasons, we retain the information about which curves of the new patch have already been tested. Curves that have loops need special treatment, if a complete loop is formed by the extension. In this case, an additional patch only consisting of the loop might be needed.

**Extension correction** One final issue concerns curves that cross outside of the patch, but within the bounding box. This might create wrongly filled areas, as shown

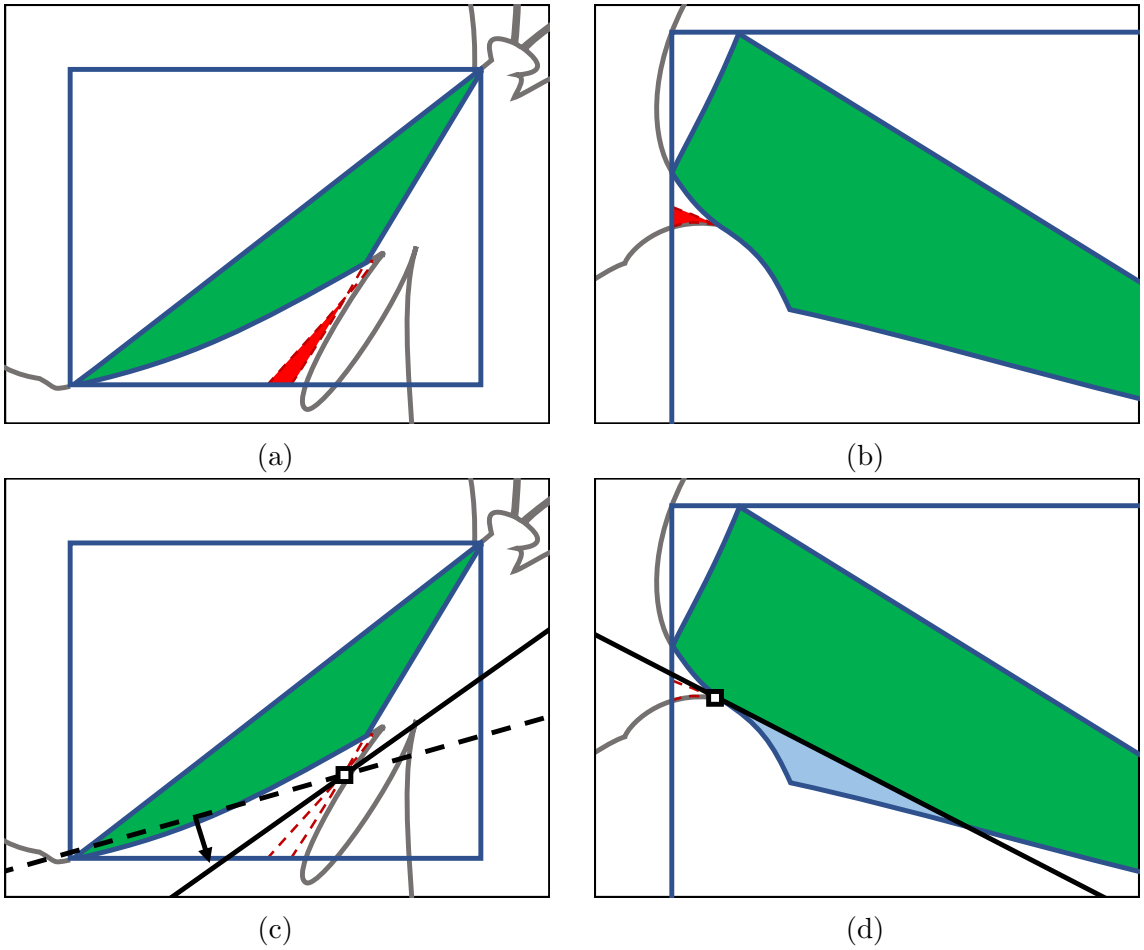



Figure 5.9: (a) Extension correction is necessary when curve extensions cross outside the patch and thus wrongly classify regions as inside. (b) Such errors are also common for cusps, where they happen directly next to the patch. (c) We fit an additional line to cut these regions. (d) Sometimes, a straight curve cannot successfully perform the cut, in which case we split the patch in two.

in Figure 5.9. Again, this issue might arise directly at the end of the curve, *e.g.*, with cusps (Figure 5.9a), or from an intersection of two curve extensions (Figure 5.9b). These unwanted regions can be handled by locating them and pruning the offending crossings by inserting an additional curve to the patch.

To locate such offending crossings, we consider all intersections of curve extension (which are guaranteed to be outside of the patch after the execution of the previous stage) as well as all intersections of curve extensions with the bounding box. For each of those points, we evaluate all implicit curve equations and keep only those that yield a wrong result. From the offending crossings, we construct connected

| | Input | | CPatch | | Tile Raster | | |
|---|---|---|---|---|---|---|---|
| Data | Pth. | C. | Ptc. | C./Ptc. | Ptc. | Ptc./Tile | C./Ptc. |
| drops | 204 | 1k | 1k | 3.58 | 18k | 2.24 | 1.09 |
| embrace | 225 | 5k | 4k | 3.20 | 43k | 5.35 | 1.37 |
| tiger | 240 | 2k | 3k | 3.38 | 22k | 2.74 | 1.81 |
| car | 420 | 12k | 7k | 3.33 | 38k | 4.75 | 1.95 |
| sample_v2 | 691 | 7k | 7k | 3.29 | 34k | 4.21 | 2.28 |
| hawaii | 1137 | 53k | 41k | 2.83 | 102k | 12.48 | 2.21 |
| boston | 1922 | 28k | 14k | 3.23 | 46k | 5.71 | 1.85 |
| paris-70k | 45k | 545k | 303k | 3.36 | 531k | 64.91 | 2.88 |
| contour | 53k | 188k | 57k | 3.42 | 115k | 14.06 | 2.81 |

Table 5.1: Statistics of the test data sets and processing results. The input SVG daasets range from 200 to 53k paths (Pth) with up to 545k curves (C). Our preprocessing generates up to 303k patches (Ptc) with an average of about 3.3 curves per patch. After tile rasterization (1k resolution), lists capture up to half a million patches.

cycles (there might be multiple).

Then, we find the point that is closest to the original patch—for cusps, this could even be a node of the patch. We use this point as anchor and place a line to cut the wrong region, which we add as a curve to the patch. There are infinitely many line directions to consider. We optimize by starting with a random direction and rotate it depending on where we hit the falsely positive region (or the patch), as shown in Figure 5.9c. We iterate with a reduced rotation angle, until we find a fitting direction or end up with no movement. In case no solution is found, we cut the patch in two (Figure 5.9d).

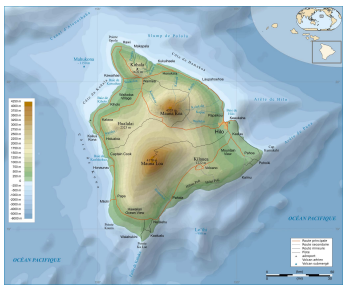
**Remarks** Even though the preprocessing sounds complex, our non-optimized, single-threaded CPU code runs efficiently. For example, it loads and processes the Tiger image (Figure 5.1, right) in less than a second. Given our simple implementation, there is a large optimization potential. Furthermore, a CPatch representation only needs to be constructed once; it could easily be stored as additional information alongside the vector drawing. Especially when using our technique in a graphics editor, such as Adobe Illustrator, only one path is edited at a time, and thus only a single CPatch representation needs to be computed, which can easily be done at interactive rates.

(a) embrace

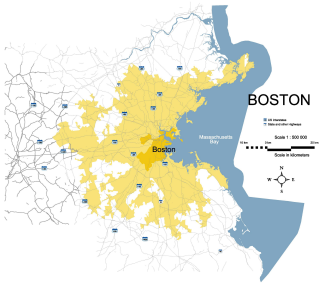
(b) tiger

(c) car

(d) sample_v2

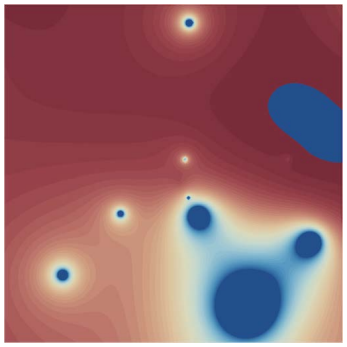


(e) hawaii




(f) boston


(g) paris-70k



(h) contour

Figure 5.10: The test data set spans simple ($\approx$ 200 paths a,b), medium sized (1000 to 2000 paths e,f), and large graphics ($>$ 40 000 paths g,h).

## 5.4 Results

To evaluate the performance of our approach, we tested a variety of common vector graphics benchmark drawings, as outlined in Table 5.1 and Figure 5.1 and 5.10. All tests were run on an NVIDIA GeForce GTX 1080Ti (3584 CUDA cores, 11GB of global memory) hosted by an Intel Core i7-6850K CPU 3.60GHz with 64GB of system memory. As comparison methods, we use NVIDIA path rendering [29] (NV) and the GPU scanline rasterizer (SL) of Li et al. [36] . We use their original published implementation.

Our approach uses a tile size of $8 \times 4$, a maximum number of four curves per patch, and every element in the tile list can hold 32 elements. For sorting, we use per-block radix sort and choose the best fitting block size among 32, 64 and 128, depending on the average guessed number of patches per tile. While these choices put slightly more pressure on the preprocessing and increase memory requirements, they favor speed.

**Preprocessing** As can be seen in Table 5.1, our preprocessing usually cuts each input path into 5 to 40 CPatches on average, creating up to 300 000 CPatches for the largest input. Drawings with smaller and more complex curved structures, *e.g.*, *hawaii* or *boston*, are cut into more patches than rather simple drawings, like *drops*. As *contour* mostly consists of triangular and rectangular data, it is already very close to a usable CPatch representation and thus hardly needs any processing.

After tile rasterization, the overall number of patch references throughout all lists ranges from 18 000 to 530 000 (1k resolution). List lengths are relatively short on average for most simple drawings with 2 to 14 entries. *paris-70k* forms an exception with its large number of small patches. The number of referenced curves after tile rasterization is strongly reduced to 1.9 to 2.9 on average, indicating the success of the hierarchical approach.

**Timing** Performance numbers are shown in Table 5.2. When multisampling is disabled, our approach shows the best performance in eight out of nine cases for 1k resolution and four out of nine cases for 2k resolution. NV takes the lead in two and four cases, respectively. SL is always the slowest approach. For 16× multisampling, the situation slightly shifts, with our approach winning in four and three cases, NV in four and two cases, and SL in one and three cases, respectively. Overall, we achieve a mean (harmonic) speed up of 1.48× and 1.80× without multisampling and 1.43× and 1.17× for 16× multisampling over NV and SL, respectively. Our

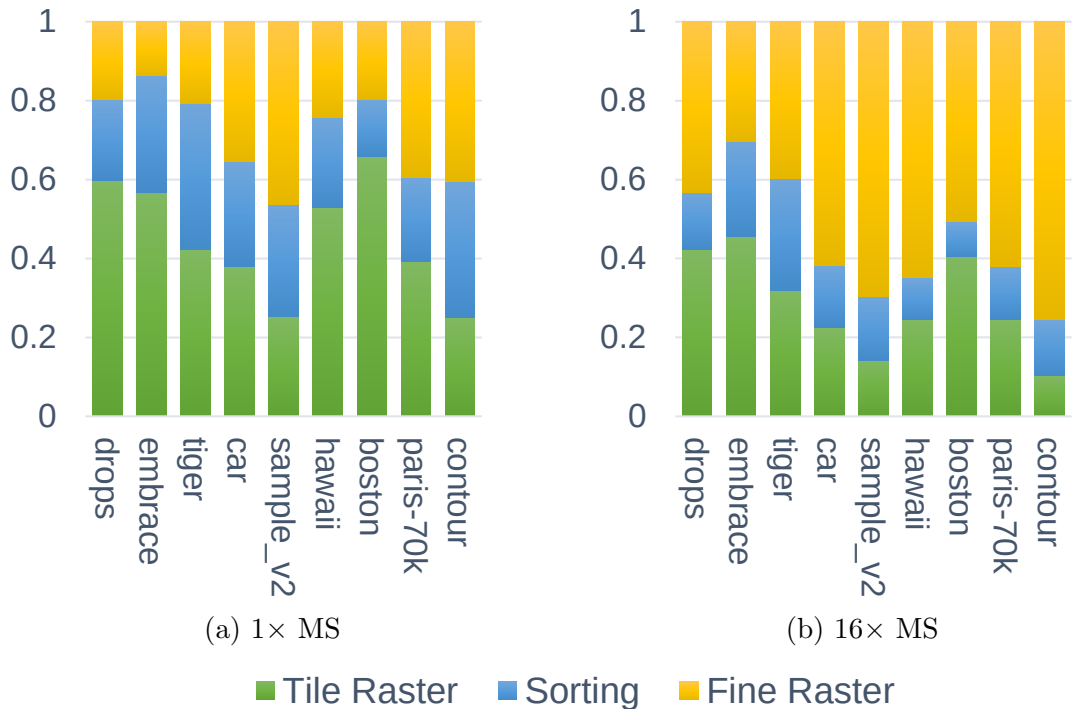
| | res | 1× Multisampling | | | 16× Multisampling | | |
|---|---|---|---|---|---|---|---|
| | | Our | NV | SL | Our | NV | SL |
| drops | 1k | **0.51** | 0.61 | 1.62 | 0.72 | **0.71** | 1.75 |
| | 2k | 1.31 | **0.61** | 1.72 | 2.12 | **1.44** | 2.10 |
| embrace | 1k | 0.75 | **0.63** | 1.95 | 0.93 | **0.84** | 2.08 |
| | 2k | 0.92 | **0.62** | 2.01 | 1.81 | **1.75** | 2.39 |
| tiger | 1k | **0.66** | **0.66** | 1.63 | 0.87 | **0.81** | 1.73 |
| | 2k | 0.79 | **0.66** | 1.69 | **1.72** | 1.75 | 2.04 |
| car | 1k | **0.82** | 1.17 | 2.16 | 1.38 | **1.12** | 2.35 |
| | 2k | **1.07** | 1.17 | 2.21 | **1.91** | 2.22 | 2.54 |
| sample_v2 | 1k | **0.47** | 2.57 | 1.37 | **0.83** | 2.57 | 1.52 |
| | 2k | **0.88** | 2.53 | 1.39 | **1.66** | 2.56 | 1.72 |
| hawaii | 1k | **0.88** | 2.07 | 2.49 | **1.90** | 2.10 | 2.97 |
| | 2k | 2.31 | **2.06** | 5.53 | **4.45** | 5.44 | 9.96 |
| boston | 1k | **0.64** | 3.42 | 1.30 | **1.04** | 3.41 | 1.44 |
| | 2k | **1.26** | 3.43 | 1.33 | 2.14 | 3.41 | **1.73** |
| paris-70k | 1k | **1.96** | 74.6 | 2.43 | 3.13 | 74.1 | **2.58** |
| | 2k | 3.52 | 72.5 | **2.52** | 4.81 | 73.5 | **2.99** |
| contour | 1k | **0.63** | 90.9 | 1.48 | **1.53** | 90.9 | 1.57 |
| | 2k | **0.85** | 90.1 | 1.55 | 3.24 | 90.9 | **1.89** |

Table 5.2: Runtime performance of our approach in milliseconds compared to NV path rendering and GPU scanline rasterization.

approach shows the most balanced performance, keeping up with NV for smaller drawings (*drops*, *tiger*, *car*) and showing very competitive performance for large drawings with complex structures (*paris-70k*, *contour*), which are typically vastly in favor of alternative approaches.

It should be noted that NV, in many cases, is not limited by the compute power of the GPU, but rather suffers from synchronization delays due to the 'stencil, then cover' approach, which reduces the amount of parallel workload. Thus, increasing the resolution or multisampling has hardly any influence for NV. While SL also follows a 'stencil, then cover' approach, they first generate strides that are then rendered in parallel by OpenGL. SL uses an approximation for stride boundaries and thus multisampling is less costly in their approach. Therefore, SL results may slightly differ visually. Our approach scales with the workload, reducing performance when the resolution is increased or multisampling is activated.

The relative performance of the three steps of our approach is shown in Figure 5.11.

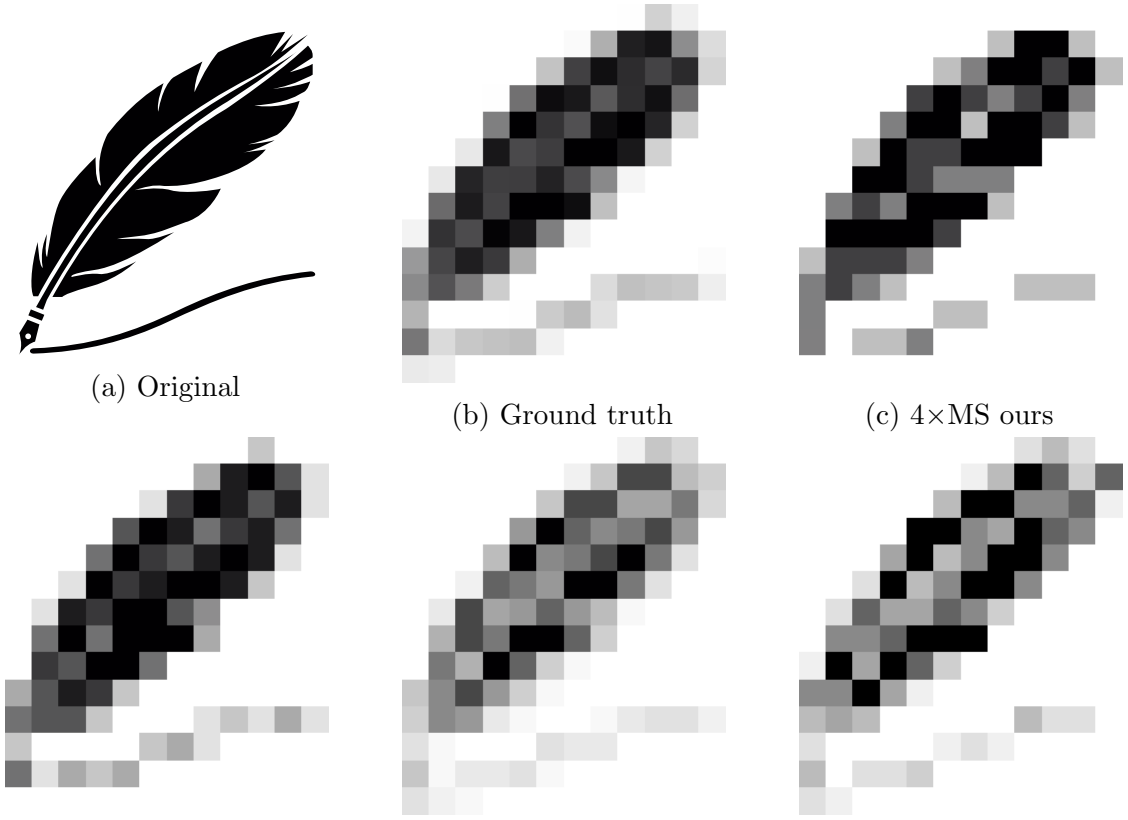



(a) 1× MS (b) 16× MS

Figure 5.11: Relative run time of the three major steps of our approach. Multisampling influences fine raster time only.

If multisampling is disabled, tile rasterization is usually the most time consuming step. Fine rasterization is slightly more costly than sorting. When multisampling is enabled, fine raster takes over the majority of the workload for most tests, which is not surprising, as the number of tested samples is increased 16×.

**Quality** Figure 5.12 shows quality examples for 8× multisampling of the tested approaches in comparison to a 256× supersampled ground truth (16×16 downsampled image). Our approach clearly achieves the best result for this challenging case (even 4× multisampling is superior in image quality). We can only speculate about the errors of the other approaches which both rely on hardware multisampling. Both NV and SL render geometry for the fine structures, which is subject to sub-pixel snapping for fixed point rasterization, which may influence the evaluated equations and generated stencils. Additionally, SL represents both scanline ends with simplified geometry, which leads to additional errors. As our approach does not perform any boundary simplifications and fully evaluates curve equations for all sub-pixels, we achieve a higher quality.



(a) Original (b) Ground truth (c) 4×MS ours

(d) 8×MS ours (e) 8×MS NV (f) 8×MS SL

Figure 5.12: Quality example for 12×14 pixels large renderings of the feather image compared to 256× super-sampled ground truth of the feather image. SL and NV show higher errors due to their treatment of sub-pixels. Even our 4×MS image achieves high accuracy in comparison to the 8×MS renderings of NV and SL.

**Discussion** While CPatches are usable for all vector graphics, some drawings, like `contour`, are already close to a CPatch representation and thus more efficient. Paths with many curves, like the butterfly in Figure 5.1, will typically get cut into many patches, which explains the high expansion factor of some drawings. However, while the number of CPatches increases, memory requirements only increase marginally when using references to the original curves.

Our approach is most efficient when rendering patches that fill out their bounding box well, *e.g.*, rectangular CPatches result in most efficient rendering. However, also thin and slanted patches can be rasterized efficiently, as empty regions are discarded early in the hierarchical rasterizer, whereas traditional 'stencil, then cover' methods would test all curves for all pixels in the bounding polygon. Thus, our approach is also well suited for boundary rasterization which naturally consist of many thin

segments.

Due to the nature of our approach, all types of fill types and blending can easily be integrated. As the fine rasterizer is executed in compute mode, not only color gradient or textures are naturally supported as fill types, but any type of computations can be performed, *e.g.*, noise evaluations, complex sampling, or lighting are possible. Similarly, as blending is performed in software, we can use any combination of color spaces and blend functions.

## 5.5 Summary

We have presented a novel approach for representing and rendering vector graphics using curved primitives, CPatches, which enable parallel rendering, similar to how triangle rasterization is performed in real-time rendering. A CPatch representation allows the construction of a complete parallel hierarchical rasterizer on the GPU. Our software prototype, running in GPU compute mode, shows competitive performance when compared to the hardware supported NVPR for small vector graphics. It performs on the same level as previous state-of-the-art methods for complex drawings, while completely avoiding all approximations. Thus, our approach not only achieves speedups of 17% to 80% over the previous state-of-the-art, but also achieves higher quality for multi sampling. Our approach shows the best performance, when the input vector graphics is already close to a CPatch representation.

To show the applicability of our approach, we have also presented a preprocessing pipeline to convert arbitrary vector graphics to a CPatch representation. We see high potential for better preprocessing in the future, which would not only increase preprocessing speed, but also generate CPatches that can be rendered more efficiently.

# 6 Visibility Sampling

## Contents



In the last decade, VR has again risen in popularity. Alongside, new technical difficulties have surfaced, mostly due to the high number of shaded pixel required at very high frame rates to establish a smooth VR experience. The combination of a large field of view to cater immersion, high pixel densities to hide display properties of near-eye displays, as well as low latency to avoid VR sickness simply overload the capabilities of most modern graphics cards.

As a consequence, alternate rendering strategies such as object space shading approaches as well es decoupled shading are rising in popularity. For example, shading can be computed on a powerful server in object space and sent to the head-mounted display (HMD), which displays the geometry for new head poses [48]. In this way, the shading rate can be reduced, while the final rendering always considers the correct view. Similarly, object-space shading [24] and decoupled shading [38] first determine which primitives are visible in a scene before shading them.

A separation into visibility, shading and display becomes troublesome, when visibility and display do not use the exact same viewing parameters, i.e., if the

camera parameters or resolution is not identical. For VR scenarios, considering small view offsets from visibility to display can effectively reduce display latency and VR sickness. Thus, it is highly desirable to support camera adjustments between those stages. When rendering the scene from a view point that does not correspond to the one where shading took place, disocclusions may occur. Disocclusions become visible as black or colorless spots on screen where no shading information is available because that location was not meant to be seen from the original pose. For this reason, the exactly visible set (EVS) of triangles need to be extended into a PVS in decoupled rendering to support a certain offset between shading and display view position. In a first person VR setting, the movement of the user can be predicted for a few frames into the future, and the triangles that become visible in the additional views can be taken into account when shading. With this approach, a simple PVS can be created to mitigate the disocclusion problem for many situations. Abrupt view changes like sudden head rotations or jumps will still cause the effect to occur [48]. An additional source of disocclusions is insufficiently accurate sampling of visibility. When determining which triangles are visible at the center of each pixel in the output surface, geometry in-between two sample points will be lost. A simple solution would be to increase the sampling resolution. This brute force approach quickly exceeds the constraints of real-time rendering. Figure 6.2 displays the problem. On the left, the triangles that could be sampled from a view point further away are shown when moving the camera closer. The center image shows the result of our approach, and the one to the right is a visualization of the triangles that could be visible from the original view point according to a ground truth we establish.

In this work, we propose a method for sampling the visibility of geometric primitives from a certain view point that also includes primitives that would otherwise fall in-between sample locations. Our approach comprises two passes, whereas the first pass is a simple G-buffer pass and the second uses conservative rasterization to compare primitives with the G-buffers. During the comparison, primitives that are likely from the same surface are identified and added to the visible set. We make the following contributions:

- We propose heuristics that consider depth differences and triangle sizes to identify primitives that are likely to correspond to the same surface. These heuristics take the specifics of conservative rasterization into account.
- We show that dynamically adjusting the heuristics depending on triangle properties increases the true positives and reduces false negative classifications.
- We identify additional considerations, such as geometric relations of primitives,

Figure 6.1: Visualizing high density hot spots of triangles per pixel where spatial aliasing occurs due to primitives missed in standard visibility sampling (marked in red). Our method can detect small triangles within these areas to include them in the visible set.

their distances and overlap to further increase the classification rate among multiple scenes.

- We solve the problem of sub-pixel-size gaps in scenes by introducing a pixel coverage measure and extend our approach to multiple passes limiting the work to only those pixels that show gaps.

We compare the effectiveness of our method by comparing to a fine grid sampling per pixel method that serves as a ground truth.

## 6.1 Sampling Methods

In this section, we describe the three methods we use to determine visibility of triangles (the last one being our novel approach). The first uses standard rasterization to record the front-most triangle per pixel. The second serves as our ground truth of what should be visible. We use the first method in a two dimensional grid to increase the sampling density. The third is our newly developed method that uses conservative rasterization to consider triangles that would not be detected using a

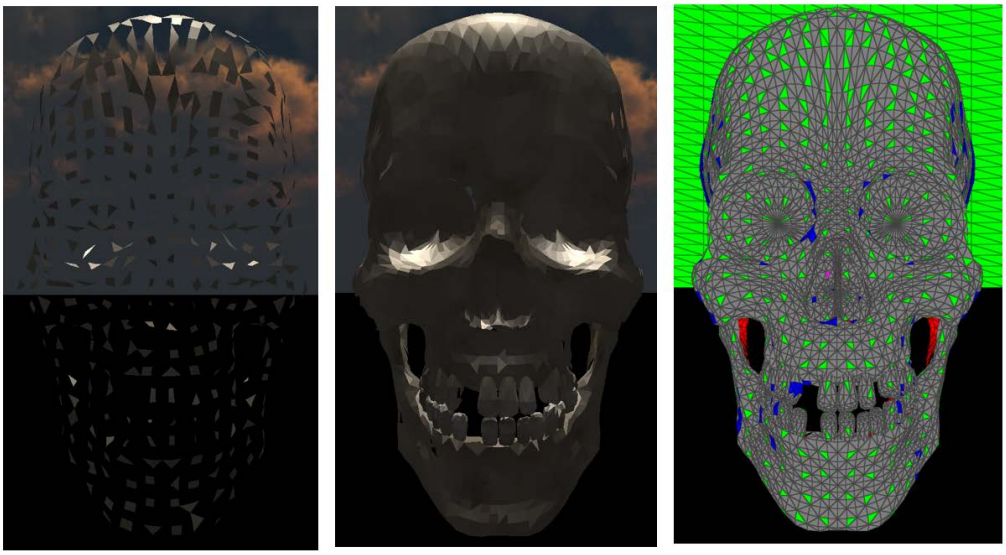


Figure 6.2: Left: Closeup of visible triangles after sampling from further away. Center: After sampling from the same view point with our method. Right: Visualization of triangle visibility. Green: Detected with standard and conservative sampling; Grey: Conservative only; Red: False positives (too much); Blue: False negatives (missing)

standard one sample per pixel approach.

All three methods are comprised of two stages. The first stage determines the triangle id for every pixel at the center, and the second stage gathers the visibility information into data structures for the shading system.

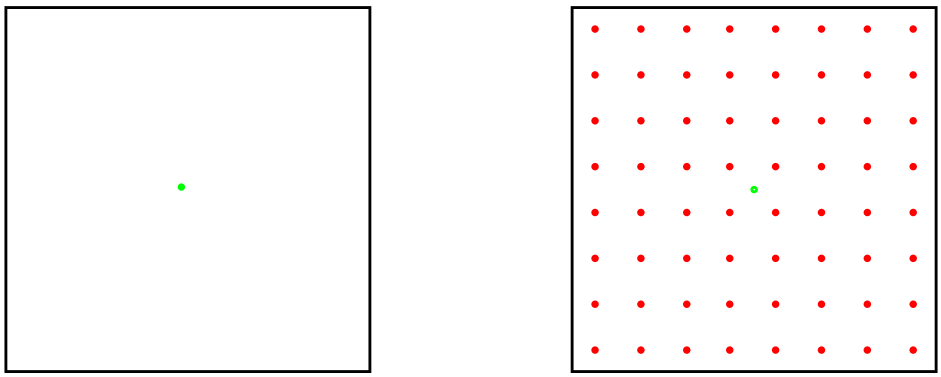
### 6.1.1 Naive Sampling

In its most simple form, the visibility is sampled by rendering the geometry with a forward render pipeline consisting of a vertex and a fragment shader. The only difference is that instead of a color value written to a framebuffer, the ID of the triangle that invoked the fragment shader in that location of the render area is saved to a texture. One fragment shader invocation, caused by the triangle that overlaps with the pixel center, is issued per pixel. We call this the standard or naive method (Figure 6.3a). The triangle ID is determined by a running index of invocations provided by the rendering API and an offset into the draw buffer that contains the primitives to be drawn. Once all draw calls for all visible models in the scene have completed, the output texture to which all the draws have written, contains only the front-most triangle ID for every pixel. We call this texture the ID buffer. The second pass in this approach has only the single purpose of gathering the IDs from

the two dimensional ID buffer into a one dimensional array of rendering primitives. This visibility stage has to be performed separately from the ID stage, so all draw calls could complete their writes before the information is read again.

### 6.1.2 Brute-Force Oversampling

To construct a method to serve as ground truth, we use the standard method within an offset area to thoroughly sample the visibility at a certain viewpoint. For our purposes, we set this region to be all possible view points within the distance of half a pixel to the left and to the right in horizontal and vertical direction of the position from where we want to sample (Figure 6.3b). The density of that grid can be set arbitrarily high (although hardware limits apply in practice) to get a more exact result of the true visible set.



(a) Standard sampling (once at pixel center) (b) Pixel grid sampling

Figure 6.3: (left) While standard sampling operates at the pixel center (green dot), our (right) ground truth. Super-sampling is generated by repeatedly running the standard sampling pipeline with sub-pixel projection offsets in an eight by eight grid in addition to the center sample (red dots).

### 6.1.3 Sub-Pixel Visibility

This novel visibility sampling approach uses two render passes (and an optional compute shader pass to further improve performance). The first pass is similar to the naive sampling but stores the minimum and maximum depth per pixel in addition to the triangles' ID. The second render pass uses conservative rasterization to invoke a fragment shader program for every pixel that is touched by a triangle. In the fragment shader we apply heuristics to keep or discard the invoking triangle. The computation of the minimum and maximum depth can be moved to a compute

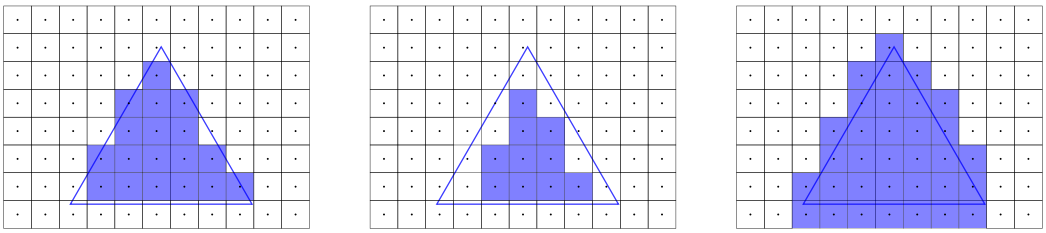
shader, launched between the first and second render pass, to only compute that information for triangles that are visible after the first pass.

#### 6.1.3.1 Conservative Rasterization

Rasterization is the process of determining which cells of a grid of screen locations (pixels), the raster, are covered by a geometric primitive. The standard behavior of this part of the graphics pipeline, which is a fixed function unit in today's implementations, is to consider a cell covered if the center overlaps with the primitive. In contrast, conservative rasterization [2, 25, 77] defines any coverage of the cell as valid:

- *Under-estimation* mode will report only those cells of the raster covered that are fully covered by the primitive.
- *Over-estimation* will mark all pixels that are at least partially overlapping with the primitive as covered.

Our visibility algorithm relies on the over-estimation mode. For every triangle, even if it falls through the sampling grid, a fragment shader invocation will be generated, so we can decide on its visibility in the executed shader program.



(a) Standard rasterization behavior fills pixels where the center is covered by the primitive.

(b) Conservative underestimation fills only pixels which are fully covered by a primitive.

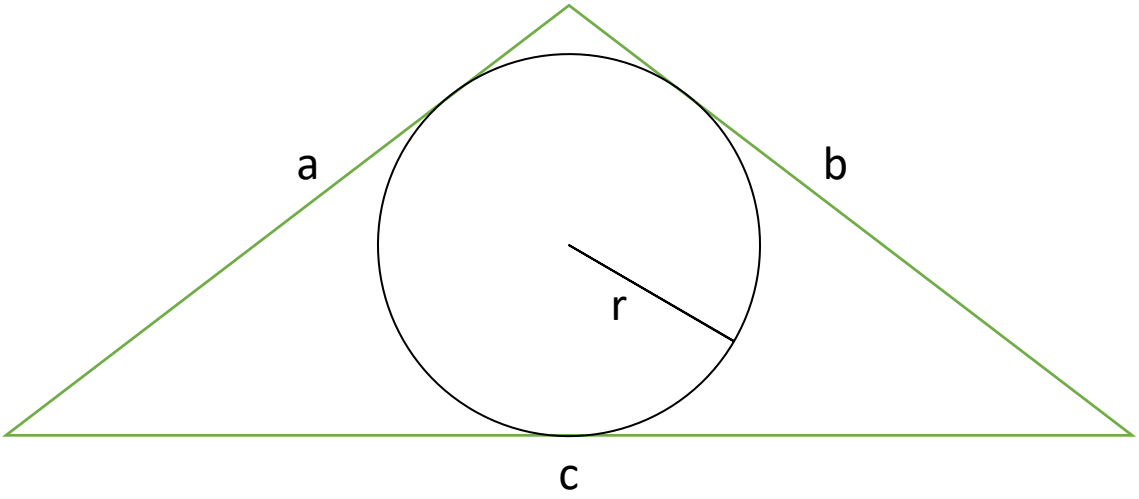
(c) Conservative overestimation, the technique our method makes use of, fills all pixels which are touched by a primitive, regardless of how much area is covered.

Figure 6.4: **Rasterization modes** supported by modern graphics hardware. The pixels in the raster (center marked with dots), will be filled with color depending on the position of the triangle and the rasterization mode.

#### 6.1.3.2 Heuristics

If every triangle that hits the fragment shader stage in the conservative visibility pass was marked visible, a lot of false positives would be generated causing unnecessarily

high load on the shading stage for triangles that will never be visible.




$$r = \frac{2A}{a+b+c} = \sqrt{\frac{(s-a)(s-b)(s-c)}{s}}; s = \frac{a+b+c}{2} \tag{6.1}$$

Figure 6.5: In-circle radius $r$ of a triangle with area $A$, side lengths $a$, $b$, $c$ and half-circumference $s$

With the help of information gathered in the first render pass and settings which enable fine tuning to some degree, we can formulate criteria on whether to keep or discard a triangle. We use the following components:

- **Triangle ID:** If the ID matches from the first pass, we accept the triangle without even rasterizing it because it is already marked visible.
- **In-circle radius:** Large triangles that get detected only in the second render pass are bad. The rationale being that they generate unnecessary high load not only at the subsequent fragment stage, especially when using conservative rasterization, but also in the later shading of the triangle of which only a very small portion will be visible. Otherwise the large triangle would have been detected in the first render pass already. We calculate the in-circle radius (Figure 6.5) for triangles that are not marked visible and discard them before rasterization if they are larger than a pixel.
- **Depth:** If the projected position of two triangles is almost the same (within the area of one pixel) and the one was already marked visible in the first pass, while the other was not, we compare their depth to determine if the triangle

that was missed in the first pass is likely to be visible or not. We compare the minimum and maximum depth of the triangle currently processed within the current fragment location to the depth stored for this location from the first render pass.

$$\epsilon = \Delta_{abs} + \Delta_{dyn} * (1 - abs(< n_s, n_t >)) \tag{6.2}$$

$$\delta = \begin{cases} d_{s,min} - d_{t,max} & \text{if } d_{s,min} > d_{t,max} \\ d_{t,min} - d_{s,max} & \text{if } d_{t,min} > d_{s,max} \end{cases} \tag{6.3}$$

$$\text{accept} = (\delta < \epsilon) \tag{6.4}$$

The depth heuristic is applied in the fragment shader of the conservative render pass. We calculate a threshold $\varepsilon$ according to equation 6.2 based on configurable parameters $\Delta_{abs}$ and $\Delta_{dyn}$. We compare this to the depth difference $\delta$ of the current triangle and the triangle sampled in the first render pass (Equation 6.3). This depth difference is based on the minimum and maximum depth of the triangle within the area of the current pixel. $d_{s,min}$ and $d_{s,max}$ are the minimum and maximum depth of the triangle sampled at that pixel in the first render pass. $d_{t,min}$ and $d_{t,max}$ are the minimum and maximum depth of the current triangle in the conservative rasterization pass.

The diagram in Figure 6.6 shows a schematic description of the situation of two triangles within a pixel. At the sample point (center vertical line), only the upper triangle (long blue line on the right) would be detected with standard sampling. Since we also get to process the missed lower triangle (shorter left blue line) with conservative rasterization, we take the depth of that triangle into consideration and measure how far it is behind or in front of the originally sampled triangle. We take two settings into account to configure our depth discard criterion, a static depth offset and a *dynamic* offset. The dynamic part of the $\varepsilon$ takes the relative position of the two triangles using the dot product of the normalized triangle normals into consideration.

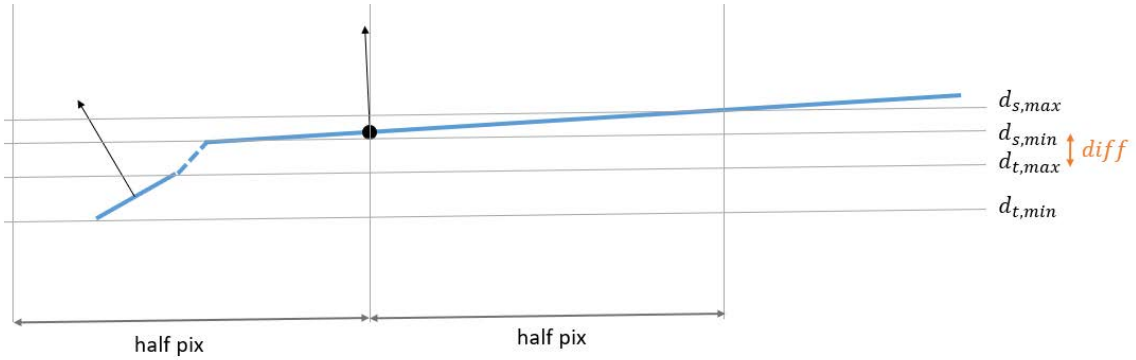



Figure 6.6: **Depth comparison heuristics:** In order to create reasonable comparisons of triangle depths, we do not directly look at the depth value at the sample point in the middle, but determine the minimum and maximum depth within the pixel boundary. We also take the triangle's normals into consideration to adjust the boundary at which we discard a triangle dynamically according to their relative angles.

## 6.2 Implementation

Our implementation of the newly developed conservative visibility method is realized in a decoupled shading system [48]. In contrast to their implementation of a fully fledged VR streaming system, we use a reduced set of functionality that serves as a testbed to better focus on the research problem of improving the visibility sampling.

The rendering system is implemented in the C++ and GLSL programming languages and makes use of the Vulkan graphics API and its conservative rasterization extension, supported since version 1.0.67 [27].

Our software design evolves around a pipeline with the individual stages depending on the tested scenario. After an initialization that sets up buffers for input and output data, the visibility stage is the first stage where render passes happen. This stage can be exchanged at compile time with classes implementing the visibility producer interface. Our test application uses three separate pipelines, one compiled with a standard visibility stage, one with the conservative visibility variant and one for the view cell version. The pipeline can then be switched at run-time. The output of the visibility stage is subsequently used as the input to the shading stage, that renders the triangles marked visible into a shading atlas [48]. At the end of the pipeline, there is the display stage that simply renders the visible triangles by mapping the generated shading information.

The system is decoupled in terms of being able to decide what rendering work to

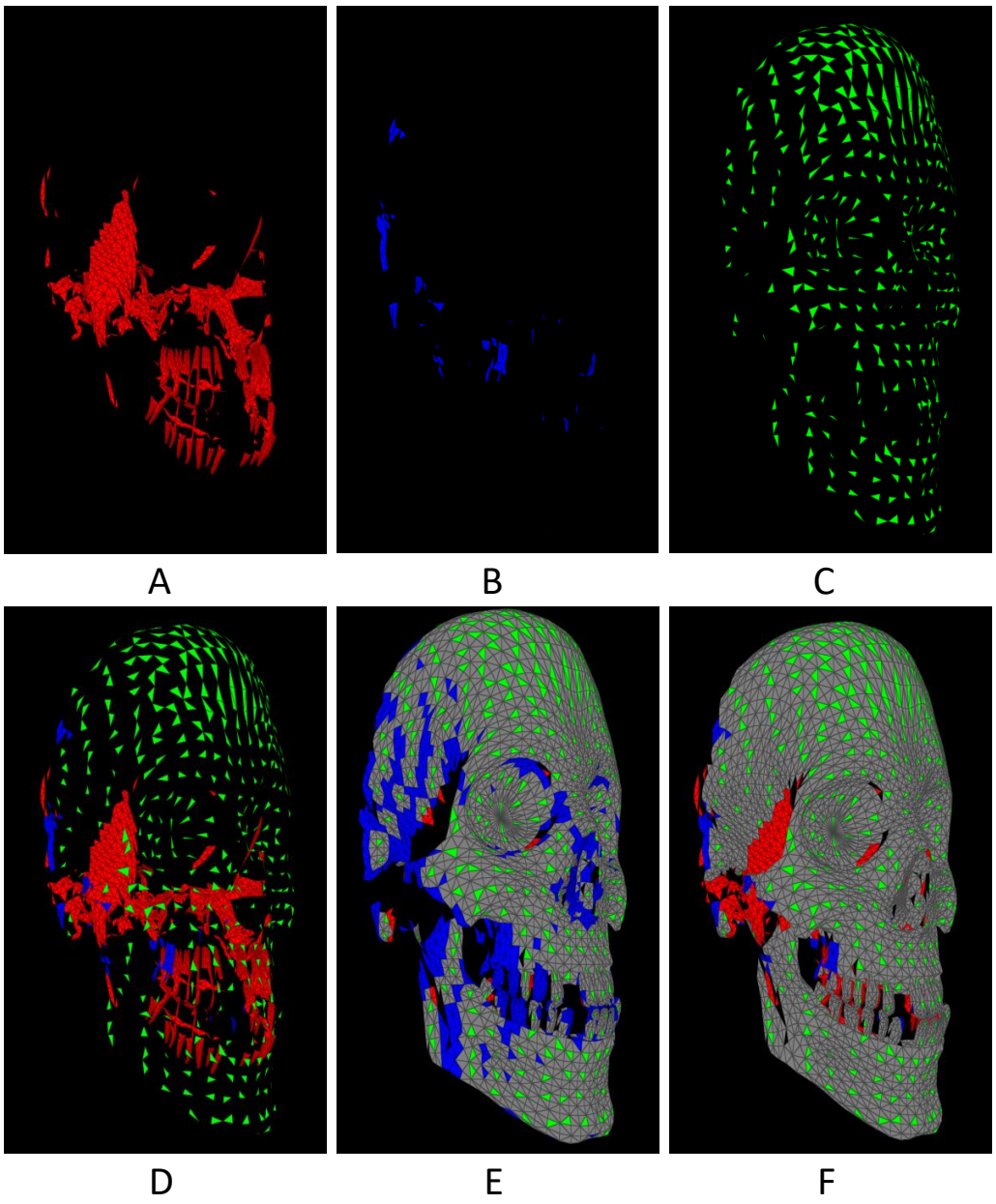



Figure 6.7: **Color coded visibility visualization:** A: False positives (too much), B: False negatives (missing), C: Standard sampling, D: Union of A, B, C. E: Heuristics too strict, F: Heuristics relaxed; Grey triangles in E and F: Correctly recognized by the conservative method (according to ground truth and in addition to the green triangles, which are of course also marked visible by the conservative method)

process each frame. The stages up to the shading stage are recorded into a different Vulkan command buffer [28] than the display stage. The command buffer containing the shading part can be submitted to the Vulkan system at a different rate than the one containing the display stage. The matrices containing the view information are updated separately for the shading and the display stage, allowing to generate several fast view updates from one shading update. With this mechanism, we can also pause the shading updates and inspect what has been rendered - a behavior used extensively in our visibility visualization renderer (Figure 6.7).

The high level mechanism already describes the standard visibility sampling adequately. One implementation detail we might add here is that we use a compute shader in the visibility gathering sub-stage to quickly prepare the data for the subsequent shading.

### 6.2.1 Conservative Visibility

In the first render pass of the conservative visibility method some extra computation is carried out in addition to filling the ID buffer with IDs of the primitives that pass the depth test and are closest to the camera at the center location of each pixel.

A geometry shader is used to compute and store the triangle's normal vector and store it in a buffer containing per triangle entries. The only other thing this shader program does is pass down the triangle's vertex data to the fragment shader where it is used to determine the minimum and maximum depth (in clip space coordinates) of the triangle plane intersected with the bounds of the pixel area.

To accomplish this, we first perform a point-in-triangle test to find out if any of the pixel's corner points fall within the triangle. If this is the case for all four pixel corners, we intersect a ray perpendicular to the pixel plane from the corner points to intersect with the triangle plane. The minimum and maximum depth of the four intersections will be stored together with the triangle ID in the pixel's coordinate of the ID buffer. If not all four pixel corners are covered by the triangle, we need to clip the triangle's edges against the pixel border planes. After clipping, we gather the minimum and maximum depth values from the edge's endpoints in addition to computing that information from the pixel corner points that *are* covered by the triangle.

Now, for the second render pass, we activate the conservative rasterization feature in the Vulkan API. Again, a geometry shader provides the the triangle normal and vertices to the fragment program with one addition - the screen space in-circle radius.

In the fragment shader, the minimum and maximum depth are computed like in the previous pass. The difference of the computed depth values and the sampled values is used to compare to the $\varepsilon$ that is also computed with the help of the normals. If a triangle then conforms to our heuristics, we mark it as visible for the subsequent shading stage to consider.

## 6.3 Results

To evaluate our visibility sampling approach, we not only conduct timed test runs to gather performance metrics, but also test the impact of the individual heuristics to assess their effectiveness.

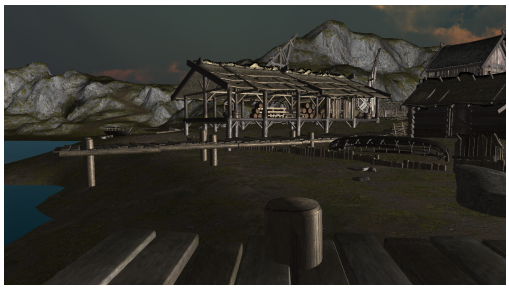
### 6.3.1 Test configuration

Testing of our algorithm was mainly done with the Viking Village scene, freely available from the Unity3D engine [82]. This is not only a popular scene for testing VR systems, but also a very demanding one. It features models of skulls that are used for decoration along the pathways of the village and on the walls of several houses. The geometric detail of this model is very high with every root of a tooth modeled with great care. Having models with such a high degree of detail in a scene is usually not wanted for simple decorative items. However, since dealing with inappropriately modeled items is a real-world scenario, and the problem of geometric aliasing becomes clearly visible with them, the skulls serve the purpose of demonstrating our technique very well. In Figure 6.1, the red spots show the high triangle density in spots where the skulls are located. In figures 6.2 and 6.7, we render one single skull with debug visualizations to have a closer look at how our sampling method deals with the geometry.

Apart from visual inspection, we also ran synthetic tests with a prerecorded walk-through in Viking Village to test a variety of settings and their effect on our method. Further test scenes are the Robot Lab (also a Unity3D scene [81]), Sponza from Crytek [13], and a scene containing a grid of cows (using Spot [12]).

The output quantities we measure are frame times, coverage and overestimation. Coverage and overestimation are comparisons to the output of the ground truth method. The tests were carried out on a Desktop PC running Windows 10 on an Intel Core i7-7700, 32 GB RAM and an NVIDIA Geforce 1080 Ti.

(a) Robot Lab



(b) Viking Village

(c) Sponza

(d) Grid of Cows

Figure 6.8: Test scenes used for evaluation.

### 6.3.2 Dense visibility sampling as ground truth

In order to compare the effectiveness of our method, we need to establish a ground truth. To determine which triangles are truly visible from a certain view point, we thoroughly sample the scene within a two dimensional grid of views, or view cell.

This implementation applies the standard sampling technique with a few modifications to prepare for sampling at offsets from the original view position. The sampling at the pixel center stays the same. The offsets are 2D coordinates altered by a fraction of pixel width according to the configured grid size. In addition to the offsets, the original position is also added to the list.

When recording the Vulkan drawing commands, we do this in a secondary command buffer that will be resubmitted to the Vulkan execution queues for every location of the grid, changing the view position in-between.

The dense sampling is computationally expensive, but a sufficiently thorough measure to use as a ground truth is necessary. We have tested configurations up to 512 by 512 sampling points and decided, that for our purposes, we use a 64 by 64 plus one (for the original view) sized grid, to be sufficiently accurate as it gathers 99.5% of the triangles detected by the maximum grid size that was practically possible. Larger grids than 512 by 512 would require more time and memory than justified by

the diminishing returns as can be seen in Figure 6.9 With a rendering resolution of 1920 x 1080, the processing times per frame for the 64 by 64 grid are in the order of seconds. One frame at 512 by 512 takes several hours.

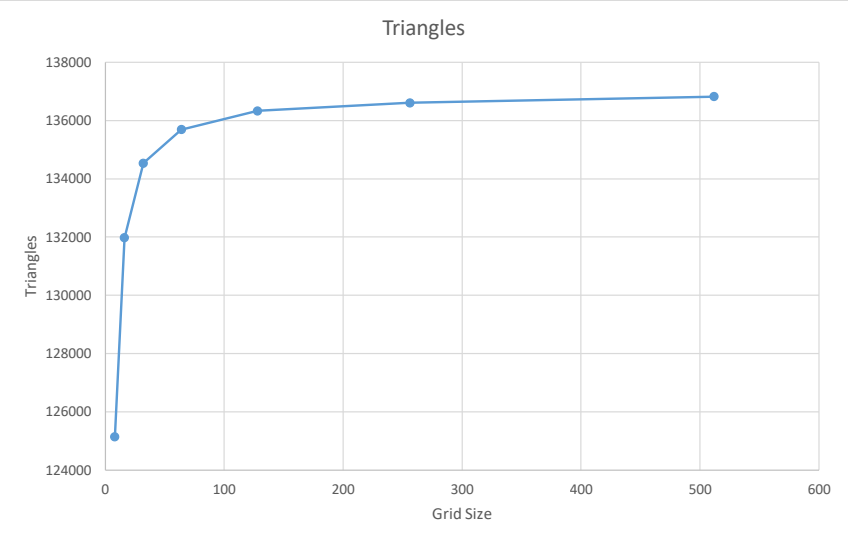



Figure 6.9: **Pixel grid effectiveness:** Increasing the amount of super-sampling per pixel yields diminishing returns. With a detection rate of 99.5 percent (relative to the largest configuration tested) we settled for a grid size of 64x64 samples per pixel to serve as the ground truth we compare to.

### 6.3.3 Sampling Resolution

Due to the nature of conservative rasterization, generating a fragment that we can process as soon as a triangle touches the area of a pixel, we expect our approach to work well already at resolutions lower than the one used for final display. If this assumption holds, we should gain an advantage in performance.

We run tests at resolutions 25%, 50%, 75% and 100% of the output resolution and show run-time results in Figure 6.10 and Table 6.1 for Viking Village. In Figure 6.11, we see the effect of the lower resolutions on coverage and overestimation. Already at quarter resolution, we achieve an increase in triangle detection rate of about 10% compared to standard sampling.

### 6.3.4 Depth Delta Variations

In Figure 6.12, we show the results for a series of configurations for the static and dynamic depth $\Delta$ settings. We cover a range of sensible values, which we found from

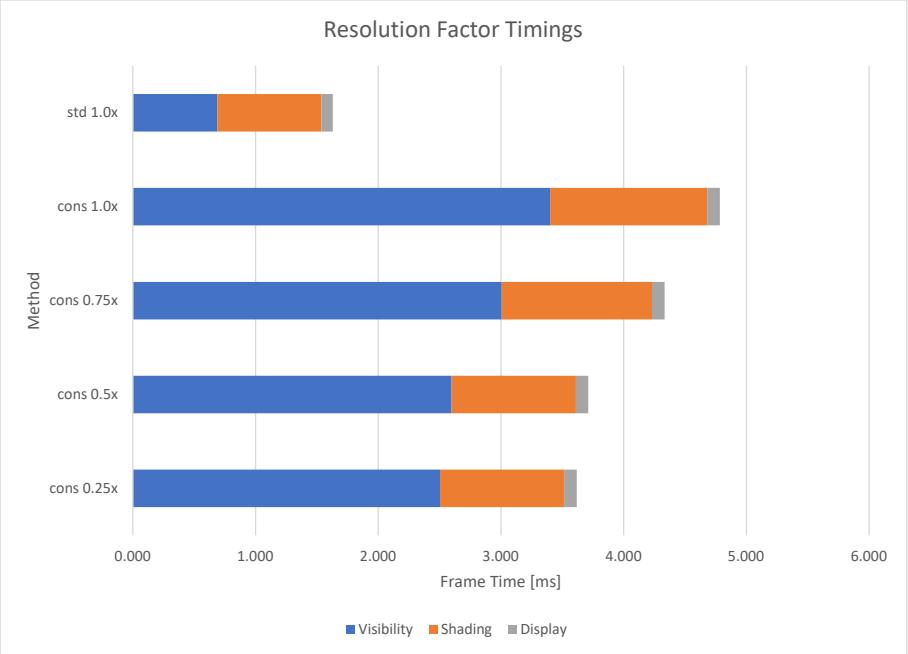



Figure 6.10: Frame times of the conservative visibility in various resolutions compared to standard sampling. Again, *cons* is our method using conservative visibility, *std* is standard sampling. 0.5x etc indicates the proportional resolution (e.g., $0.5 \cdot 1920 \times 1080 = 960 \times 540$)

| Configuration | Visibility | Shading | Display | Frame Time | FPS | Rel % |
|---|---|---|---|---|---|---|
| cons 0.25x | 2.503 | 1.011 | 0.104 | 3.618 | 276.396 | 0.450 |
| cons 0.5x | 2.597 | 1.013 | 0.101 | 3.710 | 269.511 | 0.439 |
| cons 0.75x | 3.005 | 1.227 | 0.102 | 4.333 | 230.761 | 0.376 |
| cons 1.0x | 3.401 | 1.279 | 0.103 | 4.783 | 209.064 | 0.341 |
| std 1.0x | 0.688 | 0.849 | 0.091 | 1.629 | 613.946 | 1.000 |

Table 6.1: ***Resolution timing results Viking Village*** of our decoupled shading rendering prototype in milliseconds. The conservative visibility method (cons) has been configured with several resolutions, lower than the resolution of the output surface. Static and dynamic $\varepsilon$ values have been set to 2.0. The standard sampling method (std) is only run in full resolution.

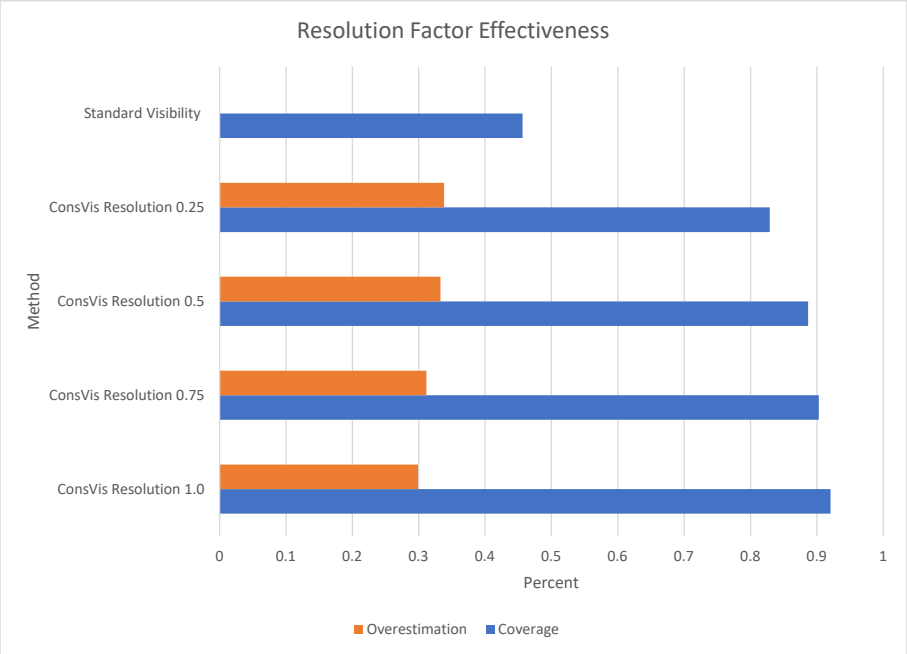



Figure 6.11: Results for coverage and overestimation rate compared to ground truth in the Viking Village scene when running our visibility detection method at various resolution settings (factors of the full output resolution)

manual experiments to be quite suitable, as well as some exaggerated settings to test out limits.

The gain in coverage shows diminishing returns upon increasing the two $\Delta$ settings, whereas overestimation increases steadily. The effect of the static $\Delta$ is, of course, solely dependent on the scene's geometry.

### 6.3.5 View Cell Sampling

A view cell in the scope of computer graphics describes the union of all view points within a region [7]. To see how our approach behaves when combined with other PVS algorithms which might use multiple view points, we constructed a test case where we sample visibility from different view points. The view points are arranged in a grid as shown in Figure 6.13. To achieve this, we simply add an offset to the view matrix before running the visibility pipeline and repeat this procedure for all view offsets. Although the area of the original view point is already covered by the grid (Figure 6.13 (a), we also test the original position to make sure we do not miss any triangles in the slight variation in sample points.

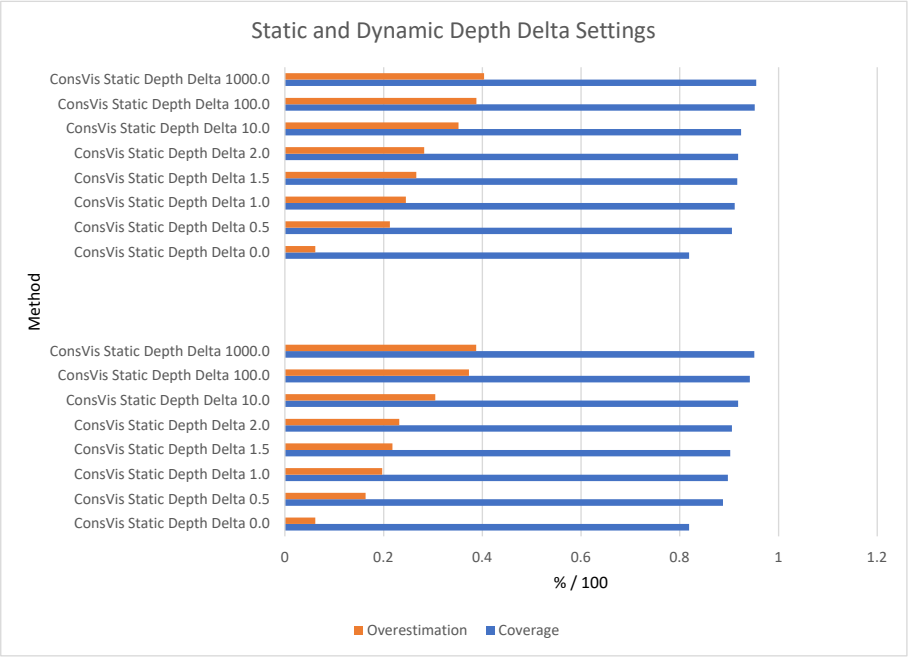
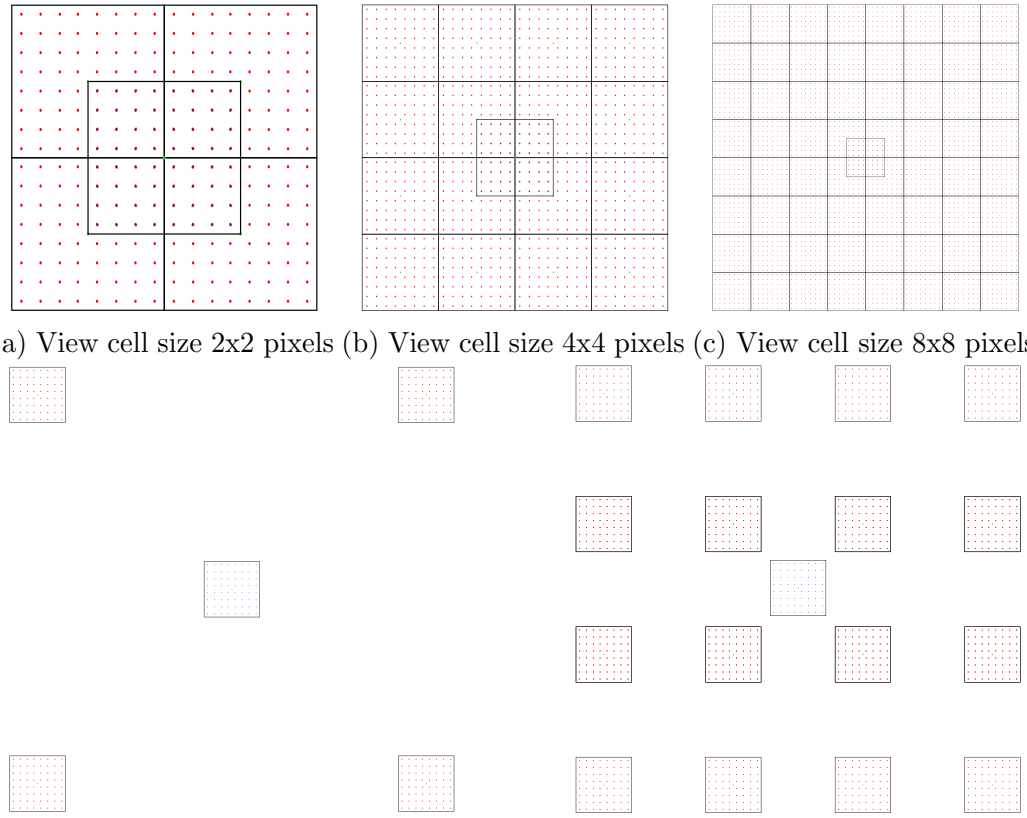



Figure 6.12: Showing the effect of various static and dynamic depth $\Delta$ settings on coverage and overestimation rate compared to ground truth in the Viking Village scene.

In Figure 6.14, we plot the tested view cell configurations and their effect on coverage and overestimation. To achieve a coverage of 95%, it is sufficient to increase the view cell to a two by two grid. These tests were run at full resolution (1920 x 1080 pixels). This should still be feasible in terms of real-time performance, even though the frame times would quadruple. The larger the view cell becomes, the more overestimation of visible triangles occurs. Since our approach tends to generally overestimate, this amount increases proportionally to the area covered. By increasing the view cell, we give the visibility sampling stage more ground to generate false positives.



(a) View cell size 2x2 pixels (b) View cell size 4x4 pixels (c) View cell size 8x8 pixels

(d) View cell size 8x8 pixels, sampling grid size 2x2

(e) View cell size 8x8 pixels, sampling grid size 4x4

Figure 6.13: **View cell configurations:** By adding offsets to the current view position we cover a view cell area from two by two up to 32 by 32 pixels in steps of powers of two (configurations up to eight by eight shown in (a), (b) and (c)). We run these configurations for our method and the ground truth to test if we can improve coverage or minimize overestimation. We also investigated the impact of skipping sample positions within the view cell (shown in (d) and (e))

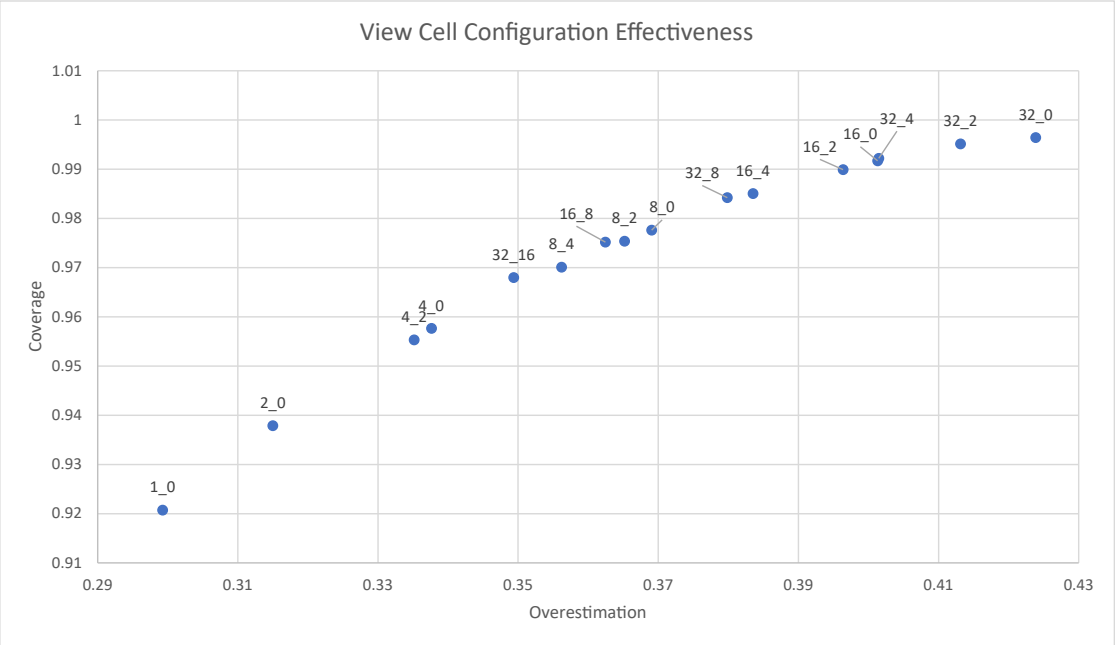



Figure 6.14: **View Cell Configurations:** The scatter plot shows overestimation versus coverage results of various view cell configurations measured in Viking Village. The data point labels indicate the view cell size in the first number and the amount of skipped view points in the second number. $32_8$, for example, indicates a 32 x 32 pixels wide view cell where only every eights view point is sampled—resulting in a 4 x 4 grid.

## 6.4 Summary

We have presented a novel method of sampling the visibility of triangles in 3D scenes. Our approach considers geometry that would normally be missed via standard sampling. The additionally detected primitives improve visual fidelity especially in the setting of decoupled shading. In this scenario, these tiny missing triangles can already cause disocclusion artifacts when the display view position is only slightly different to the view point when shading of the last full frame took place.

We conducted a wide variety of tests to prove the capabilities of our approach. Results have shown that the method is suitable for real-time rendering and already achieves good results at lower resolutions.

# 7 Conclusion

## Contents



To conclude this thesis we summarize the work and its achievements. Furthermore, we attempt to formulate take-away messages and list aspects and issues that were left open and lend themselves as starting point for future work.

In this thesis we present novel approaches to geometry processing problems with a focus on real-time graphics applications leveraging latest advances in software scheduling frameworks, compute and graphics API. Under this common theme of parallel geometry processing we offer a new view on procedural modeling by introducing operator graphs, we propose methods of representing and rasterizing vector graphics and improve visibility sampling for decoupled rendering scenarios.

**The operator graph** concisely describes the generation process for any given input object and is applicable for a variety of generation methods. Given such a graph we can reason about how to efficiently execute the mass model generation code for the specific input on the GPU. By partitioning the operator graph we generate compilable code that executes as procedures in a scheduling system. Occupancy of the streaming multiprocessors depends on various factors like how well a given

schedule fits the type of GPU used, the thread divergence caused and the memory access patterns that occur.

As the number of possible partitions of the operator graph depends on the length of the input description, an exhaustive search is only feasible for small graphs. Therefore we propose several heuristics to search the space of solutions to find a schedule that generates the output geometry within the shortest time.

Before describing three search heuristics that accommodate different aspects of execution properties intrinsic to the nature of GPU hardware, we define the concepts of static and dynamic scheduling. The former fuses nodes of the operator graph to execution groups at compile time, while the latter involves run-time scheduling decisions. Static partitioning prolongs the time a single thread has to work on a subroutine. In contrast, a dynamic scheduling decision involves slow communication over global memory, but has the ability to initiate the execution of several work items at once, increasing the ability for parallel execution.

Based on the two fundamental types of scheduling decisions we use *sequence fusion*, *divergence avoidance* and *execution group size* to prune the space of partitions of a graph. The first heuristic applies static scheduling to nodes of the graph that do not have multiple output edges. As no parallelism can be gained, we spare ourselves the scheduling overhead with this measure. With the second heuristic, we try to anticipate the unpleasant condition of thread divergence, introduced by conditional execution paths. Therefore, we avoid having variable sized strands in the subsequent path of derivation by forcing dynamic scheduling at occurrences of this kind of nodes. Lastly, the third heuristic enforces a minimum execution group size to avoid multiple dynamic scheduling decisions in sequence after each node in the graph. This balances the time taken between scheduling, operator interpretation and geometry generation.

**CPatches** are a new way of representing and rendering two dimensional vector graphics. While rasterization engines in current graphics hardware are optimized to the large scale processing of triangles, this novel geometric primitive enables parallel processing in a similar way by our hierarchical rasterization approach.

Common vector graphics formats like PostScript or SVG describe drawings by a list of paths describing contours to be filled. The paths may be represented by lines, Bézier curves of second or third degree or elliptical arcs. We convert these input descriptions to a set of patches consisting of an implicit curve representation. A patch signifies a (sub-)region of the drawing bounded by curves.

A crucial property of CPatches is the strict avoidance of self-intersection of the

curves with the area of the patch. This key requirement enables the decision of whether a given point resides inside or outside of the patch's area. This is achieved by evaluating the implicit form of all involved curves for a given coordinate, which either results in a negative or positive value, depending on which side of the curve the point. For points where all tested boundaries provide the same solution, a decision can be made to either fill a pixel at that coordinate or discard it.

We describe an implementation of a system that can process input in the form of SVG and generate CPatches from it. In a first step, we construct a graph representation from the input where start points, end points and intersections of the curves are embodied by nodes and the actual curves as edges.

The next steps in our preprocessing pipeline involve determining which side of each participating curve is to be filled by ray shooting (winding order test) and finding closed loops in the graph to remove them in an ear clipping fashion to form new sub-graphs. The sub-graphs are checked for intersection of implicit extensions of the bounding curves with the area of the patch and its bounding box. Several heuristics are needed to handle critical points, like implicit extensions reentering the bounding box of the patch or cases where the ear clipping runs into a dead end.

With the CPatch representation allowing for definite determination of inclusion or exclusion of points to be filled, we propose a hierarchical rasterization scheme and provide an efficient implementation based on the the GPU scheduling framework Whippletree.

We start at the coarsest level of a patch and test points of interest like corners of the patch's bounding box and extreme points of the involved curves against the interpolated values stemming from the implicit representation. The result of that test enables the decision of discarding the patch (result for all points is *outside*), further subdivision (some tests result in *inside*, some in *outside*), complete fill (all points report *inside*), or fine rasterization, if a certain level of subdivision has been reached. The decisions that require further processing will generate new work item for the scheduling system. For efficient blending, the processing of all final tiles is deferred to a second pass where the fine rasterizer performs the actual shading.

**Sub-pixel visibility sampling** is a method we proposed to mitigate visual artifacts we encountered in the context of decoupled rendering. When dividing a rendering pipeline into two separate parts for shading and display, where the shading happens in object space and the display stage is dependent on the scene's geometry, the visible primitives to be shaded must be determined beforehand (in contrast to

image-based rendering solutions where whole frames are provided to the display stage to warp to the current view point).

Not only missing shading information will cause disocclusion artifacts, but also triangles that were missed when doing the visibility sampling due to them falling through the sampling grid.

Our method consists of at least two render passes, where we establish a baseline in the first pass with standard sampling. The triangles are processed by a rendering pipeline that is set up to write the triangle's ID (a running index assigned when the meshes are loaded) to a texture at the coordinates where that triangle survives the depth test. In addition to that, the minimum and maximum depth of the primitive within the pixel's boundaries are computed and stored.

After the first pass, the triangles that also would have been detected by a standard sampling are already in the visible set. In the second pass, we enrich this set by running a rendering pipeline set up with conservative rasterization. This causes all pixel locations that are touched by a triangle to invoke a fragment shader in which we apply heuristics to decide whether to mark the triangle visible or not.

The proposed heuristics take geometric properties of the transformed triangles into account. We compare the minimum and maximum depth, the in-circle radius and the relative angle between the triangle in consideration and the sampled triangle from the first pass.

## 7.1 Findings and Gained Knowledge

The solutions investigated in this thesis follow the general idea that the input data set is transformed into an intermediary representation, which can be computed either offline or on-the-fly, but its memory footprint is small enough so it conveniently fits in GPU memory. This allows the actual geometry processing to operate on the intermediary representation in a massively parallel manner. The output of the geometry processing are trivial primitives, which can be fed into a standard rasterization engine. The core concept of this thesis is shaping the input data of a given problem to an intermediate form that is suitable for consumption by massively parallel processors to improve the efficiency in handling increasing amounts of geometry data used in state of the art computer graphics.

In Table 7.1, we display the case studies that have been presented to underline the effectiveness of our approach and how this core concept, the mapping of input to

an intermediary representation suitable for massively parallel real-time processing, is applied to them.

The generation of models with shape grammars is optimized by transforming the input description to the operator graph representation which can be analyzed to produce the resulting triangles as fast as possible.

Vector graphics are transformed to CPatches to remove ambiguities in the decision of whether an area of the illustration should be filled with color or not. Only by removing the ambiguities, which are caused by self intersection of the bounding curves, our hierarchical rasterizer can produce correct renderings quickly.

By identifying poorly conditioned triangles in 3D models with the help of conservative rasterization and filtering them according to geometric properties such as depth, size and relative position, we can sample triangles that would stay undetected otherwise and enrich a potentially visible set to prevent disocclusion artifacts in certain situations.

| **Input data set** | **Intermediary representation** | **Geometric processing** | **Rasterization data set** |
|---|---|---|---|
| Shape grammar | Operator graph partitioning | Operator graph scheduling | Temporary triangle set |
| SVG | CPatch | Hierarchical tiling | Simple tiles |
| Huge set of poorly conditioned triangles | G-Buffer (depth+id) | Conservative rasterization + depth heuristic | Potentially visible set |

Table 7.1: This matrix shows the mapping from problem (input data) to solution (rasterizable output) via an intermediate representation that is suitable for on-the-fly generation and rendering.

We show the effectiveness of our scheduling approach and argue other problems in the domain of geometry, are eligible for redesigning them with our massively parallel execution. We also document the pitfalls that may be encountered when mapping the described methods to the GPU. In this regard, it is not always a safe bet to try to increase parallelism. When the input in mass model production already provides enough work, gearing the derivation process towards a low frequency of context switches yields better results.

While the operator graph provides a high level view of the generation process of shape grammar derivation, it allows us to tailor the execution to both the GPU and the requirements that come with the task at hand. If a bulk of input axioms already

provides enough parallelism to fill the queues of the system, the execution paths can be fused to prevent unnecessary scheduling decisions. The downside to this is the long preprocessing time to generate and test the code for a specific case. So rather than aiding in the modeling and expressiveness of shape grammars, our findings will help in accelerating the generation process for production systems.

For our vector graphics rasterization efforts we show that a lot of unnecessary computation can be saved by giving the input generation more thought. Once a good subdivision into CPatches is found, the workload is perfectly tuned to be run on GPU. We compare to the state of the art, which is, in the case of NVPR, even able to benefit from hardware acceleration. Our approach outperforms other methods in many cases in terms of performance and quality and still performs well, if that is not the case. Furthermore, CPatches can be implemented in a slim way in terms of memory consumption which is not only favorable for GPU per se, but would also be very well suited for mobile devices with a limited amount of memory.

When implementing a decoupled rendering system, we found that sampling at output resolution can be insufficient to detect all triangles that are truly visible. While supersampling by simply increasing the resolution when constructing the visible set can mitigate the problem, the workload quickly becomes infeasible to handle. With the support for conservative rasterization in modern graphics hardware, it becomes much more efficient to run the detection quickly at lower resolution and filter out wrongly detected triangles afterwards.

## 7.2 Outlook

For geometry processing a lot of potential exists to improve processing efficiency and quality of rendered output by opening up the programmability of the graphics pipeline We also see a lot of opportunity for hardware implementation of new functional blocks to accelerate geometry processing.

In future research, the quest to parallelize algorithms in geometry processing goes on. Compression to stream-generated geometry from the GPU comes to mind. An online PVS algorithm is yet to be created. As decoupled rendering systems will put into practice forVR, these topics might gain attention from researchers as well as the graphics industry.

With our work on operator graphs we have paved the way for future work not only in procedural modeling, but also in code generation and execution optimization for GPU. Since many computing problems can be represented by graphs, our work may

help to spark the interest to adapt the operator graph paradigm to many other fields of computing. To make the search for optimal schedules more feasible, improved search heuristics will be needed to handle the long computation times it currently takes to tackle larger problems.

The vector graphics rasterization can be improved by further optimizing the rasterization pipeline. While an implementation in software helps in gaining insights to its internals and where to find potential for performance improvements, hardware support to rasterize curves is devisable. Also the patch generation lends itself to further investigation, not only to determine what resembles a good patch, but also to come up with further patching strategies to speed up the process. From an engineering point of view, CPatches would benefit from implementations in drawing libraries like Skia or Cairo. In addition, a new file format to store CPatches and filter modules for web servers to convert vector graphics, commonly used on websites, on the fly would allow thin clients to only do the rasterization.

Our novel visibility sampling approach will benefit from further research in terms of improved heuristics to discard false positives. Another item on the list of improvements is to leverage the capability of latest graphics hardware to run the pipeline simultaneously for several view points. Since our method already achieves good results at lowered resolutions, running the triangle detection in parallel for multiple view offsets could be implemented efficiently with the help of these new extensions.

# Bibliography


[1] Bryan D. Ackland and Neil H. Weste. "The Edge Flag Algorithm: A Fill Method for Raster Scan Displays". In: *IEEE Trans. Comput.* 30.1 (Jan. 1981), pp. 41–48. ISSN: 0018-9340 (page 14).

[2] Advanced-Micro-Devices. *Radeon's next-generation Vega architecture Technical Whitepaper*. 2017. URL: https://radeon.com/_downloads/vega-whitepaper-11.6.17.pdf (visited on 10/08/2018) (page 82).

[3] Magnus Andersson et al. "Adaptive texture space shading for stochastic rendering". In: *Computer Graphics Forum* 33.2 (May 2014), pp. 341–350. DOI: 10.1111/cgf.12303 (page 17).

[4] Dan Baker. *Object Space Lighting*. Talk at Game Developers Conference. 2016. URL: http://www.cogsci.rpi.edu/~destem/gamearch/gdc16/Object-Space-Lighting-Rev-21.pptx (page 17).

[5] Vineet Batra et al. "Accelerating Vector Graphics Rendering Using the Graphics Hardware Pipeline". In: *ACM Trans. Graph.* 34.4 (July 2015), 146:1–146:15. ISSN: 0730-0301 (pages 15, 54).

[6] B. Beneš et al. "Guided Procedural Modeling". In: *Comp. Graph. Forum* 30.2 (2011), pp. 325–334 (page 12).

[7] Jiří Bittner and Peter Wonka. "Visibility in Computer Graphics". In: *Environment and Planning B: Planning and Design* 30.5 (2003), pp. 729–755. DOI: 10.1068/b2957. eprint: https://doi.org/10.1068/b2957. URL: https://doi.org/10.1068/b2957 (pages 16, 92).

[8] Huw Bowles et al. “Iterative Image Warping”. In: *Computer Graphics Forum* 31.2pt1 (2012), pp. 237–246 (page 17).

[9] Christopher A. Burns, Kayvon Fatahalian, and William R. Mark. “A Lazy Object-space Shading Architecture with Decoupled Sampling”. In: *Proceedings HPG*. 2010, pp. 19–28 (page 17).

[10] Petrik Clarberg, Robert Toth, and Jacob Munkberg. “A sort-based deferred shading architecture for decoupled sampling”. In: *ACM Transactions on Graphics* 32.4 (July 2013) (page 17).

[11] Petrik Clarberg et al. “AMFS: Adaptive Multi-Frequency Shading for Future Graphics Processors”. In: *ACM Transactions on Graphics* 33.4 (July 2014), pp. 1–12 (page 17).

[12] Keenan Crane. *Spot 3D Digital Asset*. 2019. URL: `http://www.cs.cmu.edu/~kmcrane/Projects/ModelRepository/` (visited on 02/11/2019) (page 88).

[13] Crytek. *Sponza 3D Digital Asset*. 2018. URL: `https://www.cryengine.com/marketplace/sponza-sample-scene` (visited on 02/11/2019) (page 88).

[14] Paul de Faget De Casteljau. *Shape mathematics and CAD*. Vol. 2. Kogan Page, 1986 (page 68).

[15] Piotr Didyk et al. “Adaptive Image-space Stereo View Synthesis”. In: *15th International Workshop on Vision, Modeling and Visualization Workshop*. Siegen, Germany, 2010, pp. 299–306 (page 17).

[16] A. E. Fabris, L. Silva, and A. R. Forrest. “An efficient filling algorithm for non-simple closed curves using the point containment paradigm”. In: *Proceedings X Brazilian Symposium on Computer Graphics and Image Processing*. Oct. 1997, pp. 2–9 (page 14).

[17] Gerald Farin. *Curves and surfaces for computer-aided geometric design: a practical guide*. Academic Press, 1988 (page 56).

[18] Mark Finch, John Snyder, and Hugues Hoppe. “Freeform Vector Graphics with Controlled Thin-plate Splines”. In: *ACM Trans. Graph.* 30.6 (Dec. 2011), 166:1–166:10. ISSN: 0730-0301 (page 16).

[19] Francisco Ganacim et al. “Massively-parallel Vector Graphics”. In: *ACM Trans. Graph.* 33.6 (Nov. 2014), 229:1–229:14. ISSN: 0730-0301 (pages 16, 54).

[20] Google. *Skia Graphics Library*. https://skia.org/. 2018 (page 14).

[21] Paul Guerrero et al. “Learning Shape Placements by Example”. In: *ACM Trans. Graph.* 34.4 (July 2015), 108:1–108:13. ISSN: 0730-0301. DOI: `10.1145/2766933`. URL: `http://doi.acm.org/10.1145/2766933` (page 25).

[22] Simon Haegler et al. “Grammar-based Encoding of Facades”. In: *Comp. Graph. Forum* 29.4 (2010), pp. 1479–1487 (page 13).

[23] S Havemann. “Generative Mesh Modeling”. PhD Thesis. TU Braunschweig, 2005 (pages 12, 27, 39).

[24] Karl. E. Hillesland and J. C. Yang. “Texel Shading”. In: *Proceedings of the 37th Annual Conference of the European Association for Computer Graphics: Short Papers*. Lisbon, Portugal, 2016, pp. 73–76 (pages 17, 77).

[25] Intel. *Graphics API Performance Guide for Intel® Processor Graphics Gen9 Online Developer Documentation*. 2017. URL: `https://software.intel.com/en-us/documentation/graphics-api-performance-guide-for-intel-processor-graphics-gen9/conservative-rasterization` (visited on 10/08/2018) (page 82).

[26] Bernhard Kerbl et al. “Effective Static Bin Patterns for Sort-middle Rendering”. In: *Proceedings of High Performance Graphics*. HPG ’17. Los Angeles, California: ACM, 2017, 14:1–14:10. ISBN: 978-1-4503-5101-0 (page 65).

[27] Khronos-Group. *Vulkan Spec Changelog Technical Specification*. 2018. URL: `https://github.com/KhronosGroup/Vulkan-Docs/blob/master/ChangeLog.txt` (visited on 10/08/2018) (page 85).

[28] Khronos-Group. *Vulkan Specification Technical Specification*. 2018. URL: `https://www.khronos.org/registry/vulkan/specs/1.1/html/vkspec.html#commandbuffers` (visited on 10/08/2018) (page 87).

[29] Mark J. Kilgard and Jeff Bolz. “GPU-accelerated Path Rendering”. In: *ACM Trans. Graph.* 31.6 (Nov. 2012), 172:1–172:10. ISSN: 0730-0301 (pages 15, 54, 72).

[30] Yoshiyuki Kokojima et al. “Resolution Independent Rendering of Deformable Vector Objects Using Graphics Hardware”. In: *ACM SIGGRAPH 2006 Sketches*. SIGGRAPH ’06. Boston, Massachusetts: ACM, 2006. ISBN: 1-59593-364-6 (page 15).

[31] Lars Krecklau and Leif Kobbelt. “Procedural Modeling of Interconnected Structures”. In: *Comp. Graph. Forum* 30 (2 2011) (page 12).

[32] Lars Krecklau, Darko Pavic, and Leif Kobbelt. “Generalized Use of Non-Terminal Symbols for Procedural Modeling”. In: *Comp. Graph. Forum* 29 (8 2011), pp. 2291–2303 (page 12).

[33] Robert V Krejcie and Daryle W Morgan. “Determining sample size for research activities.” In: *Educ Psychol Meas* (1970) (page 47).

[34] P. Lacz and J.C. Hart. “Procedural Geometry Synthesis on the GPU”. In: *Workshop on General Purpose Computing on Graphics Processors*. 2004, pp. 23–23 (pages 12, 36, 51).

[35] Kyungmin Lee et al. “Outatime - Using speculation to enable low-latency continuous interaction for mobile cloud gaming”. In: *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. 2015 (page 17).

[36] Rui Li, Qiming Hou, and Kun Zhou. “Efficient GPU Path Rendering Using Scanline Rasterization”. In: *ACM Trans. Graph.* 35.6 (Nov. 2016), 228:1–228:12. ISSN: 0730-0301 (pages 14, 15, 54, 72).

[37] Yuanyuan Li et al. “Geometry Synthesis on Surfaces Using Field-Guided Shape Grammars”. In: *IEEE Trans. Visualization and Computer Graphics* 17.2 (2011), pp. 231–243 (page 12).

[38] Gábor Liktor and Carsten Dachsbacher. “Decoupled deferred shading for hardware rasterization”. In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM. 2012, pp. 143–150 (pages 17, 77).

[39] Markus Lipp, Peter Wonka, and Michael Wimmer. “Parallel Generation of Multiple L-systems”. In: *Computers & Graphics* 34.5 (2010), pp. 585–593. ISSN: 0097-8493 (pages 13, 26, 32, 36, 51).

[40] Charles Loop and Jim Blinn. “Resolution Independent Curve Rendering Using Programmable Graphics Hardware”. In: *ACM Trans. Graph.* 24.3 (July 2005), pp. 1000–1009. ISSN: 0730-0301 (pages 14, 15, 54–56).

[41] M. Magdics. “Real-time Generation of L-system Scene Models for Rendering and Interaction”. In: *Spring Conf. on Computer Graphics*. Budmerice, Slovakia: Comenius Univ., 2009, pp. 77–84 (page 13).

[42] Josiah Manson and Scott Schaefer. “Analytic Rasterization of Curves with Polynomial Filters”. In: *Computer Graphics Forum* 32.2pt4 (2013), pp. 499–507 (page 14).

[43] Kurt Akeley Mark Segal. *The OpenGLR©Graphics System:A Specification(Version 4.6 (Core Profile) - February 2, 2019)*. 2019. URL: `https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf` (visited on 03/05/2019) (pages 2, 4).

[44] William R. Mark, Leonard McMillan, and Gary Bishop. “Post-rendering 3D warping”. In: *Proceedings of the 1997 Symposium on Interactive 3D Graphics*. 1997 (page 17).

[45] Jean-Eudes Marvie et al. “Render-Time Procedural Per-Pixel Geometry Generation”. In: *Graphics Interface*. 2011, pp. 167–174 (page 13).

[46] Jean-Eudes Marvie et al. “GPU Shape Grammars”. In: *Comp. Graph. Forum* 31.7-1 (2012), pp. 2087–2095 (pages 13, 26, 36, 49–51).

[47] Gary H Meisters. “Polygons have ears”. In: *The American Mathematical Monthly* 82.6 (1975), pp. 648–651 (page 67).

[48] Joerg H. Mueller et al. “Shading Atlas Streaming”. In: *ACM Transactions on Graphics* 37.6 (Nov. 2018). DOI: 10.1145/3272127.3275087 (pages 77, 78, 85).

[49] Pascal Müller et al. “Procedural Modeling of Buildings”. In: *ACM Trans. Graph.* 25.3 (2006), pp. 614–623 (pages 12, 27, 38).

[50] Diego Nehab and Hugues Hoppe. “Random-access Rendering of General Vector Graphics”. In: *ACM Trans. Graph.* 27.5 (Dec. 2008), 135:1–135:10. ISSN: 0730-0301 (page 15).

[51] William M. Newman and Robert F. Sproull, eds. *Principles of Interactive Computer Graphics (2Nd Ed.)* New York, NY, USA: McGraw-Hill, Inc., 1979. ISBN: 0-07-046338-7 (page 14).

[52] Nvidia. *Nvidia Turing GPU Architecture*. 2018. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf (visited on 03/05/2019) (page 3).

[53] Marc Olano and Trey Greer. “Triangle Scan Conversion Using 2D Homogeneous Coordinates”. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*. HWWS ’97. Los Angeles, California, USA: ACM, 1997, pp. 89–95. ISBN: 0-89791-961-0 (page 58).

[54] Alexandrina Orzan et al. “Diffusion Curves: A Vector Representation for Smooth-shaded Images”. In: *ACM Trans. Graph.* 27.3 (Aug. 2008), 92:1–92:8. ISSN: 0730-0301 (page 16).

[55] Keith Packard, Carl Worth, and Behdad Esfahbod. *Cairo: A Vector Graphics Library*. https://www.cairographics.org/. 2018 (page 14).

[56] Evgueni Parilov and Denis Zorin. “Real-time Rendering of Textures with Feature Curves”. In: *ACM Trans. Graph.* 27.1 (Mar. 2008), 3:1–3:15. ISSN: 0730-0301 (page 15).

[57] Yoav I. H. Parish and Pascal Müller. “Procedural modeling of cities”. In: *Proc. SIGGRAPH 2001*. 2001, pp. 301–308 (page 12).

[58] G. Patow. “User-Friendly Graph Editing for Procedural Modeling of Buildings”. In: *Computer Graphics and Applications, IEEE* 32.2 (Mar. 2012), pp. 66–75. ISSN: 0272-1716. DOI: 10.1109/MCG.2010.104 (pages 28, 38).

[59] Przemyslaw Prusinkiewicz, Mark James, and Radomír Měch. “Synthetic Topiary”. In: *Proc. SIGGRAPH 94*. 1994, pp. 351–358 (page 12).

[60] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990 (pages 12, 27).

[61] Przemyslaw Prusinkiewicz et al. “The Use of Positional Information in the Modeling of Plants”. In: *Proc. SIGGRAPH 2001*. 2001, pp. 289–300 (page 12).

[62] Tim Purcell. “Fast tessellated rendering on Fermi GF100”. In: *High Performance Graphics Conf., Hot 3D presentation*. 2010 (page 65).

[63] Zheng Qin, Michael D. McCool, and Craig Kaplan. “Precise Vector Textures for Real-time 3D Rendering”. In: *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*. I3D ’08. Redwood City, California: ACM, 2008, pp. 199–206. ISBN: 978-1-59593-983-8 (page 15).

[64] Jonathan Ragan-Kelley et al. “Decoupled sampling for graphics pipelines”. In: *ACM Transactions on Graphics* 30.3 (May 2011), pp. 1–17. DOI: 10.1145/1966394.1966396 (pages 16, 17).

[65] Jonathan Ragan-Kelley et al. “Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines”. In: *ACM Trans. Graph.* 31.4 (July 2012), 32:1–32:12 (page 32).

[66] Bernhard Reinert et al. “Proxy-guided Image-based Rendering for Mobile Devices”. In: *Computer Graphics Forum* 35.7 (Oct. 2016), pp. 353–362. DOI: 10.1111/cgf.13032 (page 17).

[67] Daniel Ritchie et al. “Controlling procedural modeling programs with stochastically-ordered sequential Monte Carlo”. In: *ACM Transactions on Graphics (TOG)* 34.4 (2015), p. 105 (pages 49, 50).

[68] George Salmon. *A treatise on the higher plane curves: intended as a sequel to A treatise on conic sections*. Hodges, Foster, and Figgis, 1879 (page 55).

[69] Michael Schwarz and Pascal Müller. “Advanced Procedural Modeling of Architecture”. In: *ACM Trans. Graph.* 34.4 (July 2015), 107:1–107:12. ISSN: 0730-0301. DOI: 10.1145/2766956. URL: http://doi.acm.org/10.1145/2766956 (page 12).

[70] Thomas W Sederberg and Tomoyuki Nishita. “Curve intersection using Bézier clipping”. In: *Computer-Aided Design* 22.9 (1990), pp. 538–549 (page 66).

[71] Markus Steinberger et al. “On-the-fly generation and rendering of infinite cities on the GPU”. In: *Computer Graphics Forum* 33.2 (2014), pp. 105–114 (pages 6, 25).

[72] Markus Steinberger et al. “Parallel generation of architecture on the GPU”. In: *Computer Graphics Forum* 33.2 (2014), pp. 73–82 (pages 13, 26, 36, 41, 49–51).

[73] Markus Steinberger et al. “Whippletree: Task-based Scheduling of Dynamic Workloads on the GPU”. In: *ACM Trans. Graph.* 33.6 (2014), 228:1–228:11 (pages 21, 41).

[74] Markus Steinberger et al. “Whippletree: Task-based Scheduling of Dynamic Workloads on the GPU”. In: *ACM Trans. Graph.* 33.6 (Nov. 2014), 228:1–228:11. ISSN: 0730-0301 (page 63).

[75] G. Stiny. *Pictorial and Formal Aspects of Shape and Shape Grammars*. Birkhauser Verlag, 1975 (page 11).

[76] G. Stiny. “Spatial Relations and Grammars”. In: *Environment and Planning B* 9 (1982), pp. 313–314 (page 11).

[77] Jon Story. *Don’t be conservative with Conservative Rasterization Nvidia GameWorks Blog*. 2014. URL: `https://developer.nvidia.com/content/dont-be-conservative-conservative-rasterization` (visited on 10/08/2018) (page 82).

[78] Timothy Sun, Papoj Thamjaroenporn, and Changxi Zheng. “Fast Multipole Representation of Diffusion Curves and Points”. In: *ACM Trans. Graph.* 33.4 (July 2014), 53:1–53:12. ISSN: 0730-0301 (page 16).

[79] Gerry Sussman, Harold Abelson, and Julie Sussman. *Structure and interpretation of computer programs*. 1983 (page 28).

[80] Jerry O. Talton et al. “Metropolis Procedural Modeling”. In: *ACM Trans. Graph.* 30 (2011), 11:1–11:14 (page 25).

[81] Unity-Technologies. *Robot Lab 3D Digital Asset*. 2018. URL: `https://assetstore.unity.com/packages/essentials/tutorial-projects/robot-lab-unity-4x-7006` (visited on 02/11/2019) (page 88).

[82] Unity-Technologies. *Viking Village 3D Digital Asset*. 2018. URL: `https://assetstore.unity.com/packages/essentials/tutorial-projects/viking-village-29140` (visited on 10/08/2018) (page 88).

[83] William W Wadge and Edward A Ashcroft. *Lucid, the dataflow programming langugage1*. Academic Press, 1985 (page 27).

[84] Lvdi Wang et al. “Vector Solid Textures”. In: *ACM Trans. Graph.* 29.4 (July 2010), 86:1–86:8. ISSN: 0730-0301 (page 15).

[85] Yuxiang Wang et al. “Decoupled Coverage Anti-aliasing”. In: *Proceedings of the 7th Conference on High-Performance Graphics*. HPG ’15. Los Angeles, California: ACM, 2015, pp. 33–42. ISBN: 978-1-4503-3707-6. DOI: 10.1145/2790060.2790068. URL: http://doi.acm.org/10.1145/2790060.2790068 (page 16).

[86] John Warnock and Douglas K. Wyatt. “A Device Independent Graphics Imaging Model for Use with Raster Devices”. In: *SIGGRAPH Comput. Graph.* 16.3 (July 1982), pp. 313–319. ISSN: 0097-8930 (page 53).

[87] John G. Whitington. “Two Dimensional Hidden Surface Removal with Frame-to-frame Coherence”. In: *Proceedings of the 31st Spring Conference on Computer Graphics*. SCCG ’15. Smolenice, Slovakia: ACM, 2015, pp. 141–149. ISBN: 978-1-4503-3693-2 (page 14).

[88] Peter Wonka et al. “Instant Architecture”. In: *ACM Trans. Graph.* 22 (2003), pp. 669–677 (page 12).

[89] Chris Wylie et al. “Half-tone Perspective Drawings by Computer”. In: *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*. AFIPS ’67 (Fall). Anaheim, California: ACM, 1967, pp. 49–58 (page 14).

[90] Jason C. Yang et al. “Real-Time Concurrent Linked List Construction on the GPU”. In: *Computer Graphics Forum* 29.4 (2010), pp. 1297–1304 (page 63).

[91] T. Yang et al. “A Parallel Algorithm for Binary-Tree-Based String Rewriting in L-system”. In: *Proc. International Multi-symposiums of Computer and Computational Sciences*. 2007, pp. 245–252 (page 12).

[92] Jeong-Joon Yoo et al. “Tile-based Path Rendering for Mobile Device”. In: *SIGGRAPH Asia 2015 Mobile Graphics and Interactive Applications*. SA ’15. Kobe, Japan: ACM, 2015, 5:1–5:6. ISBN: 978-1-4503-3928-5 (page 15).