Marcel Pichler, BSc

# Streaming Geometry Processing in a High-Performance Software Rendering Pipeline

**MASTER'S THESIS**

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

**Graz University of Technology**

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Markus Steinberger

Institute of Computer Graphics and Vision
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Dieter Schmalstieg

Graz, September 2020

This document is set in Palatino, compiled with pdfLaTeX and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

# 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 master's thesis.

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

# Abstract

In recent years graphics processing units (GPUs) have become more and more programmable. At this point, they can be used as massively-parallel coprocessors to run arbitrary code. Real-time rendering pipelines are usually implemented in hardware to achieve the necessary performance and power efficiency to process the massive amount of data needed to render 3D scenes. With the ability to execute arbitrary programs on the GPU, software rendering pipelines emerged. These pipelines offer greater flexibility and compatibility than hardware renderers at the expense of reduced performance and power efficiency.

In this work, we show an implementation of a primitive tessellation stage in a software rendering engine. Tessellation allows subdividing primitives by creating a new mesh with smaller primitives filling the whole old primitive. Our tessellation algorithm adheres to the OpenGL specification and thus allows to create shader programs to influence the subdivision dynamically. It is possible to use triangles and quads as input primitive type as well as tessellated primitives. Our renderer also guarantees primitive ordering which is essential for many rendering techniques. The pipeline uses a persistent megakernel scheduling approach with dynamic load balancing to maximize GPU utilization. Although our implementation is less performant than a hardware renderer, we show that real-time rendering is possible for real-world scenes.

# Kurzfassung

In den letzten Jahren wurden Grafikprozessoren (GPUs) freier programmierbar. Sie können nun als massiv-parallele Coprozessoren verwendet werden, um beliebige Programme auszuführen. Echtzeit Grafikpipelines wurden zuerst direkt in Hardware implementiert, um die nötige Leistung und Effizienz zu erreichen, um die massiven Datenmengen für die Darstellung von 3D Szenen verarbeiten zu können. Mit der Möglichkeit beliebige Programme auf der GPU auszuführen, kamen auch Software-Grafikpipelines zum Vorschein. Diese Pipelines bieten mehr Flexibilität und Kompatibilität als Hardware-Pipelines auf Kosten von geringerer Leistung und Energieeffizienz.

In dieser Arbeit stellen wir eine Implementierung eines Tessellierungsalgorithmus in einer Software-Grafikpipeline vor. Tessellierung ermöglicht es Primitive zu unterteilen, indem neue Dreiecksnetze erstellt werden, die aus kleineren Polygonen bestehen welche das ursprüngliche Polygon vollständig ausfüllen. Unser Tessellierungsalgorithmus haltet sich an die OpenGL Spezifikation und erlaubt es daher Shader Programme zu erstellen, die die Unterteilung der Polygone dynamisch beeinflussen. Es können sowohl Dreiecke als auch Vierecke als Primitive verwendet werden. Unsere Pipeline garantiert die richtige Reihenfolge der Primitive, was für viele Renderverfahren essenziell ist. Es wird ein Persistent Megakernel Scheduling Konzept mit dynamischer Lastverteilung verwendet, um die Ausnutzung der GPU zu maximieren. Obwohl unsere Implementierung weniger Leistung als ein Hardware Renderer bietet zeigen wir, dass komplexe Szenen in Echtzeit gerendert werden können.

# Contents

# List of Figures

# 1. Introduction

Real-time rendering applications use hardware rendering pipelines to achieve high performance and power efficiency. They use modern application programming interfaces (APIs) such as Vulkan [Khr20] or Direct3D [Bly06] to interface with the hardware. As an example, Figure 1.1 [Wik19] shows the rendering pipeline of OpenGL [AS19], which consists of several programmable and fixed-function stages. These APIs allow programming graphics applications independent of the underlying hardware. Hardware rendering pipelines process the primitive data in multiple sequential stages. As many processing steps are implemented directly in hardware, only some of the stages are programmable with fixed-function stages in between. These programmable stages are further limiting the developer with predefined data structures for input and output as well as a single-thread program model. Due to the arrangement of the stages, the pipelined processing, and the programming through an API, adaption to special rendering techniques is usually complex and imposes a performance overhead. It is also impossible to add new features to the pipeline without changing the hardware. Moreover, the graphics pipeline evolved only slowly. The stages of the pipeline stayed the same, and over time only a few additional stages have been added.

The performance of graphics hardware continues to increase exponentially, and specialized interfaces like CUDA [NVI19] or OpenCL [SGS10] have been created to allow the graphics processing unit (GPU) to be operated in compute mode. In this mode, the GPU can be programmed as a massively-parallel general-purpose co-processor, allowing efficient manipulation of large blocks of data.

By using the GPU in compute mode, it is possible to deploy software rendering pipelines that sacrifice some performance and power efficiency to implement sophisticated graphics algorithms that would be difficult to deploy on a

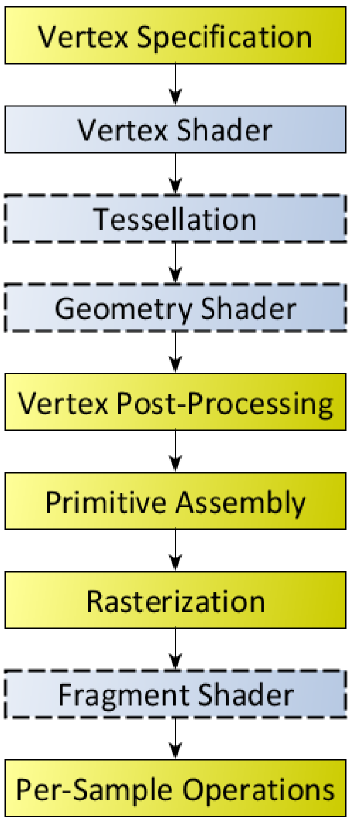


Figure 1.1.: Diagram of the OpenGL rendering pipeline, where the blue boxes represent programmable shader stages, and the yellow boxes are fixed-function stages. [Wik19]

traditional graphics pipeline. In this way, we can change the functionality of every stage and change the pipeline composition to create a different arrangement of stages. Moreover, we can completely change the functionality of fixed-function stages. As an example, the rasterization algorithm can be changed to do nonlinear rasterization [Llo+07], which can be useful for shadow mapping or similar techniques. We could also add features to the pipeline at a later point without the need to change the hardware.

This thesis contributes to the area of computer graphics. Specifically, we introduce a real-time software rendering pipeline with primitive tessellation. We describe the necessary processing structure and scheduling techniques to reach the required performance and present a primitive ordering approach, allowing the pipeline to use order dependent rendering techniques. To benchmark the implementation, we created multiple real-world demo scenes to ensure the coverage of different use cases.

For an introduction to the targeted software rendering pipeline and the adapted tessellation code, the previous and related work are shown first. The next chapters will give details to the tessellation procedure and primitive ordering. The implementation of the algorithm in a streaming pipeline is described afterwards in chapter 5. The results and performance evaluation will be shown in chapter 6.

## 1.1. Tessellation

One of the stages of a hardware rendering pipeline is called tessellation. It was the latest extension to the pipeline and is probably one of the most important techniques for all modern real-time graphics applications. Tessellation allows to dynamically increase the geometric resolution of primitives by dividing them into smaller primitives. An example of the tessellation of a single triangle is shown in Figure 1.2. In this way, we can decrease the quality of models and increase it dynamically for close by and large objects or objects in the center of attention of the viewer. As the divisions are done at runtime and only when intended and needed, the overall memory usage and bandwidth is reduced, while maintaining the same visual quality.

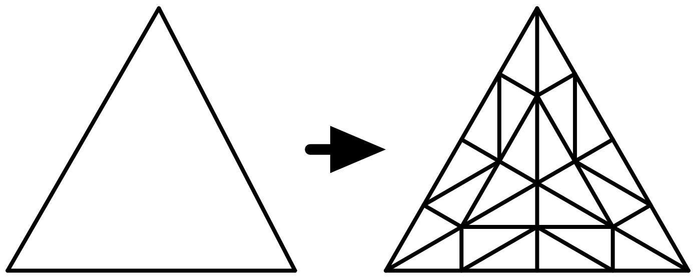


Figure 1.2.: Illustration of the subdivision of a triangle done by tessellation. A primitive is replaced with a mesh of primitives filling the whole initial primitive without overlaps or intersections.

There are several ways to split a primitive into smaller ones. The algorithm used in this thesis uses the technique specified by OpenGL, which uses a multi-stage approach. The first stage only determines the amount of subdivision to be done and may use a provided application-specific shader program. Then, the actual subdivision is performed, and custom attributes for all vertices can be computed in another application-specific shader program.

For this thesis, a tessellation stage, that was created for CUDA, is integrated into a software rendering pipeline. The software rendering pipeline has a streaming architecture, and therefore, the algorithm must be able to divide the workload and distribute it on the GPU.

## 1.2. Nvidia GPU

Graphics cards were initially designed for the sole purpose of rendering real-time high-resolution 2D and 3D scenes on a computer display, which requires very efficient manipulation of large blocks of data. As graphics applications became more and more versatile and applied more special rendering techniques, graphics pipelines together with the GPUs became more and more programmable. Therefore, GPUs evolved into highly parallel many-core systems, which can be used for general-purpose processing.

To make it easier for developers to create general-purpose programs for the GPU, Nvidia created CUDA [NVI19]. It is a programming API, allowing

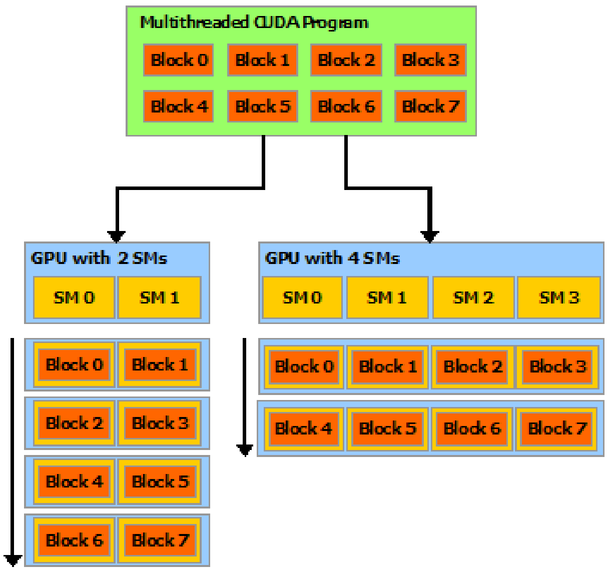


Figure 1.3.: Illustration of the distribution of blocks to Streaming Multiprocessors. While the programmer can choose the number of blocks, the number of Streaming Multiprocessors only depends on the GPU. A GPU with more multiprocessors will automatically execute the program faster. Reprinted from "CUDA C Programming Guide" by NVIDIA Corporation. Copyright 2007-2020 by NVIDIA Corporation.

developers to use C++ or Fortran as a high-level programming language for the GPU. Additionally, it provides an abstraction of the hardware threads and memory layout as a minimal set of language features. Therefore, programs can scale from simple main-stream GPUs to high-performance professional GPUs without any changes, which is illustrated in Figure 1.3 [NVI19].

The programming model allows defining C++ functions, called *kernels*, that execute code on multiple threads on the GPU in parallel. The exact number of threads which execute the kernel can be specified with the kernel launch. The threads are combined, forming a one-dimensional, two-dimensional or three-dimensional thread-block. The size of a thread-block is limited as all threads of the block share the limited resources of the same processor core. However, it is possible to combine multiple equally-shaped thread-blocks to a grid that executes a kernel. The hierarchical structure of thread-blocks is illustrated in Figure 1.4. A variable number of thread-blocks will be assigned

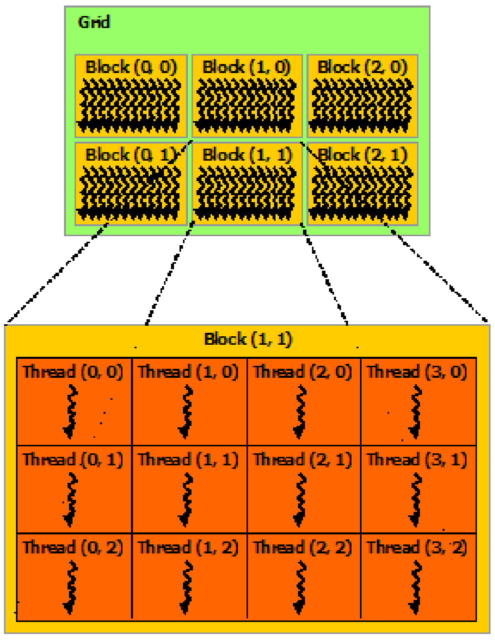


Figure 1.4.: Hierarchical organization of thread-blocks. Reprinted from "CUDA C Programming Guide" by NVIDIA Corporation. Copyright 2007-2020 by NVIDIA Corporation.

to each Streaming Multiprocessor (SM) of the GPU.

The SMs work on groups of 32 parallel threads called warps. All threads within the warp share the same program counter, start at the same program address and are scheduled together. To execute code branches with individual threads, the SM has an active mask to track the individual threads that take the branch and only threads marked as active execute an instruction. However, as all threads within a warp share the program counter, all inactive threads are taken along without executing these instructions which reduces the overall efficiency. Therefore, branching within warps should be kept to a minimum, and the workload should be optimized to fit within the boundaries of a warp.

This architecture is called SIMT (Single-Instruction, Multiple-Thread). It is an extension of the SIMD (Single-Instruction, Multiple-Data) architecture which applies a single instruction to all elements in a vector. In contrast to

SIMD machines, SIMT allows programmers to write thread-level parallel code for single threads as well as data-level parallel code for coordinated thread groups in the same way.

Threads may access data from different memory spaces which have different access times and visibilities (see Figure 1.5). Each thread has registers and local memory only accessible by this thread. All threads within the same thread-block can access the same shared memory space, which can act as a programmable cache and to share data within a thread-block. All threads can access global memory.

Shared memory is very fast, but each SM only has a limited amount available to split among all active thread-blocks. On the other hand, global memory has a very high access latency (approx. 100 times slower than shared memory), but there is much more global memory than shared memory. Local memory resides in global memory and therefore has the same access time. There are additional memory types for special use-cases which are not mentioned here.

Fatahalian and Houston [FH08] and Nickolls et al. [Nic+08] give a more detailed description of the GPU architecture and programming model.

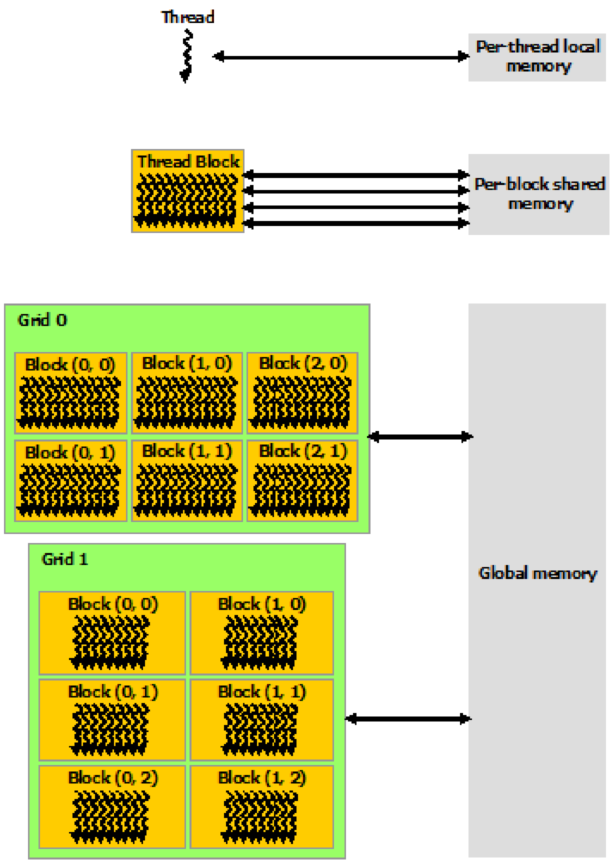


Figure 1.5.: Different memory spaces in CUDA with different visibilities. Reprinted from "CUDA C Programming Guide" by NVIDIA Corporation. Copyright 2007-2020 by NVIDIA Corporation.

# 2. Related Work

As this work describes the implementation of tessellation in streaming software rendering pipelines, we will first give a short overview of developments in this field. We will describe software rendering pipelines as well as tessellation algorithms and introduce the pipeline and algorithm used in the implementation of this thesis.

## 2.1. Software Rendering Pipeline

Rendering pipelines use various processing steps to transform lists of numbers (which represent primitives) into displayable images. The first rendering pipelines were implemented in software on the CPU, but specialized hardware was introduced to overcome the limited parallelism of the CPU and thus speed up rendering. Parallelism is critical for providing the necessary performance for real-time rendering as a large amount of data needs to be processed in a short time frame. Therefore, software rendering pipelines reemerged when the compute mode for GPUs became available and provided the necessary computing power.

One of the first software rendering engines on the GPU is Freepipe [Liu+10]. It implements a fully programmable rendering pipeline, which has problems with high depth complexity due to a non-linear sorting complexity, and it has a significant memory overhead. CudaRaster [LK11] implements a highly-optimized rasterizer on the GPU. The stages are executed sequentially, meaning all threads are working on the same stage at the same time. Piko [Pat+15] is a more recent implementation of a software rendering pipeline on the GPU using the same sequential execution of the stages.

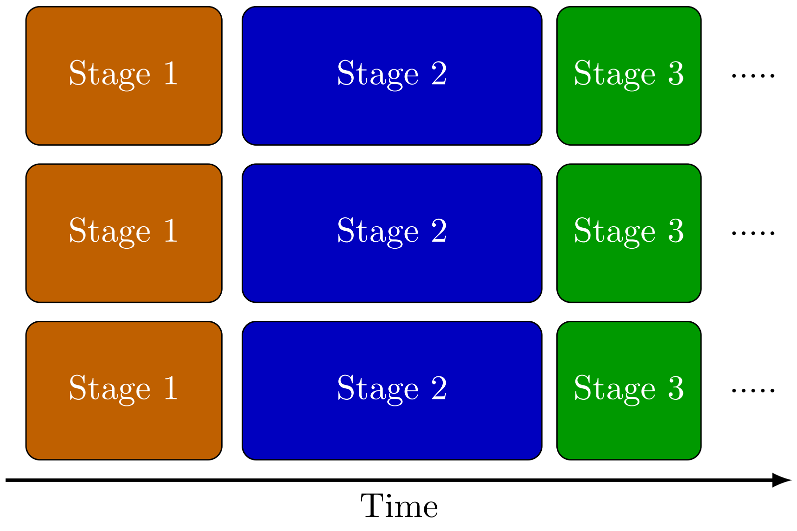


Figure 2.1.: Sequential execution of stages.

The most straightforward scheduling algorithm on the GPU is sequential execution of the stages or also called kernel-by-kernel and is illustrated in Figure 2.1. Consequently, all SMs of the GPU execute the same stage at the same time. However, this approach requires all data to be passed from one stage to the next through slow global memory and exploiting data locality through fast on-chip memory becomes impossible. Moreover, as the amount of data typically increases from one stage to the next, this also leads to a significant memory overhead. Another problem is the uneven distribution of the workload, as processors will be left to idle if a stage cannot fully occupy the GPU.

These restrictions can be avoided by using more complex scheduling algorithms. A simple yet powerful architecture uses *persistent threads* [AL09], which fill the whole GPU and fetch work from a global pool until it is empty. With further improvements, it is possible to create new work items dynamically and insert them into the pool [TPO10]. However, the global queue storing the work items for all tasks is a significant bottleneck as all threads need to access and modify it.

Steinberger et al. [Ste+14] created an implementation of a *persistent megakernel*, which uses persistent worker threads drawing work from task-specific queues.

Consequently, the bottleneck of a global queue which is accessed by all threads is eliminated. All threads execute the same kernel function in a loop which contains branches for each stage. An illustration of a megakernel architecture executing three stages is shown in Figure 2.2. The main limitation of the persistent-megakernel is that the resource configuration is determined by the most expensive stage and shared between all multiprocessors.

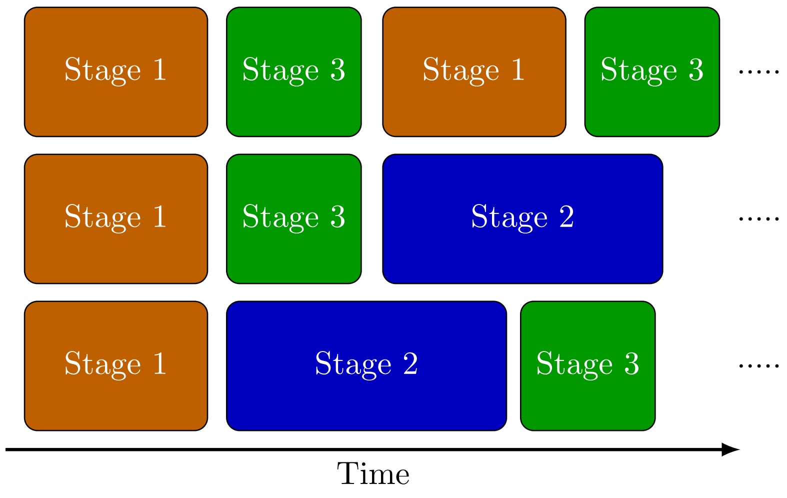


Figure 2.2.: Scheduling of the stages with a megakernel approach.

In this thesis, we will describe the integration of a tessellation stage into the pipeline introduced in "A high-performance software graphics pipeline architecture for the GPU" [Ken+18], which is called *CUDA Rendering Engine* (cuRE). It uses the above mentioned persistent-megakernel approach for scheduling and efficient work distribution. An overview of the pipeline stages is shown in Figure 2.3 [Ken+18].

The CUDA Rendering Engine is a real-time graphics pipeline implemented entirely in software to run on a modern GPU. Unlike previous approaches, it uses a streaming design for the geometry processing and rasterization with dynamic load balancing and can operate within bounded memory. Moreover, primitive order is sustained, and it delivers real-time rendering performance for real-world scenes.

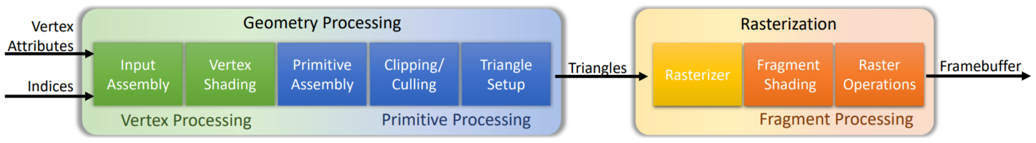


Figure 2.3.: Diagram of the cuRE graphics pipeline for indexed drawing. [Ken+18]

## 2.2. Tessellation

Before the introduction of tessellation and geometry shaders in hardware rendering pipelines, mesh refinement (i.e. increasing the detail and polygon count of models) could only be done on the CPU and introduced a bandwidth bottleneck on the graphics bus as well as a high load on the CPU. The size of the models which are transferred to the GPU increase substantially and complex computations on large amounts of data are performed on the CPU. To overcome these problems, Boubekeur and Schlick [BS05] introduced a mesh refinement technique in 2005 which uses the *vertex shader* to apply a generic refinement pattern to a polygon. In a later work, Boubekeur and Schlick [BS08] improved this technique and presented an adaptive mesh refinement process, which adjusts the refinement level based on the distance from the camera and is also implemented in the vertex shader. At this point, GPUs were already able to create new polygons dynamically with the geometry shading stage. However, due to the limitations of the *geometry shader*, only a few levels of refinement can be achieved.

Schwarz and Stamminger [SS09] presented a tessellation algorithm using the compute mode of a GPU to perform parallelized adaptive tessellation. In contrast to the previous methods using vertex shaders, this approach is faster for scenes with low tessellation levels and large numbers of primitives.

As dynamic mesh refinement methods became more and more popular, tessellation stages were included in the hardware rendering pipelines with DirectX11 [Mic15] and OpenGL4.0 [Khr10]. With these new additions, it was possible to perform adaptive dynamic mesh refinement inside the rendering pipeline. These new stages are simple to operate as it just requires to provide two new shader programs. The first shader specifies the amount of tessellation for each primitive and the second computes vertex attributes similar to a vertex shader. The first shader program can even be omitted if identical tessellation levels for all primitives suffice.

At this point, research work was focused on using the existing tessellation stages as efficient as possible instead of creating new implementations for mesh refinement. An overview of the recent work and challenges, as well as comparisons of the different techniques in the topic of hardware tessellation, is given by Nießner et al. [Nie+16].

### 2.2.1. Mesh Shaders

With Nvidia's Turing architecture [NVI18], a new programmable geometric shading pipeline was released in 2018. *Mesh shading* [Kub18] offers a new shader model for the vertex, tessellation and geometry shading stages of the graphics pipeline. With the compute programming model now in the graphics pipeline, threads can be used cooperatively to generate compact meshes (*meshlets*), which are consumed by the rasterizer. In this way, the entire vertex pipeline consists only of two stages: a Task shader followed by a Mesh shader; it is shown in Figure 2.4. This flexible approach allows efficient pre-culling, current tessellation scenarios and other level-of-detail (LOD) techniques as well as procedural generation. Thus, the approach reduces the workload on the CPU and allows the GPU to be used more parallel.

The new geometry processing pipeline contains only two shader stages. The mesh shader produces the primitives (triangles, lines, points) for the rasterizer using a cooperative thread model. Before the mesh shader in the pipeline is the optional task shader stage. It operates similar to the tessellation control stage, meaning it generates work dynamically, but uses a cooperative thread model and has user-defined input and output (meshlets).

A meshlet represents a variable number of vertices and primitives, which also may not be connected. Of course, there is a hardware limit to the maximum size of the meshlets, but their maximum size must also be specified in the shader code.

The mesh shading pipeline is showcased in the Asteroids demo, which uses extremely efficient culling and LOD techniques to reduce the number of

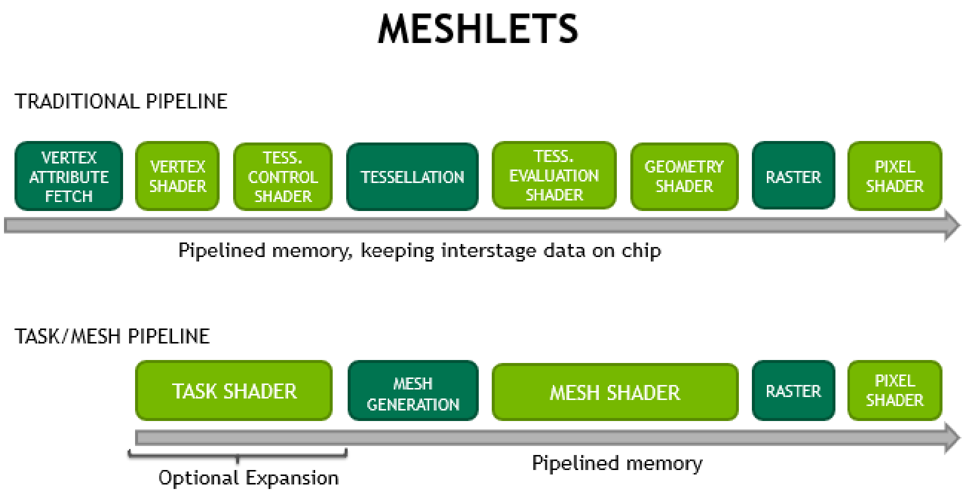


Figure 2.4.: Comparison of the traditional rendering pipeline with the new mesh shading pipeline. Reprinted from *Introduction to Turing Mesh Shaders — NVIDIA Developer Blog* by NVIDIA Corporation. Copyright 2018-2020 by NVIDIA Corporation.

rendered triangles by several magnitudes, keeping only those necessary to show a high level of image fidelity. The use of the mesh shading pipeline in this demo drastically reduces the workload of the GPU and CPU for the culling and LOD operations.

However, the integration of mesh shaders into a graphics application requires developers to refactor their geometry toolchain by separating geometry into compact meshes (optimally with maximal vertex re-use within the meshlets) and create own LOD algorithms instead of using the geometry and tessellation stages.

### 2.2.2. Used Algorithm

It makes sense to use an algorithm that is already implemented in CUDA and contains a work distribution, to include a tessellation stage into the software rendering pipeline as mentioned above. Consequently, we used the tessellation procedure described by Stadlbauer [Sta19].

His algorithm performs dynamic mesh refinement, as specified by OpenGL, for triangles and quads. The algorithm is organized in stages similar to OpenGL, which are executed consecutively. Between the stages, the algorithm returns to the CPU and performs dynamic memory allocations and the work distribution with prefix sums. Consequently, most parts of the algorithm must be adapted to work in a streaming software rendering pipeline, especially since we cannot return to the CPU between stages nor allocate memory of dynamic size.

# 3. Tessellation Procedure

This chapter will focus on the tessellation procedure, which divides primitives, in our case, only triangles and quads, into smaller sections. The generated primitives are always triangles to simplify the processing in later stages of the pipeline, especially in the rasterizer.

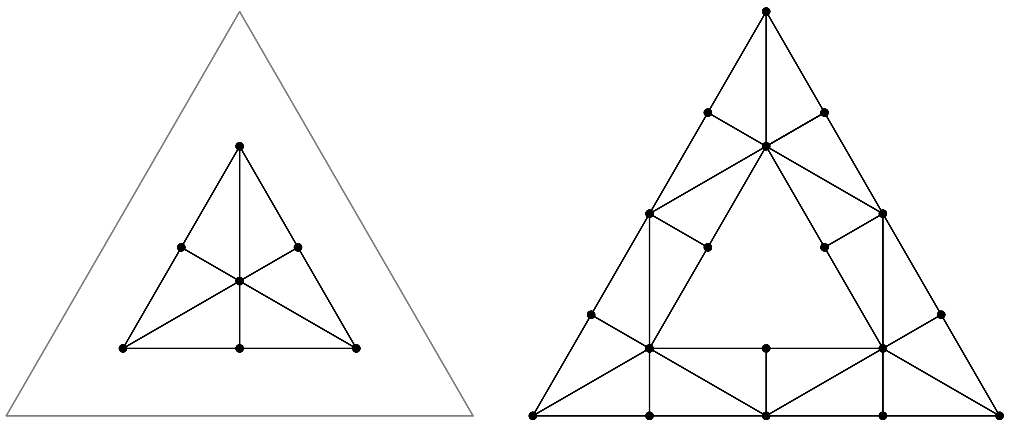
## 3.1. Existing Algorithm

We use an existing tessellation algorithm for CUDA, which was presented by Stadlbauer [Sta19]. The details of the algorithm are described in his work. Therefore we will only highlight the modifications and additions to use the algorithm in a streaming rendering pipeline in this chapter.

## 3.2. Primitive Sections

Tessellation allows each primitive to be subdivided into an arbitrary number of triangles (up to a specified maximum). The amount of subdivision is controlled with multiple tessellation levels, which specify how many segments an edge is split into. For triangles, there is only one inner tessellation level and three outer levels; one for each outer edge. For quads, there are two inner levels, one for the horizontal and one for the vertical tessellations and four outer levels; one for each outer edge. It is possible to displace the newly generated vertices by assigning them arbitrary positions in a shader program.

Each primitive is split into an inner and outer section, which is illustrated for a triangle in Figure 3.1. The inner section forms either a quad or a triangle, and



(a) Inner triangle section

(b) Outer triangle section

Figure 3.1.: Inner and outer sections of a triangle.

the outer section is a triangle strip connected to form a circle. Both sections have their individual tessellation levels allowing to create a mesh without cuts or holes in it. In this way, it is possible to control the geometric resolution with the inner tessellation levels and use the outer tessellation levels to specify the number of vertices on each outer edge. The outer primitive section then connects the inner section which has an arbitrary number of vertices (determined by the desired geometric resolution) on its outer edges to the outer edges of the patch with a fixed number of triangles (controlled by the neighbouring patches). This makes it possible to properly connect multiple patches, with each having different tessellation levels within the patch and thus different numbers of triangles, but the same number of vertices on the shared edges. In this way, we obtain a mesh without holes or cuts. Figure 3.2 shows how we can have different numbers of vertices on each outer edge with the outer subpatches connecting the outer edge to the inner subpatches.

## 3.3. Subpatches

As already mentioned, the primitives are split into subpatches for the tessellation, which is illustrated in Figure 3.2. The subpatches are pieces of the mesh with a specific maximum number of triangles and vertices to match the warp size of CUDA, which is 32. In this way, each thread can process one vertex and one triangle while a whole subpatch is always processed by the same warp and therefore at the same time. Moreover, we can exploit the most efficient synchronization and data sharing methods. The subpatches are also used to simplify the work distribution, as we only have to manage the subpatches instead of individual triangles. Same as with the primitive sections, we also have the distinction of inner subpatches and outer subpatches.

**Inner Subpatches** have the shape of squares or rectangles of up to 32 triangles. Due to the reuse of vertices, a full subpatch can have 32 triangles with only 25 vertices. Hence, each thread of the warp will create a triangle. However, not all threads will participate in vertex processing, which may add an overhead, especially in the execution of the tessellation evaluation shader. The number and size of the inner subpatches are only determined by the inner tessellation levels.

**Outer Subpatches**, on the other hand, form triangle strips, which have poor vertex reuse. Here, the subpatches have a maximum of 28 triangles and 32 vertices. It would be possible to form 30 triangles with 32 vertices in a triangle strip, but we also have to consider a special case in the triangle tessellation where the outer subpatch consists of two triangle strips. The number and size of the outer subpatches are determined by both the inner and outer tessellation levels.

## 3.4. Index Computations

In the algorithm described by Stadlbauer [Sta19], global indices for triangles and vertices are used. Consequently, global arrays (one for each primitive) of dynamic size are needed to store the vertices. In this way, the vertices only have to be calculated once and can be reused to form multiple triangles. However, this approach requires to allocate memory dynamically and compute prefix sums to index these arrays, which would add latency and reduce the performance. To avoid this, we use individual arrays for each subpatch. The size of each array is 32, which is equal to the maximum number of vertices in a subpatch. Therefore, we may waste memory if a subpatch is small, but there

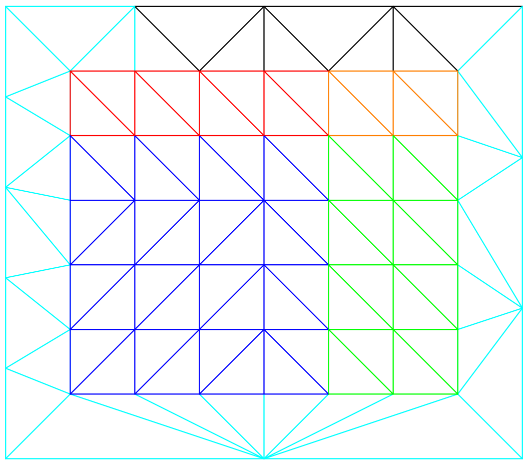


Figure 3.2.: Illustration of the division of a quad into subpatches. Each subpatch is marked with a different color and the only subpatches with the maximum size are the dark blue and light blue ones. The inner tessellation levels are 8 and 7, and the outer levels are 2, 3, 4 and 5 beginning with the bottom edge and going counterclockwise.

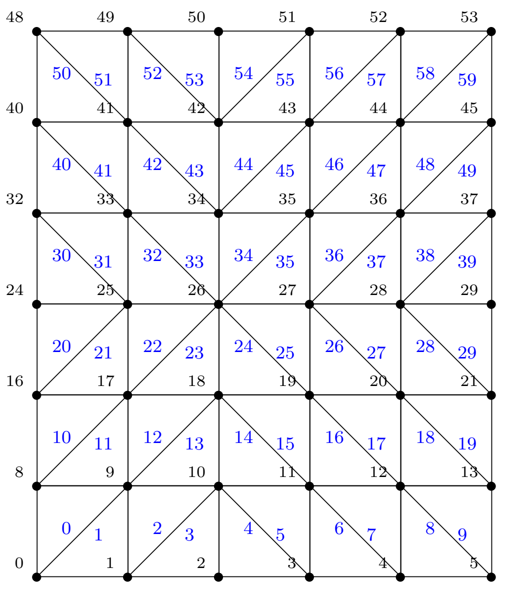
is no performance overhead for dynamic memory management. Moreover, the arrays can be stored in fast shared memory instead of global memory, and no prefix sums are required. On the other hand, we need to change the algorithms that compute and use the vertex indices to use the local indices in the range $[0, 31]$.

## 3.5. Inner Tessellation

The tessellation of the inner section differs for triangles and quads and is therefore described in separate subchapters.

### 3.5.1. Quad Inner Tessellation

The process for the inner tessellation of quads is illustrated in Figure 3.4. The inner tessellation of quads ignores the outer tessellation levels and


(a) Global enumeration of triangles (blue) and vertices (black) of the whole primitve.

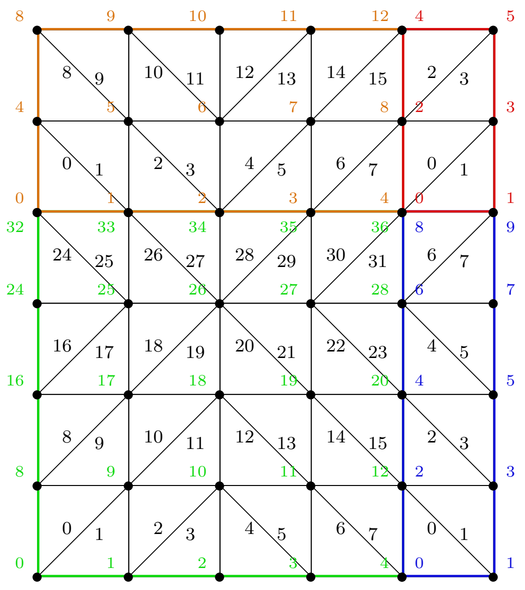

(b) Local enumeration of trianles and vertices within each subpatch. The local triangle ids are in black whereas the local vertex ids are colored with the subpatch color.

Figure 3.3.: Local and global enumeration of triangles and vertices.

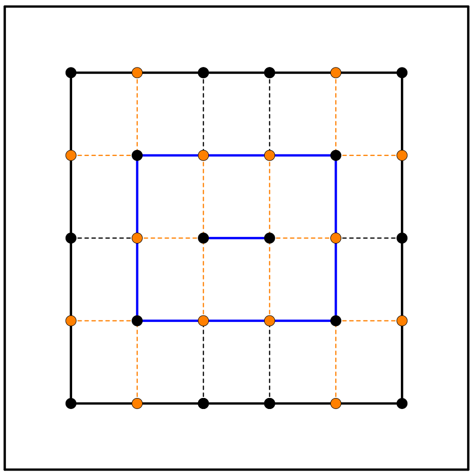


Figure 3.4.: Inner quad tessellation with tessellation levels inner0= 7 and inner1= 6.

applies the inner level 0 to both horizontal outer edges and level 1 to both vertical edges. Then we build a series of concentric rings (blue in the figure). The neighbouring vertices of each corner vertex of the outer ring are selected (orange). A vertex for the inner ring is created at the intersection of perpendicular lines (orange dashed) of the selected vertices. The edges of the inner ring are then subdivided at the intersection with perpendicular lines from the remaining vertices.

The process is repeated with the new ring to generate the remaining inner quads until any side of the new quad has either exactly one or two segments. In the example in the figure, the second ring has two segments on both vertical sides, so the algorithm only creates the last ring, which is a line with two vertices and then stops.

## Vertices

The vertex positions are computed as Barycentric coordinates as described in chapter 3.4.1 of “Streaming Primitive Tessellation for High-Performance Software Rendering Pipelines” [Sta19].

### Triangle and Vertex Indices

The triangles are indexed row-by-row starting with the edge that connects the first and second vertex of the original primitive as the first row. The indices of the vertices are local indices that are only unique within a subpatch and restart with every new subpatch. In this way, the indices are in the range $[0, 31]$ and can be used to index into subpatch local arrays. Figure 3.3 shows how the triangles and vertices of a patch are indexed.

The algorithm for the computation of the vertex indices that form each triangle is shown below. The calculation differs depending on the quadrant the triangle is in, to create symmetric patches.

**Algorithm 3.1** CalculateQuadInnerTriangle

```
1: $pointingDown \leftarrow idx \bmod 2$
2: $row \leftarrow \frac{threadIdx}{2 \cdot subpatchSize}$        ▷ $threadIdx$ is the index of the thread within a warp
3: $globalRow \leftarrow \frac{idx}{2 \cdot numColumns}$
4: $x \leftarrow \left(\frac{idx}{2} + globalRow\right) \bmod numRows$
5: $y \leftarrow \left(\frac{idx}{2} + globalRow\right) \cdot \frac{1}{numRows}$
6: $startIdx \leftarrow \frac{threadIdx}{2} + row$
        ▷ Different calculations depending on the quadrant ensures symmetric tessellation
7: if $\left(x < \frac{numRows}{2} \wedge y < \frac{numColumns}{2}\right) \vee \left(x \geq \frac{numRows}{2} \wedge y \geq \frac{numColumns}{2}\right)$ then
8:     $vertex.x \leftarrow startIdx$
9:     $vertex.y \leftarrow startIdx + (subpatchSize + 1) + 1 - pointingDown \cdot (subpatchSize + 1)$
10:    $vertex.z \leftarrow startIdx + (subpatchSize + 1) + pointingDown$
11: else
12:    $vertex.x \leftarrow startIdx + pointingDown$
13:    $vertex.y \leftarrow startIdx + 1 + pointingDown \cdot (subpatchSize + 1)$
14:    $vertex.z \leftarrow startIdx + (subpatchSize + 1)$
15: end if
```

## 3.5.2. Triangle Inner Tessellation

The inner tessellation of triangles is done in the same way as quads and is illustrated in Figure 3.5. Meaning, the outer tessellation levels are again ignored, and the inner tessellation level is applied to all edges. We build a series of concentric rings (blue in the figure) and select the two neighbouring

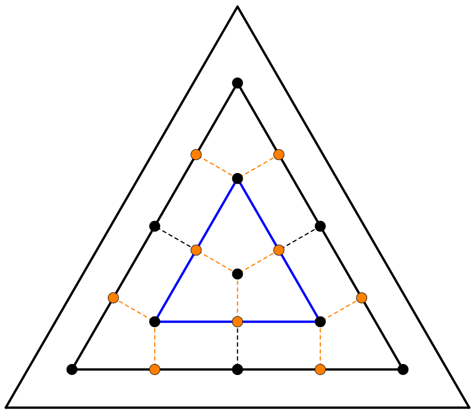


Figure 3.5.: Inner triangle tessellation.

vertices (orange) of each corner vertex. A vertex for the inner ring is created at the intersection of perpendicular lines (orange dashed) of the selected vertices. Perpendicular lines from the remaining vertices are used to subdivide the edges of the inner ring. This process is repeated until the inner ring is only a single vertex, or it has no subdivided edges.

## Vertices

The vertex positions are computed as Barycentric coordinates as described in chapter 3.5.1 of "Streaming Primitive Tessellation for High-Performance Software Rendering Pipelines" [Sta19].

## Triangle and Vertex Indices

We reuse the L-shaped grid as described by Stadlbauer [Sta19] for simpler and more efficient computations. However, we changed the vertex indexing from global indices to local indices which are only unique within the subpatch to use them as an index in local arrays. Therefore, the triangles and vertices are indexed as shown in Figure 3.3, but the patch is L-shaped instead of rectangular.

The algorithm that computes the indices of the vertices that form the triangles is shown below.

**Algorithm 3.2** CalculateTriangleInnerTriangle

```
1: $pointingDown \leftarrow idx \bmod 2$
2: $subpatchRow \leftarrow \frac{threadIdx}{subpatchSizeX*2}$    ▷ $threadIdx$ is the index of the thread within a warp.
3: $vStartLower \leftarrow \frac{threadIdx}{2} + subpatchRow$    ▷ $subpatchRow$ is the local row within the subpatch.
4: $vStartUpper \leftarrow \frac{threadIdx}{2} + subpatchRow + (subpatchSizeX + 1)$

5: $vertex.x \leftarrow vStartLower$
6: $vertex.y \leftarrow vStartLower + 1$
7: $vertex.z \leftarrow vStartUpper + \neg(row \geq numRings \vee column \geq numRings)$
8: if $\neg pointingDown$ then
9:     $vertex.x \leftarrow vStartUpper + 1$
10:    $vertex.y \leftarrow vStartUpper$
11:    $vertex.z \leftarrow vStartLower + \neg(row \geq numRings \vee column \geq numRings)$
12: end if
```

## 3.6. Outer Tessellation

The outer tessellation is done similar to chapter 3.6 of "Streaming Primitive Tessellation for High-Performance Software Rendering Pipelines" [Sta19], but we changed the computation of the triangle indices again, so they are local indices within each subpatch. Therefore, we use the indices from the `CalculateOuterTriangle` algorithm [Sta19], which returns the local indices within each side. We then add the vertices from the other sides processed which are in the same subpatch as offset and subtract the vertices from already processed vertices of the same side, i.e. if the subpatch does not start at the beginning of the side. We then obtain continuous vertex indices in the range $[0, 31]$.

An example is shown in Figure 3.6, where the different subpatches are marked with colors and the segments for the individual sides are marked

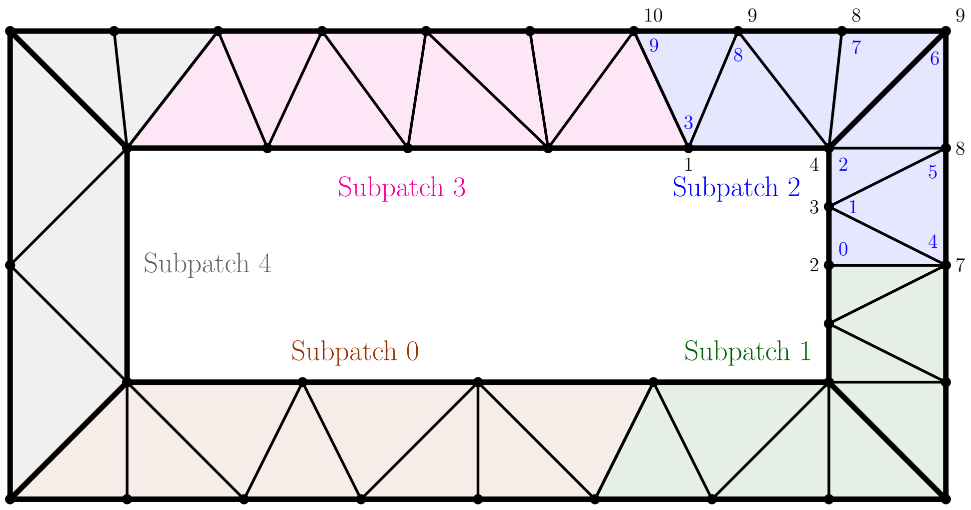


Figure 3.6.: Illustration of the outer tessellation of a quad. In this example, the size of the subpatches is reduced to 8 triangles instead of the normal size of 28 triangles to simplify the presentation. Each subpatch is filled with a different color, and the individual segments for each side of the outer section are marked with bold lines.

with bold lines. The vertex indices for subpatch 2 are marked as an example. The black numbers are the local indices within each side returned from the `CalculateOuterTriangle` function, and the blue numbers inside are the corrected ids.

# 4. Primitive Order

Hardware rendering pipelines such as OpenGL guarantee primitive ordering, which means, that primitives are drawn in the fragment buffer in the same order they were received in the pipeline. This guarantee is essential for order-dependent rendering techniques such as transparency. The primitives are ordered based on the order of their vertices in the vertex buffer. However, as rendering pipelines rely heavily on parallel processing, the order cannot be maintained throughout the pipeline, which is shown in Figure 4.1. Triangles start in the correct order with geometry processing, but take different amounts of time and thus end up in the wrong order. Therefore, primitives are processed out-of-order, and reordering is performed at some point before drawing the fragments. [MB05, chapter 8.5.5]

There are different possibilities to restore fragment order. A simple approach would be to keep the primitives ordered throughout the pipeline by performing a reordering step after each stage. This approach would reduce GPU utilization since we have to synchronize after each stage and add a large overhead due to the many sorting operations.

Another approach would buffer all fragments and then sort the individual fragments instead of primitives. In this way, there is no synchronization between the stages necessary, allowing for more efficient scheduling algorithms. However, we would have to store all possible fragments and sort them after all primitives have been processed, which adds a large memory and processing overhead.

The most efficient approach, which is also used in current hardware graphics pipelines reorders the primitives between geometry processing and rasterization stages. This approach requires the rasterizers to be split between disjoint regions of the screen. Thus, we remove the ordering constraint between rasterizers, since triangles processed by different rasterizers cannot overlap.

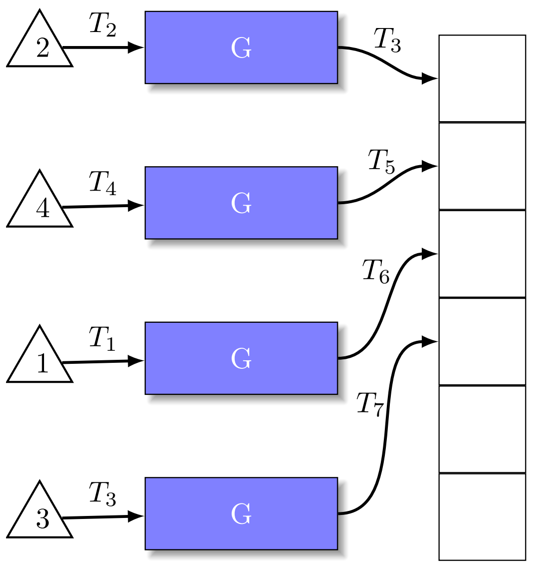


Figure 4.1.: Illustration of multiple triangles being processed in parallel in multiple instances of the geometry processing stages. The triangles take different amounts of time within the stage which is illustrated with the timestamp annotations $T_1$ to $T_7$ where $T_1 < T_2 < ... < T_7$.

However, triangles crossing the region boundaries also need to be processed by multiple rasterizers.

We use a similar approach, as described by Kenzel et al. [Ken+18] and collect the triangles in the *triangle buffer*, which is located right before the rasterizer. The triangle buffer caches the primitives to allow the rasterizer to work with full utilization and guarantee the correct order of primitives. The utilization of the rasterizer is provided by dequeuing primitives in batches of a specific minimum size. Before dequeuing the buffer is sorted, to ensure the order of the primitives. However, to prevent skipping primitives that did not yet arrive at the triangle buffer, a *progress queue* is used. Moreover, this queue also improves performance because it can track the number of primitives that are ready for sorting. Then we will only sort the triangle buffer if it contains enough primitives.

To distinguish between the primitives that enter the pipeline and the ones that have been subdivided by the tessellation stage, we will use the following terms in the remainder of this chapter: primitives describe patches entering the pipeline whereas triangles and quads describe subdivided primitives.

## 4.1. Progress Tracking

There are different approaches to implement a progress queue. If the pipeline does not use tessellation or geometry stages, no new primitives are generated in the pipeline. Therefore, each primitive can be indexed upon entering the pipeline, and a bitfield can be used to track the primitives. When a triangle is enqueued in the triangle buffer or discarded, the corresponding bit is set by the geometry processing thread. In this way, each triangle corresponds to a bit in the bitfield, where the position of the bit is equal to $primitiveID \bmod bitfieldSize$. Checking the number of triangles that can be sorted and rasterized is then simply done by finding the position of the first unset bit. This is always done by the rasterizer before sorting the input queue.

However, this approach cannot be used with tessellation or geometry stages, since arbitrary numbers of new primitives are generated dynamically. Consequently, bitfields of variable size would have to be allocated and managed dynamically, which adds significant overhead. A static version of the bitfield would have one bit for each triangle that could be created by the tessellation stage, which would need more than 6000 bits for each primitive, and therefore add an unmanageable memory overhead.

Instead of using a bitfield for the triangles, we implemented two different versions of the progress tracking with tessellation, which are described subsequently.

### 4.1.1. Counter for each primitive

The tessellation stage works on subpatches in a synchronized manner, meaning that all triangles within a single subpatch are processed at the same time. In consequence, we can track just the subpatches instead of individual triangles.

A simple way of tracking the subpatches is using a counter for each primitive that enters the pipeline. Thus, we use a large global ring buffer containing 8-bit counters, which are initially set to $-1$. At the end of the tessellation control stage, we can calculate the number of subpatches a primitive is split into and set the corresponding counter to that number. We can then reduce the counter by one every time the triangles of a subpatch are enqueued in the triangle buffer or discarded.

The maximum number of subpatches for a triangle is 208, while a quad can have up to 272 subpatches. To be able to use 8-bit counters for quad tessellation, at least two subpatches are combined to packets that are then used for the work distribution. In this way, the overall maximum number of subpatches for a primitive is 208. Therefore, each counter only needs to have 8-bit, thus adding an overhead only for very small tessellation levels. Checking for finished primitives is simply done by finding the first byte containing a nonzero bit, which is then the first unfinished primitive. However, this approach only allows for tracking the completion of the initial primitives that enter the pipeline and may be split into several thousands of triangles. This approach reduces the granularity of the rasterizer as the triangle batches that the rasterizer waits on may contain several thousand triangles.

### 4.1.2. Bitfield for subpatches

It is also possible to use a static bitfield in a large global ring buffer with one bit for each possible subpatch of the primitives entering the pipeline. Consequently, we need 208 bits for each primitive, which adds a significant memory overhead to the progress tracking. Moreover, we have to set all unused bits (i.e., $208 - numSubpatches$) at the end of the tessellation control stage for each primitive, so we get a continuous range of ones for finished primitives. Whenever the triangles of a subpatch are enqueued in the triangle buffer or discarded we simply set the corresponding bit in the bitfield.

In this way, we gain the ability to track individual subpatches instead of whole primitives. Thus, the rasterizer can wait on much smaller batches of triangles, which is beneficial if the subpatches and primitives are already processed partially in the correct order.

## 4.2. Sorting

Tracking the progress of primitives is only one part of guaranteeing primitive order. The primitives also need to be sorted before entering the rasterizer, which is done similar to [Ken+18]. The process is illustrated in Figure 4.2. First, we check the progress queue to ensure that enough primitives are ready to fill the rasterizer. Then we store the largest primitive id of the triangles that are ready as $p_{prog}$. The bits or counters for all primitives below $p_{prog}$ are reset and can be reused. We use a block-wide radix sort for reordering. As only a certain number of primitives (more specifically, only primitives with id smaller than $p_{prog}$) need to be delivered to the front of the queue, a moving sorting window is used. The window of size $w$ starts at the back of the queue and moves $^{w}/_{2}$ primitives towards the front in each step. When the window reached the front, we search for the largest id smaller than $p_{prog}$, which is the end of the part of the queue that is ready for consumption; it is marked as $p_{rdy}$. The sorting is repeated while the id of all sorted primitives is below $p_{prog}$. We only need to sort primitives with id less than $p_{prog}$ correctly and thus, we can remap the ids from the range $[p_{rdy}, p_{prog}]$ to $[0, p_{prog} - p_{rdy}]$. In this way, we limit the number of bits that radix sort has to consider.

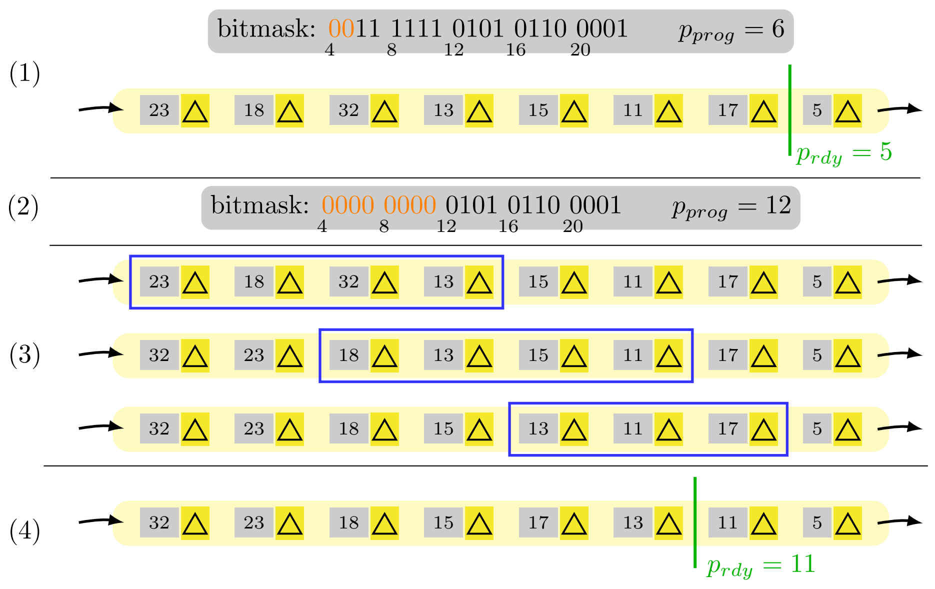


Figure 4.2.: Illustration of the sorting process. In this example, we use the progress tracking with a bitmask. (1) $p_{prog} = 6$ indicates that all subpatches up to id 6 have been accounted for. $p_{rdy}$ marks the end of the queue currently known to be complete and in order. (2) Since all bits from $p_{prog}$ up to 12 are now set, the block updates $p_{prog}$ and clears the bits. (3) It then sorts the queue by moving a sorting window from the back to the front. (4) The first two elements are ready to be processed and $p_{rdy}$ is updated.

# 5. Pipeline

This section describes the added pipeline stages for tessellation.

## 5.1. Tessellation Control Stage

The first stage for the tessellation algorithm is the tessellation control stage. This stage calls the tessellation control shader (TCS), which is application-specific and returns the tessellation levels for a primitive. Similar to OpenGL, the shader is executed once for every vertex of the primitive. A simple example is shown in listing 5.1. The TCS sets the tessellation levels and returns the spacing, which is used to convert the decimal tessellation levels to integers. The spacing also determines if and how the fractional part of the levels influences the locations of the generated vertices. With the levels, the stage also computes the number of subpatches the primitive will be split into.

**Listing 5.1:** Simple example of a tessellation control shader in cuRE. The `__constant__` variable is set once for the whole draw call and sets the tessellation levels for all primitives.

```
    __constant__ float tessLevel[6];

int tcs(int invocationID, int patch_vertices, const math::float4* pos,
        const VS_Out* in, float* tess_level_outer,
        float* tess_level_inner, TCS_Out* out)
{
    tess_level_inner[0] = tessLevel[0];
    tess_level_outer[0] = tessLevel[2];
    tess_level_outer[1] = tessLevel[3];
    tess_level_outer[2] = tessLevel[4];

    return 0;
}
```

### 5.1.1. Execution Context

The execution context for the tessellation control stage is *threads*, meaning that each thread is not aware of any other threads to exchange data with or synchronize. Each thread processes one primitive at a time.

### 5.1.2. Input

The input to this stage is the primitive generated after the execution of the vertex shader in the vertex processing stage, which can be a triangle or quad. The primitive data contains its primitive id as well as vertex positions, and all triangle attributes defined as output in the vertex shader.

### 5.1.3. Shader

First of all, the stage executes the TCS, which provides the data for all further processing steps. The TCS is executed once for each vertex of the primitive. As we have only one thread for the whole primitive, the TCS is executed in a loop. Listing 5.1 shows an example of a TCS for a triangle. The only input parameter that changes for the executions of vertices of the same primitive is the `invocationID`, which is just the loop counter. The parameters of the function are:

- **`invocationID`**: Index of the vertex in the patch, starting with 0
- **`patch_vertices`**: Number of vertices in the patch
- **`pos`**: Array containing the vertex positions
- **`in`**: Array containing the user-defined attributes from the vertex shader
- **`tess_level_outer`**: Output of the outer tessellation levels
- **`tess_level_inner`**: Output of the inner tessellation levels
- **`out`**: Output of user-defined attributes, which will be passed to the TES

The function returns the selected spacing for the vertex placement, where 0 means *equal spacing*, 1 sets *fractional even spacing* and 2 specifies *fractional odd spacing*. The spacing controls how the fractional tessellation levels are converted to integers and the position of the newly generated vertices.

After the shader has been executed for all vertices, we convert the fractional tessellation levels to integers and compute the number of subpatches which will be emitted for this primitive. We do not need to compute the number of triangles or vertices generated, since it is not needed in this or any later stages.

### 5.1.4. Subdivision

For efficient computations after the tessellation control stage, the workload is combined into subpatches. Each thread of the tessellation control stage creates between 0 and 208 subpatches for each primitive depending on its tessellation levels. The subpatches are smaller patches of the subdivided primitive that can be computed individually and have a maximum size that matches the warp size of CUDA. Consequently, the number of triangles and vertices is less or equal than 32, which is the warp size. In this way, a whole subpatch is always processed simultaneously. Therefore, we can manage subpatches instead of individual triangles in the work distribution stages, which reduces the overhead. The subpatches are constructed in a way to take advantage of the largest possible vertex reuse. The inner and outer sections are divided into individual subpatches, because of a different processing procedure which results from the different layout.

The inner subpatches are squares or rectangles. This subdivision scheme is easy to compute and results in an excellent triangle to vertex ratio. At the maximum size, the number of triangles is the limiting factor, as it is possible to reach 32 triangles with only 25 vertices.

The outer sections are triangle strips, and therefore the subpatches are also just triangle strips. The size is again chosen to match the warp size. Therefore, the maximum size is limited by the number of vertices as it is possible to build a triangle strip with 30 triangles using 32 vertices. However, there is a special case to be considered for triangles, which limits the size of outer subpatches to 28 triangles with 32 vertices.

### 5.1.5. Subpatch Redistribution

For the redistribution of the subpatches, they need to be buffered in queues. Therefore, each thread in the tessellation control stage adds all generated subptaches to a subpatch buffer. There are multiple buffers to reduce the work distribution overhead. Each multiprocessor has a buffer in shared memory which is implemented as a stack. Shared memory is only visible within the same multiprocessor, which is why each multiprocessor has its own buffer where only the threads of this multiprocessor will add subpatches. However, the amount of available shared memory is quite small. In consequence, not all subpatches will fit into the shared memory buffers, especially when using large tessellation levels. The remaining subpatches will be added to a queue in global memory, which threads of all multiprocessors can access. This queue has much larger access times and must be handled with atomic operations to avoid data races.

We can reduce the overhead for the work distribution further by combining multiple subpatches of the same primitive into a package because except for the subpatch id all subpatches of a primitive have identical data. In this way, we also reduce the data passed between the stages by a factor of the number of combined subpatches because the data only needs to be transferred once per package instead of once for each subpatch. This approach is implemented by emitting the combined subpatches in the tessellation control stage and repeating the execution of the subpatches in the tessellation evaluation stage with the different subpatch ids.

## 5.2. Tessellation Evaluation Stage

The Tessellation Evaluation Stage computes the Barycentric coordinates of the vertices, creates the triangle layout and executes the Tessellation Evaluation Shader (TES) which is application-specific and computes per-vertex attributes.

### 5.2.1. Execution Context

The execution context for the tessellation evaluation stage is *blocks*, meaning that the threads within a block can exchange data and synchronize with all other threads of the block. Each warp (consisting of 32 threads) within a block processes one subpatch. Therefore, the block dequeues as many subpatches at a time as it has warps. Since the block has access to a local queue in fast shared memory and a global queue, it will prioritize pulling elements from the local queue, to free up space in the faster memory first.

### 5.2.2. Input

The input to this stage are the subpatches or combined subpatches generated in the tessellation control stage. The subpatch contains the primitive id and subpatch id, the number of subpatches of the primitive, the fractional and integer tessellation levels and the attributes which are defined as outputs in the tessellation control shader.

The stage contains an input processing step which is implemented as a wrapper, to handle the packages of combined subpatches. This wrapper executes the stage for each subpatch in the package after computing the correct subpatch id. This is done by iterating over the number of subpatches in the package and multiplying the subpatch package id with the number of subpatches in the package and then adding the current loop counter.

### 5.2.3. Tessellator

The first processing step in this stage is the computation of an internal vertex id, which determines if a thread participates in vertex processing and if so, which vertex it processes. Then the Barycentric coordinates for each vertex in the subpatch are calculated, and the TES is executed. The vertex position returned by the shader is stored in shared memory so it can be reused by the other threads processing the same subpatch. In this way, vertices can be reused for multiple triangles, and each vertex only has to be processed once.

The last step is the creation of triangles using all vertices and filling the whole subpatch.

**Listing 5.2:** Simple example of a tessellation evaluation shader in cuRE. We only interpolate the vertex position and apply the view and projection transformations in this example.

```
math::float4 tes(const math::float3& v1, const math::float3& v2,
                 const math::float3& v3, const math::float3& uvw,
                 const TCS_Out* in, TES_Out& out)
{
   math::float3 coords;
   coords.x = v1.x * uvw.x + v2.x * uvw.y + v3.x * uvw.z;
   coords.y = v1.y * uvw.x + v2.y * uvw.y + v3.y * uvw.z;
   coords.z = v1.z * uvw.x + v2.z * uvw.y + v3.z * uvw.z;

   return camera.PV * math::float4{coords, 1.0f};
}
```

### Shader

The TES is executed once for each existing and newly generated vertex of the subpatch. It is executed in parallel for all vertices since we have one thread for each vertex. Listing 5.2 shows an example of a simple TES for a triangle. The parameters of the function are:

- **`v1`-`v3`**: Positions of the original vertices.
- **`uvw`**: Barycentric coordinates of the vertex within the original triangle.
- **`in`**: Array containing the user-defined attributes passed from the TCS.
- **`out`**: Output of user-defined attributes, which will be passed to the fragment shader.

The function returns the screen-space coordinates for the vertex.

The stage outputs between 1 and 32 triangles for each subpatch, meaning that each thread emits zero or one triangle. The generated triangles are directly forwarded to the primitive processing stage, which performs clipping and culling and computes values needed for the rasterizer such as the edges or the screen-space bounding box.

# 6. Implementation

This chapter will focus on the implementation of the algorithm described in the previous chapters. First, we will describe the framework used to develop the algorithm as well as the demos to test the implementation. The next sections will describe the implementations in OpenGL and cuRE, respectively, in more detail.

## 6.1. Framework

This section will describe the framework used to develop and test the pipelines. The framework uses C++templates to adapt to different hardware and pipeline configurations without influencing the runtime computations because in C++templates are evaluated and instantiated at compile time.

### 6.1.1. Testbed

The *Testbed* allows comparing different rendering engines in various scenes in an easy way. The renderer can be changed at runtime, and it has a navigator to traverse the scene as desired. Each renderer is provided with the same input models and render targets, and the frame time is measured. It allows to register keyboard inputs and forward them to the active renderer. Moreover, it can store the frame times and other events in a CSV file for profiling. We also record the number of clock cycles spent in each individual stage.

### 6.1.2. cuRE

*cuRE* contains all the components necessary to build the stages and work distribution of a rendering pipeline. Moreover, it provides an easy to use interface to specify the pipeline and data exchange interface between the stages. Different stream buffers and schedulers are available to control the work distribution. Shared memory can be specified for each stage individually and independent from the execution context. The execution context can be chosen individually for each stage and changed within the stage or between stages in the work distribution.

### 6.1.3. Rasterpipe

The *Rasterpipe* defines the rendering pipeline itself as well as the specification of all the stages. Some specialized work distribution stages that require structures and concepts from the rasterpipe are also defined there. The `pipeline.h` file contains the complete pipeline composition and configurations for the individual stages. Following parameters can be configured in this file:

- **`TESSELLATION`**: Enables or disables the tessellation stages.
- **`USE_PRIMITIVE_ORDER`**: Specifies if primitive order shall be enforced or not.
- **`USE_FRAGMENT_SHADER`**: Determines if the color is computed by a provided fragment shader or by the primitive id.
- **`USE_RANDOM_TESS_LEVEL`**: Can be set to bypass the tessellation control shader and compute the tessellation level based on the primitive id as $\textit{primitiveID} \mod 64 + 1$.
- **`SUBPATCHES_PER_THREAD`**: Sets the number of subpatches, which are combined into a package for the work distribution. The tessellation evaluation stage then iterates over the actual subpatches within this package. Increasing this number will reduce the cost for the work distribution, but may lead to an unbalanced utilization for small tessellation levels.

- `PROGRESS_TRACKING`: Select the used progress tracking approach for the primitive order. A value of 0 uses the version described in chapter 4.1.1 and a value of 1 the version of chapter 4.1.2.

The stages are defined in multiple files where each stage is an individual class. Therefore, the stages could be developed separately, if the interfaces between the stages matched.

### 6.1.4. Demos

Several test cases, which are described in the following sections, were created to check the correctness and compare the performance of the software rendering engine with OpenGL.

All demos have the same rather simple fragment shader, which computes a diffuse reflection by performing Lambertian shading. The light is static with a position far up the z-axis. The renderer can be switched to wireframe mode, to see the individual triangles and check the resolution of the mesh.

#### Simple Demo

This test applies uniform tessellation levels to provided models. More precisely, all primitives have the same tessellation levels. There is no displacement applied to the new vertices, which results in a flat tessellation, which is not noticeable without the wireframe mode. It is possible to change the tessellation levels and spacing mode at runtime with keyboard inputs or load them from a config file. Moreover, it is also possible to change between triangle and quad rendering, if the model specifies both. As a complete uniform tessellation is a rather unusual case, this test is mainly designed to check the basic functionality of the tessellation algorithm and model loading.

#### Terrain

A procedurally generated mountain range as shown in Figure 6.1 is rendered in the terrain demo. The input to the pipeline is a plane in the XY-plane

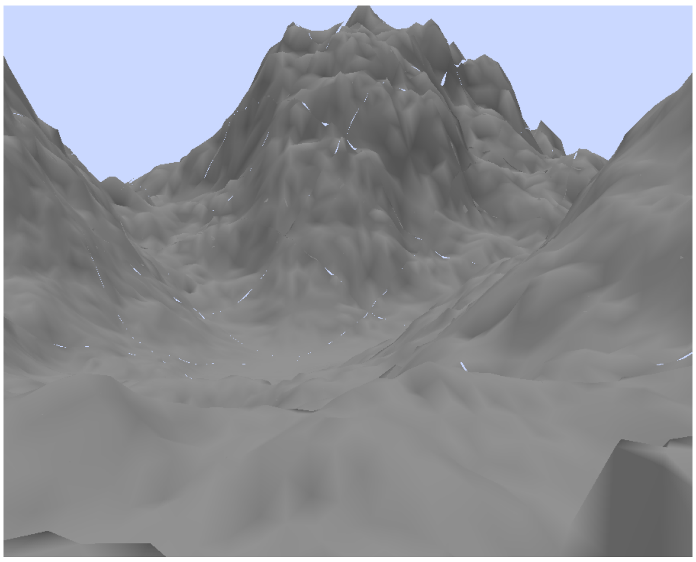


Figure 6.1.: Screenshot of the terrain demo rendered with cuRE.

composed of quads with a coarse resolution. As each quad can be divided into more than 8000 triangles with tessellation, it would suffice to have only 250 quads on the screen to create pixel-sized triangles for a Full HD screen. The user can modify the size of the terrain as well as the resolution of the coarse input mesh either at runtime with keyboard inputs or load the parameters from a config file. The geometric resolution of the plane is then increased dynamically with tessellation based on the distance to the camera to create triangles with approximately the same size on the screen. It is essential that the following computations are performed on the already displaced vertices, since the vertical offset may significantly affect the distance to the camera. The tessellation levels are computed by encapsulating each edge in a sphere (so the sphere has the same diameter as the edge length). Then the sphere is projected into screen-space, which gives us the projected edge length in pixels which is equivalent to the screen-space diameter of the sphere. This length is used to compute the tessellation level for the edge. In this way, we can render as few triangles as possible while maintaining high visual quality and good geometric detail. The targeted triangle size can be tuned (at runtime or in the config file) to achieve the best performance/detail trade-off.

The height of the terrain is computed procedurally using noise as displacement in the y-axis. The noise is generated with a fractal Brownian motion (fBm), which is a popular method for mixing samples obtained from Perlin noise to create more pleasing results. We can achieve this by taking several noise samples with different frequencies and summing them together. The samples with increasing frequency are scaled-down more and more before the summation. The sum is then returned as a noise sample and in our case used as height value of the terrain.

### PN Triangles

This demo uses the point-normal triangle algorithm as described by Vlachos et al. [Vla+01] to calculate a cubic Bézier triangle from the vertex positions and normals of a regular flat triangle. In this way, we can smooth out a triangle mesh without any additional information like displacement maps. A Bézier triangle is a smooth and continuous surface described by a cubic polynomial function.

The construction of the Bézier triangle is done by calculating the positions of 10 control points $\mathbf{b}_{ijk}$ with the condition $i + j + k = 3$. The vertex positions of the triangle are given as $\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$ and the normals at the vertices are $\mathbf{N}_1, \mathbf{N}_2, \mathbf{N}_3$. Three of the control points are just the positions of the original vertices and remain unchanged:

$$
\begin{aligned}
\mathbf{b}_{300} &= \mathbf{P}_1 \\
\mathbf{b}_{030} &= \mathbf{P}_2 \\
\mathbf{b}_{003} &= \mathbf{P}_3
\end{aligned}
$$

Six control points are related to the edges of the original triangle and are calculated as follows:

$$
\begin{aligned}
\mathbf{b}_{210} &= \frac{1}{3}\left(2\mathbf{P}_1 + \mathbf{P}_2 - \omega_{12}\mathbf{N}_1\right) \\
\mathbf{b}_{120} &= \frac{1}{3}(2\mathbf{P}_2 + \mathbf{P}_1 - \omega_{21}\mathbf{N}_2) \\
\mathbf{b}_{021} &= \frac{1}{3}(2\mathbf{P}_2 + \mathbf{P}_3 - \omega_{23}\mathbf{N}_2) \\
\mathbf{b}_{012} &= \frac{1}{3}(2\mathbf{P}_3 + \mathbf{P}_2 - \omega_{32}\mathbf{N}_3) \\
\mathbf{b}_{102} &= \frac{1}{3}(2\mathbf{P}_3 + \mathbf{P}_1 - \omega_{31}\mathbf{N}_3) \\
\mathbf{b}_{201} &= \frac{1}{3}(2\mathbf{P}_1 + \mathbf{P}_3 - \omega_{13}\mathbf{N}_1) \quad \text{with } w_{ij} = (\mathbf{P}_j - \mathbf{P}_i) \cdot \mathbf{N}_i
\end{aligned}
$$

The remaining control point is in the interior of the triangle and computes to

$$
\begin{aligned}
\mathbf{b}_{111} &= \mathbf{E} + \frac{1}{2}(\mathbf{E} - \mathbf{V}) \quad \text{with} \\
\mathbf{E} &= \frac{1}{6}(\mathbf{b}_{210} + \mathbf{b}_{120} + \mathbf{b}_{021} + \mathbf{b}_{012} + \mathbf{b}_{102} + \mathbf{b}_{201}) \quad \text{and} \\
\mathbf{V} &= \frac{1}{3}(\mathbf{P}_1 + \mathbf{P}_2 + \mathbf{P}_3)
\end{aligned}
$$

The position of any point **b** in this Bézier triangle can then be computed by interpolating between the control points with the Barycentric coordinates $\begin{bmatrix} u & v & w \end{bmatrix}^T$:

$$
\mathbf{b}(u,v,w) = \sum_{i+j+k=3} b_{ijk} \frac{3!}{i!j!k!} u^i v^j w^k \tag{6.1}
$$

The vertex positions of the tessellated triangles can now be calculated with equation 6.1.

The vertex normals can be either a linear interpolation of the original normals or a quadratic function of the original positions and normals. In this demo, we choose a quadratic function for the normals, as linearly interpolated normals will ignore inflections. Similar to the calculations for the positions, we also need control points for the normals. Three control points are again at the original vertices and remain unchanged:

$$\begin{aligned}\mathbf{n}_{200} &= \mathbf{N}_1\\ \mathbf{n}_{020} &= \mathbf{N}_2\\ \mathbf{n}_{002} &= \mathbf{N}_3\end{aligned}$$

Three additional control points are related to the edges and defined as follows:

$$\begin{aligned} v_{ij} &= 2 \cdot \frac{(\mathbf{P}_j - \mathbf{P}_i) \cdot (\mathbf{N}_i + \mathbf{N}_j)}{(\mathbf{P}_j - \mathbf{P}_i) \cdot (\mathbf{P}_j - \mathbf{P}_i)}\\ \mathbf{n}_{110} = \frac{\mathbf{h}_{110}}{||\mathbf{h}_{110}||}, \quad h_{110} &= \mathbf{N}_1 + \mathbf{N}_2 - v_{12}(\mathbf{P}_2 - \mathbf{P}_1)\\ \mathbf{n}_{011} = \frac{\mathbf{h}_{011}}{||\mathbf{h}_{011}||}, \quad h_{011} &= \mathbf{N}_2 + \mathbf{N}_3 - v_{23}(\mathbf{P}_3 - \mathbf{P}_2)\\ \mathbf{n}_{101} = \frac{\mathbf{h}_{101}}{||\mathbf{h}_{101}||}, \quad h_{101} &= \mathbf{N}_3 + \mathbf{N}_1 - v_{31}(\mathbf{P}_1 - \mathbf{P}_3)\end{aligned}$$

The normal vector of each vertex of the tessellated triangle can then be calculated as a linear interpolation of the control points with the Barycentric coordinates:

$$\mathbf{n}(u, v, w) = \sum_{i+j+k=2} \mathbf{n}_{ijk} u^i v^j w^k$$

The above-described calculations for the vertex positions and normals are performed in the tessellation evaluation shader.

The calculation of the tessellation levels uses the orientation of the triangle relative to the camera. The tessellation level is minimal for triangles which are perpendicular to the camera and maximal for triangles parallel to the view direction. In this way, the contour of the object appears even smoother, while the geometric resolution is not unnecessarily increased where it is not needed. An example of the adaptive tessellation levels is shown in Figure 6.2.

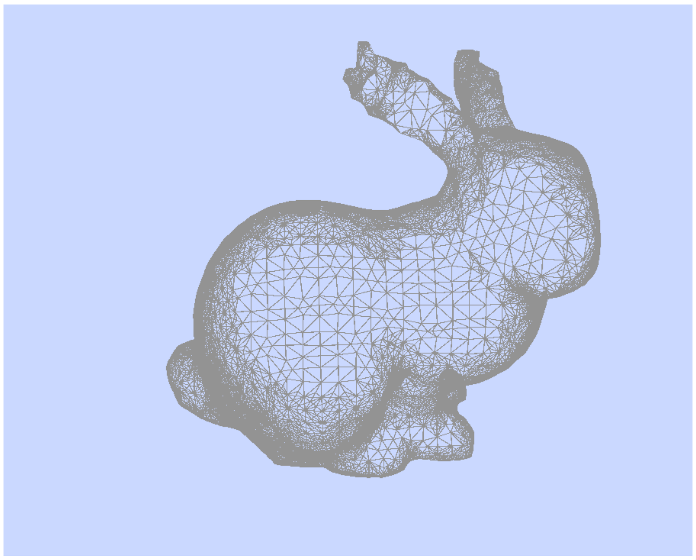


Figure 6.2.: Screenshot of the PNtriangles demo with enabled wireframe mode rendered in cuRE.

## 6.2. OpenGL

In OpenGL, the developer can control the tessellation with two shader programs. Between the two shaders, a fixed-function stage performs the actual tessellation. This stage can only be configured with a handful of parameters. The TCS allows the developer to specify how much subdivision should be done by choosing appropriate tessellation levels. This shader is optional and can be replaced with static tessellation levels that are valid for the whole draw call. If the shader is present, it is executed for each vertex of the primitive. A simple example of a TCS for triangles is shown in listing 6.1. In this example, we only set the tessellation levels with values from uniform variables and forward the vertex positions.

**Listing 6.1:** Simple example of a tessellation control shader in OpenGL. The tessellation levels are set with uniform variables and are identical for all primitives of the draw call.

```glsl
#version 450

layout(vertices = 3) out;

uniform float tessLevel[6];

 void main()
 {
    gl_TessLevelInner[0] = tessLevel[0];
    gl_TessLevelOuter[0] = tessLevel[2];
    gl_TessLevelOuter[1] = tessLevel[3];
    gl_TessLevelOuter[2] = tessLevel[4];

    gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
 }
```

The tessellation levels and additional information such as the spacing mode are then used in the tessellator. The tessellator is a fixed-function stage which subdivides the primitive according to predefined rules. The only way to influence these rules is by modifying the parameters of the input layout in the tessellation evaluation shader (cf. listing 6.2 line 5).

The TES allows the developer to compute per-vertex data since the shader is executed for each existing and newly generated vertex. The Barycentric coordinates of the vertex are input into the shader and allow the developer to interpolate the attributes of the existing vertices and perform other computations. A simple example of a TES for triangles is shown in listing 6.2. The shader only interpolates the original vertex positions to compute the position of a new vertex and applies the view and projection transforms to the computed position.

**Listing 6.2:** Simple example of a tessellation evaluation shader in OpenGL. The program only interpolates the position and applies the view and projection transformations.

```
#version 450

#include <camera>

layout(triangles, equal_spacing, cw) in;

 void main()
 {
    vec4 pos = (gl_TessCoord.x * gl_in[0].gl_Position +
                gl_TessCoord.y * gl_in[1].gl_Position +
                gl_TessCoord.z * gl_in[2].gl_Position);

   gl_Position = camera.PV * pos;
 }
```

OpenGL requires the developer to use *GL_PATCHES* as the primitive type to enable tessellation. It is a general-purpose primitive type that can only be used with tessellation, and the number of its vertices is user-defined. However, we limited the patch size to three and four in all tests to render triangles and quads respectively, because the pipeline in cuRE only supports these two sizes.

## 6.3. cuRE Implementation

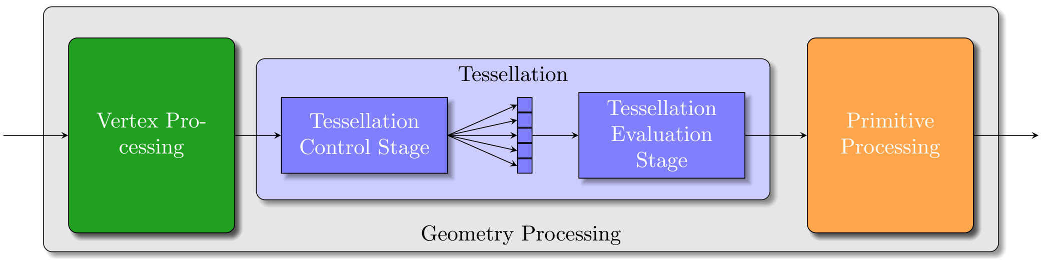


Figure 6.3.: Structure of the tessellation stages in the pipeline

The tessellation phase is included between the vertex processing stage, which executes the vertex shader and performs the triangle setup, and the primitive processing stage performing computations necessary for the rasterizer such as clipping, culling and calculating triangle edges. There are two stages needed for tessellation, the Tessellation Control Stage, and the Tessellation Evaluation Stage, which are described in chapter 5.1 and chapter 5.2, respectively. Between these two stages, there is also a work distribution stage. The setup is illustrated in Figure 6.3.

In contrast to the OpenGL pipeline, the shaders are programmed in C++ instead of GLSL.

To guarantee primitive order, we need to track the progress of primitives throughout the pipeline. Since tessellation creates new primitives dynamically within the pipeline, we do not register primitives when they enter the pipeline, but after the tessellation control stage. The tessellation control stage registers a primitive at the *progress queue* before forwarding its subpatches to the work distribution, to initialize the counters or bitfield. Then after the subpatch of a primitive reached the end of the geometry processing and is stored in the triangle buffer or discarded, the primitive processing stage informs the progress queue to mark the subpatch as finished.

The progress queue should return the number of ready triangles when we check the progress. However, the queue only manages the subpatch packets, and we need to convert this number to the number of triangles. For the progress tracking with counters as described in section 4.1.1, we just multiply the number of counters that are zero (ready primitives) with the maximum number of triangles a primitive can be split into. For the progress tracking with a bitfield as described in section 4.1.2, we multiply the number of set bits (ready subpatch packets) with the maximum number of triangles per subpatch packet, meaning the number of triangles per subpatch multiplied with the number of subpatches per packet.

The numbers computed by both progress queue methods which we call $p_{prog}$ do not represent the actual number of triangles which are ready, but the largest possible triangle id of the ready primitives. Computing the exact number of ready triangles would be extremely costly and therefore add a high overhead. However, we only need a triangle id which is less than the id of the first triangle which is not ready for the sorting, and the id does not need to belong to a valid triangle, because the sorting process only outputs valid triangles whose id is less or equal than $p_{prog}$.

# 7. Evaluation

To evaluate the correctness and performance of our approach, we tested it on the three demos mentioned in section 6.1.4. As test platform, we used an Intel Core i7-7700k CPU @ 4.5 GHz with 32 GiB of RAM running Windows 10. Experiments were performed on an NVIDIA GeForce RTX 2080 Ti. We used Cuda version 10.1, and the OpenGL reference implementation is based on OpenGL 4.5 core profile.

For a shorter notation, we use shorthands for different configurations of cuRE in tables and plots, i.e. $w/o$ indicates that primitive order is deactivated, $w/1$ is with primitive order and progress tracking with counters as described in section 4.1.1 and $w/2$ with primitive order and progress tracking with a bitfield for subpatches as described in section 4.1.2.

## 7.1. Results

To enable a comparison with OpenGL, we execute the tests of each demo with enabled rasterizer and fragment shading stage. However, as these stages take a significant amount of rendering time, we also provided rendering times with disabled rasterization and fragment shading. For these tests, we disabled the rasterizer in the frontend, so the timings still contain the triangle buffer and primitive ordering.

We use multiple models with different numbers of triangles for the demos. The exact number of triangles for each model with different tessellation levels is shown in Table 7.1.

| | Sphere | Suzanne | Bunny |
|---|---|---|---|
| TL 1 | 320 | 967 | 5,120 |
| TL 2 | 1,920 | 5,802 | 30,720 |
| TL 4 | 7,680 | 23,208 | 122,880 |
| TL 8 | 30,720 | 92,832 | 491,520 |
| TL 16 | 122,880 | 371,328 | 1,966,080 |
| TL 32 | 491,520 | 1,485,312 | 7,864,320 |
| TL 64 | 1,966,080 | 5,941,248 | 31,457,280 |

Table 7.1.: Number of triangles for different models and tessellation levels. Tessellation is applied uniformly, meaning that the inner and outer tessellation levels are identical.

### 7.1.1. Simple Demo

With this demo, we mainly compare the relative overhead of tessellation in cuRE with OpenGL. Therefore, we just divided the frame times at different tessellation levels through the times with tessellation levels equal to 1 to obtain overhead factors. The absolute frame times for all three models are shown in Table 7.2, whereas the relative overhead for the Sphere and Bunny model are shown in Figure 7.1 and Figure 7.2, respectively. For the larger models, we could not measure the times with tessellation levels 64 in cuRE, because the test timed out. The demo was executed with an active rasterizer and fragment shader, to compare the results of cuRE with OpenGL. As expected, the software approach cannot reach the performance of the hardware pipeline. As this demo uses extremely simple shader programs, OpenGL is more than 100 times faster than cuRE. This performance overhead is partially a constant overhead which explains the massive overhead of OpenGL for small models and small tessellation levels. The performance edge of OpenGL over cuRE can be attributed mostly to the better optimization and the implementation of stages, such as the rasterizer, in hardware. For complex scenes with large numbers of triangles, most of the performance overhead can be attributed to the sorting of the triangle buffer for restoring the primitive order.

Depending on the test case, restoring primitive order costs 15 % (for simple scenes with low triangle counts) to 1200 % (for very complex scenes) additional performance, due to the increasing sorting complexity. The type of progress tracking for the primitive ordering has only a small influence on the rendering time for most scenes. Only for very complex scenes with a large number of primitives before tessellation, the difference becomes larger until about 30 % with the Bunny model. The reason for this is the larger overhead from the atomic operations in the progress tracking method with counters.

| | | OpenGL | cuRE$_{w/o}$ | cuRE$_{w/1}$ | cuRE$_{w/2}$ |
|---|---|---|---|---|---|
| Sphere | TL 1 | 0.007 | 0.898 | 1.038 | 1.024 |
| | TL 2 | 0.008 | 0.987 | 1.131 | 1.102 |
| | TL 4 | 0.009 | 0.983 | 1.134 | 1.132 |
| | TL 8 | 0.014 | 1.101 | 1.322 | 1.326 |
| | TL 16 | 0.022 | 1.471 | 1.843 | 1.820 |
| | TL 32 | 0.042 | 2.860 | 4.177 | 4.174 |
| | TL 64 | 0.162 | 7.218 | 18.485 | 18.565 |
| Suzanne | TL 1 | 0.006 | 0.972 | 1.083 | 1.093 |
| | TL 2 | 0.009 | 1.078 | 1.304 | 1.341 |
| | TL 4 | 0.016 | 1.431 | 1.609 | 1.635 |
| | TL 8 | 0.026 | 2.348 | 2.916 | 2.934 |
| | TL 16 | 0.043 | 5.393 | 9.280 | 9.255 |
| | TL 32 | 0.110 | 14.955 | 70.621 | 70.110 |
| | TL 64 | 0.391 | 53.037 | — | — |
| Bunny | TL 1 | 0.008 | 0.983 | 1.118 | 1.123 |
| | TL 2 | 0.017 | 1.246 | 1.534 | 1.515 |
| | TL 4 | 0.024 | 1.796 | 2.328 | 2.119 |
| | TL 8 | 0.050 | 3.561 | 6.087 | 4.970 |
| | TL 16 | 0.132 | 9.396 | 32.862 | 23.346 |
| | TL 32 | 0.314 | 29.190 | 353.454 | 242.257 |
| | TL 64 | 1.666 | 110.585 | — | — |

Table 7.2.: Rendering time in ms of a few models of the simple demo. We compare OpenGL with different configurations of cuRE with different uniform tessellation levels.

We can also measure the time consumption or more precisely the number of elapsed clock cycles of each individual stage. We used these numbers to create stacked bar charts for the bunny model with and without primitive ordering

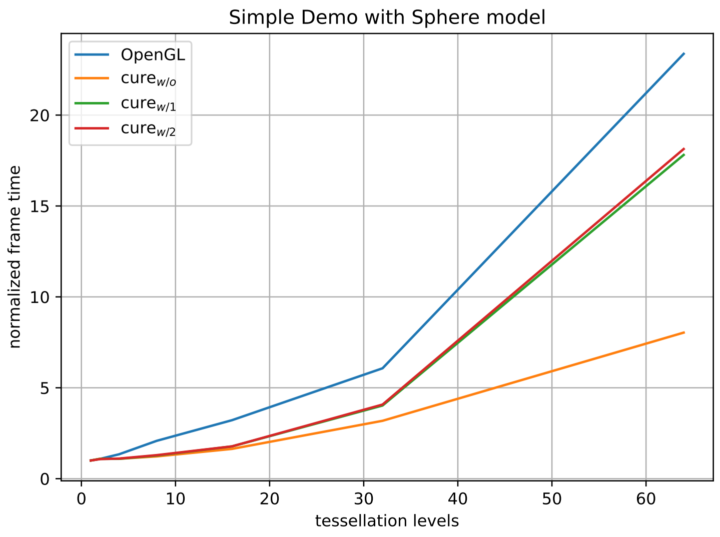


Figure 7.1.: Rendering time overhead (i.e. rendering time divided through the respective rendering time of tessellation level 1) of the Sphere model. We compare OpenGL with different configurations of cuRE with different tessellation levels.

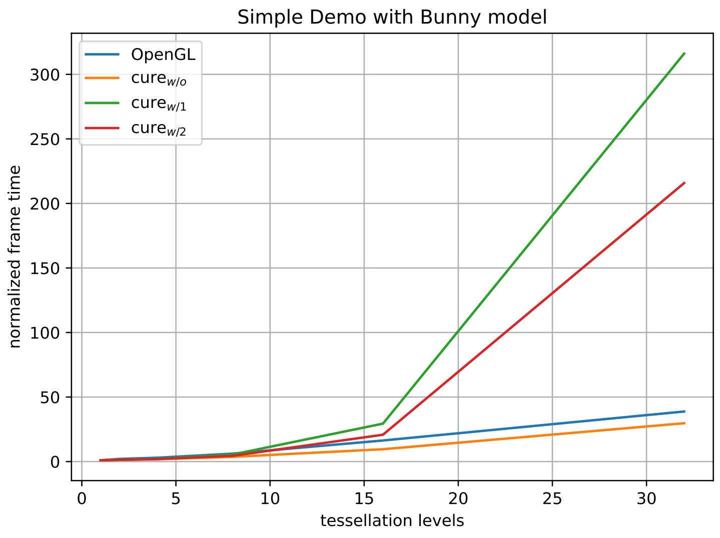


Figure 7.2.: Rendering time overhead (i.e. rendering time divided through the respective rendering time of tessellation level 1) of the Bunny model. We compare OpenGL with different configurations of cuRE with different tessellation levels.

which are shown in Figures 7.3 and 7.4, respectively. The rasterizer takes a considerable amount of the rendering time, but only uses about half of the multiprocessors, because of a fixed rasterizer pattern. Each rasterizer renders a fixed disjoint area of the image, and if there are no triangles in this area, the rasterizer does not take any time. The rasterizers lead to a suboptimal utilization, which is even more extreme with enabled primitive ordering since the sorting of the triangle buffer is done by the same multiprocessor that dequeues data. Moreover, without primitive order constraints, the rasterizers can start working as soon as any tringles finished geometry processing, which is also shown in Figure 7.4, where rasterizers start almost immediately in contrast to Figure 7.3.

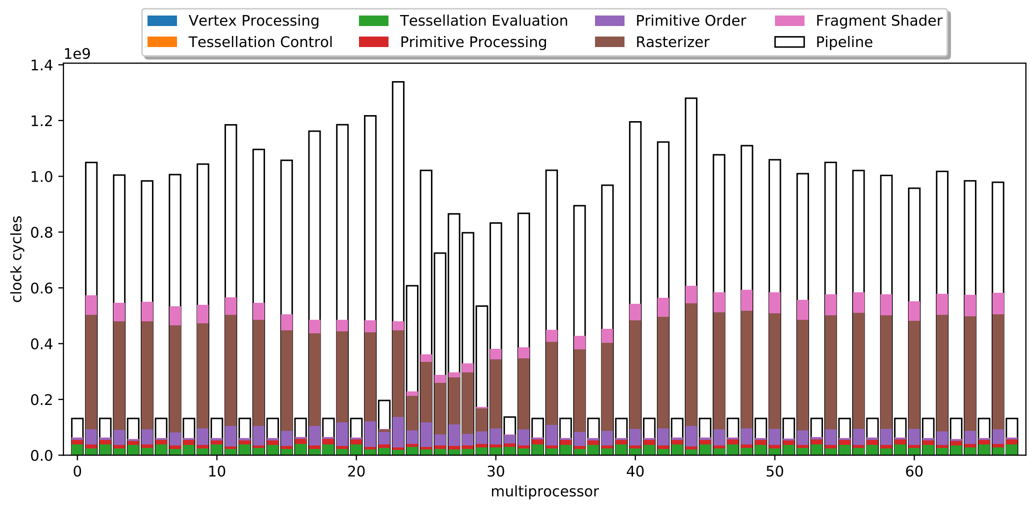


Figure 7.3.: Elapsed clock cycles for each stage of the simple demo with the Bunny model and primitive ordering. We used the progress tracking method with counters.

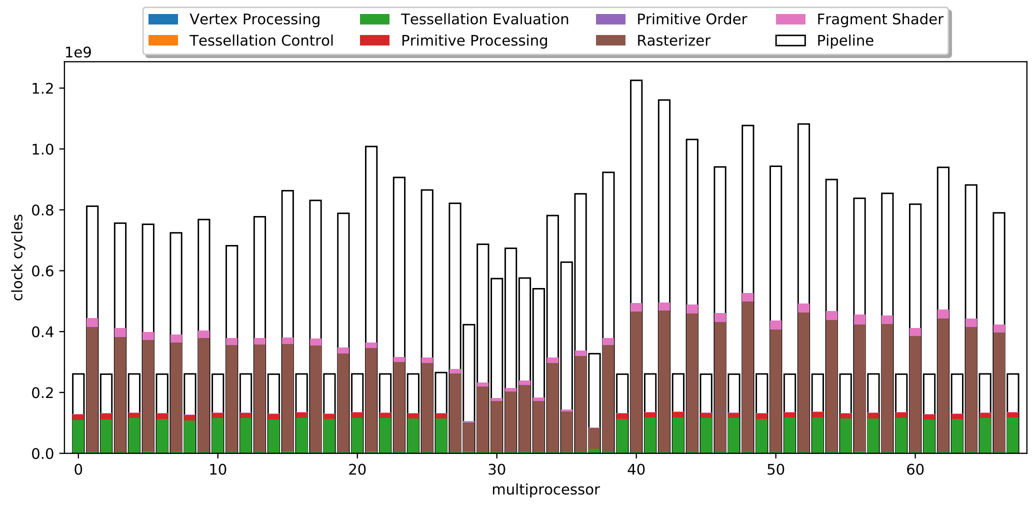


Figure 7.4.: Elapsed clock cycles for each stage of the simple demo with the Bunny model and without primitive ordering.

## 7.1.2. Terrain

We use the terrain demo to evaluate the performance of non-uniform and more commonly used scenes. Moreover, we are using quads as input primitives and for the tessellation instead of triangles. This demo uses tessellation levels depending on the distance of a triangle to the camera. Since the viewing angle is very flat, a wide range of tessellation levels will be present in the scene at the same time. Moreover, the demo also uses more complex shaders. The dynamic tessellation levels are computed in the tessellation control shader, and procedural noise for the displacement mapping is computed in the tessellation evaluation shader.

Table 7.3 shows frame times for the terrain demo rendered in cuRE and OpenGL. Surprisingly, increasing the size of the subpatch batches does not gain any performance but decreases it. We suspect that it may be due to unbalanced utilization of the hardware. Because of the large variety of different tessellation levels used simultaneously, warps processing a smaller batch of triangles and which are finished earlier have to wait on warps that take longer before they can get new data. Consequently, the overall utilization of the hardware is reduced, and the rendering time increases. The

| Batch Size | cuRE$_{w/o}$ | cuRE$_{w/1}$ | cuRE$_{w/2}$ |
|---|---|---|---|
| 2 | 9.128 | 10.650 | 10.680 |
| 4 | 11.050 | 12.486 | 12.526 |
| 8 | 14.848 | 16.009 | 15.956 |
| 12 | 17.570 | 18.696 | 19.138 |
| 16 | 20.549 | 21.698 | 21.374 |
| 20 | 22.012 | 22.919 | 23.522 |
| 24 | 23.810 | 24.669 | 24.859 |
| 32 | 26.873 | 28.117 | 28.242 |

Table 7.3.: Rendering time in ms of the terrain demo. We test different sizes of subpatch batches in cuRE and compare the rendering times to OpenGL which takes 0.509 ms.

work distribution between the tessellation control and evaluation stage is relatively simple, especially compared to the work performed in the stages.

With the more complex shader programs, the difference between OpenGL and cuRE also decreases, because the shaders are just programs running on the processor and are therefore equally fast in OpenGL and cuRE. In contrast to the simple demo, here OpenGL is only about 20 times faster. If we would also use more complex fragment shaders, we could further reduce the rendering time difference.

For this demo we also measured the number of elapsed clock cycles of each individual stage and created stacked bar charts with and without primitive ordering which are shown in Figures 7.5 and 7.6, respectively. In this demo, the rasterizer takes less of the rendering time than in the simple demo, and more of the multiprocessors are used since the scene fills the whole screen with triangles. Figure 7.6 shows again that the scheduler tries to equalize the rendering time across the multiprocessors by offloading the tessellation evaluation stage to multiprocessors which do little to no rasterization. Most of the rendering time is consumed by the tessellation evaluation stage, which makes sense since the shader is more complex than in the simple demo.

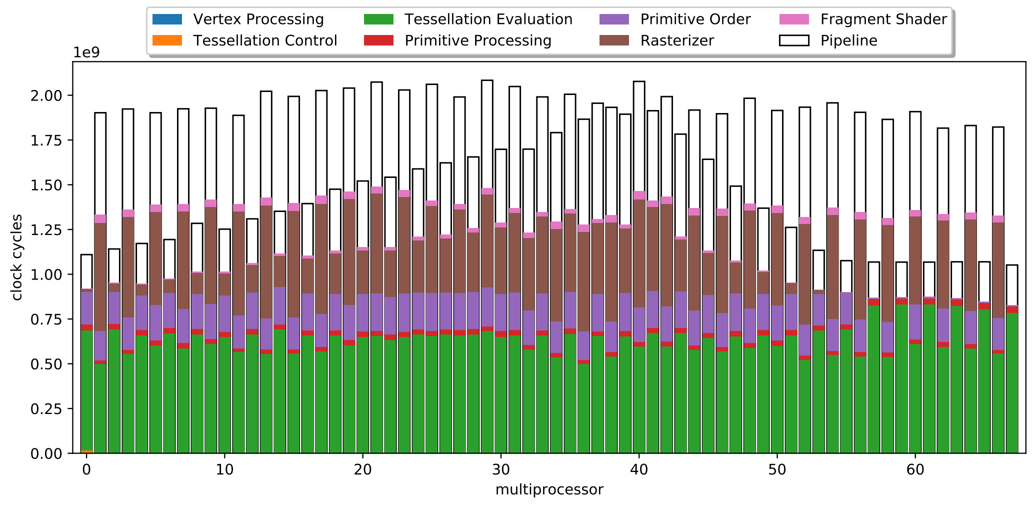


Figure 7.5.: Elapsed clock cycles for each stage of the terrain demo with primitive ordering. We used the progress tracking method with counters.

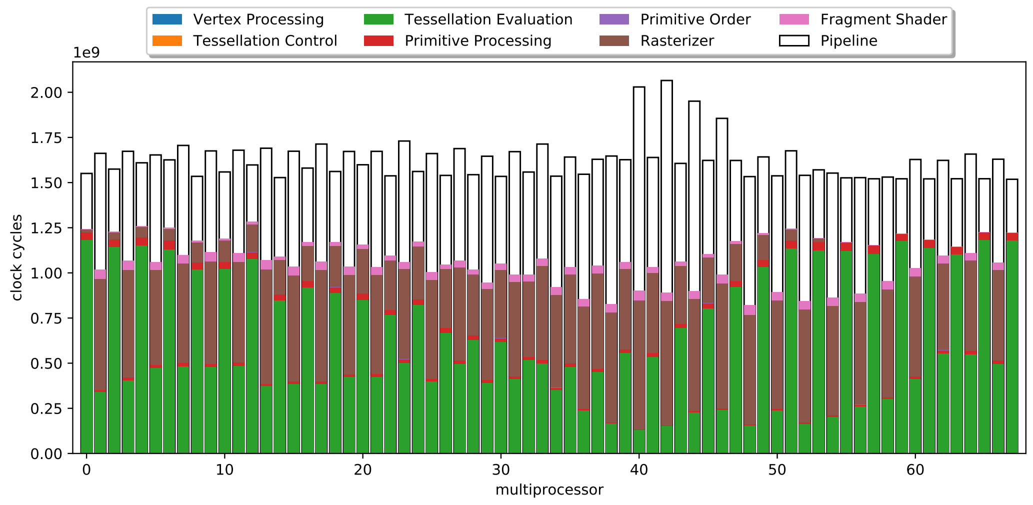


Figure 7.6.: Elapsed clock cycles for each stage of the terrain demo without primitive ordering.

### 7.1.3. PN Triangles

The PN Triangles demo again provides a more complex and praxis-relevant test scene using triangles. We compute adaptive tessellation levels in the tessellation control shader and Bézier coefficients in the tessellation evaluation shader, which increases their complexity. The overall complexity of the demo can be adjusted easily by using different models. However, it is also possible to influence the amount of tessellation at runtime with keyboard inputs.

Table 7.4 shows frame times for this demo with different models rendered in cuRE and OpenGL.We tested different subpatch batch sizes in cuRE and compared the results to OpenGL. The results are quite similar to the terrain demo: increasing the batch size does not improve the performance but decreases it. This behaviour can again be attributed to suboptimal utilization with larger batch sizes. Depending on the model, OpenGL is 3 to 15 times faster in this demo. The rendering time difference to OpenGL decreases with more complex scenes which have more triangles. Restoring primitive order increases the rendering time by 12 % - 46 %, depending on the model. The overhead is larger for models with more triangles which is again attributed to the sorting complexity.

The number of elapsed clock cycles for each stage with and without primitive ordering is shown in Figures 7.7 and 7.8, respectively. Again not all multiprocessors are used for rasterization since the model does not completely fill the screen. Moreover, the density of triangles varies strongly across the screen, which leads to different time consumption of different rasterizers. With primitive ordering enabled, the tessellation evaluation stage is distributed evenly across all multiprocessors since the rasterizers have to wait until the triangles in the correct order are ready until they can start. Without primitive ordering, the rasterizers with high work loads start almost immediately.

## 7.2. Summary

Overall, the work distribution for the tessellation stages is excellent in all demos, but the rasterizer and especially primitive ordering add significant overhead and suboptimal utilization. The performance is acceptable, but some

| | Batch Size | cuRE$_{w/o}$ | cuRE$_{w/1}$ | cuRE$_{w/2}$ |
|---|---|---|---|---|
| Sphere | 1 | 1.030 | 1.158 | 1.165 |
| | 2 | 1.358 | 1.533 | 1.253 |
| | 4 | 1.643 | 1.696 | 1.313 |
| | 8 | 1.876 | 1.970 | 1.397 |
| | 16 | 1.860 | 1.990 | 1.397 |
| | 20 | 1.893 | 2.021 | 1.374 |
| | 30 | 1.304 | 1.429 | 1.415 |
| Suzanne | 1 | 1.410 | 1.627 | 1.676 |
| | 2 | 1.541 | 1.954 | 1.807 |
| | 4 | 1.637 | 1.999 | 1.773 |
| | 8 | 1.735 | 2.205 | 1.858 |
| | 16 | 1.825 | 2.281 | 1.885 |
| | 20 | 1.877 | 2.301 | 1.914 |
| | 30 | 1.466 | 1.901 | 1.845 |
| Bunny | 1 | 1.824 | 2.667 | 2.261 |
| | 2 | 2.002 | 2.743 | 2.904 |
| | 4 | 2.035 | 2.703 | 2.716 |
| | 8 | 1.997 | 2.629 | 2.659 |
| | 16 | 2.082 | 2.795 | 2.772 |
| | 20 | 2.092 | 2.786 | 2.787 |
| | 30 | 1.835 | 2.569 | 2.605 |

Table 7.4.: Rendering time in ms of different models of the PN triangles demo. We test different sizes of subpatch batches in cuRE and compare the rendering times to OpenGL which takes 0.102 ms, 0.108 ms and 0.958 ms for the Sphere, Suzanne and Bunny model, respectively.

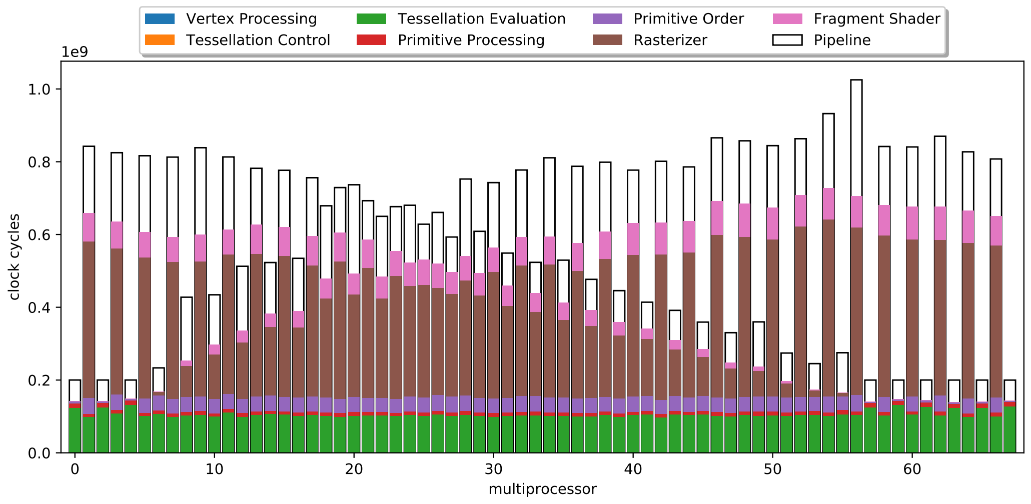


Figure 7.7.: Elapsed clock cycles for each stage of the PN-triangles demo with the Bunny model and primitive ordering. We used the progress tracking method with counters.

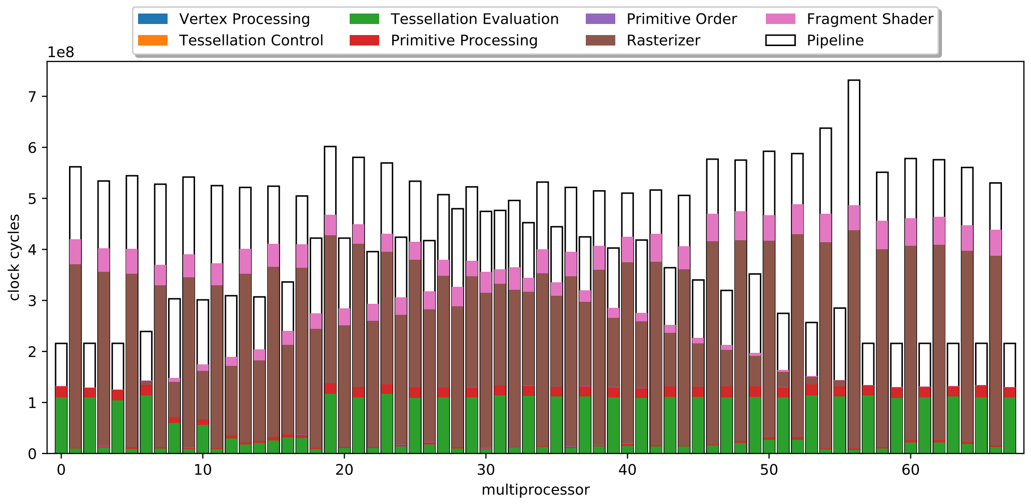


Figure 7.8.: Elapsed clock cycles for each stage of the PN-triangles demo with the Bunny model and without primitive ordering.

models with high tessellation levels show large rendering times. However, these cases generate an enormous amount of triangles which may not be viable for real-time applications anyway.

# 8. Conclusion

This thesis shows an implementation of a scalable tessellation stage in a streaming software rendering pipeline. The algorithm is fully parallelized on the GPU and adapts to different hardware configurations. We explained the subdivision and work distribution methods between the stages to make the process streamable. The tessellation algorithm adheres to the OpenGL specification and allows to define custom shader programs. We support triangles and quads as input primitives, and shaders can be programmed in C++.

We have shown that this implementation runs in real-time with praxis-relevant scenes, but some models with high tessellation levels show large rendering times. However, these cases generate an enormous amount of triangles which may not be viable for real-time applications.

## 8.1. Future Work

The compliance with the OpenGL specification adds additional overhead to the tessellation algorithm. Therefore, it may be possible to improve the performance of the algorithm by deviation from the specification. For special rendering techniques, it may also be beneficial to change the tessellation algorithm to achieve more pleasing results, improve the performance or simplify the implementation.

Patches are used as input primitive type for tessellation. In OpenGL, the size of the patches can be chosen arbitrarily with 1 to 64 vertices per patch. Our implementation, however, only supports patch sizes of three or four vertices per patch for triangle and quad rendering, respectively. Supporting additional patch sizes allows the implementation of more intricate tessellation techniques.

Our tessellation algorithm works exclusively on the GPU and processes the data in compact meshes (subpatches). Therefore, it is possible to implement the tessellation stage in the shaders of the new shading pipeline [Kub18] of the Turing architecture [NVI18].

# Appendix

# Appendix A.

# Algorithms

**Algorithm A.1** CalculateOuterTriangle

```
1: $pointingDown \leftarrow idx \bmod 2$
2: $row \leftarrow \frac{threadIdx}{subpatchSizeX*2}$    ▷ $threadIdx$ is the index of the thread within a warp.
3: $vStartLower \leftarrow \frac{threadIdx}{2} + subpatchRow$    ▷ $subpatchRow$ is the local row within the subpatch.
4: $vStartUpper \leftarrow \frac{threadIdx}{2} + subpatchRow + (subpatchSizeX + 1)$

5: $vertex.x \leftarrow vStartLower$
6: $vertex.y \leftarrow vStartLower + 1$
7: $vertex.z \leftarrow vStartUpper + \neg(row \geq numRings \mathbin{||} column \geq numRings)$
8: if $\neg pointingDown$ then
9:     $vertex.x \leftarrow vStartUpper + 1$
10:    $vertex.y \leftarrow vStartUpper$
11:    $vertex.z \leftarrow vStartUpper + \neg(row \geq numRings \mathbin{||} column \geq numRings)$
12: end if
```

# Appendix B.

# List of control keys

| Key | Action |
|---|---|
| General: | |
| 'Tab' | Switch the Renderer |
| 'F8' | Screenshot |
| 'Esc' | Exit the program |
| General Rendering control: | |
| 'W' | Toggle wireframe rendering |
| Simple demo controls: | |
| 'N' | Toggle triangle/quad rendering |
| 'M' | Switch the tessellation spacing mode |
| 'Left' | Increase inner tessellation level 0 |
| 'Right' | Decrease inner tessellation level 0 |
| 'Up' | Increase inner tessellation level 1 |
| 'Down' | Decrease inner tessellation level 1 |
| 'Num 7' | Increase outer tessellation level 0 |
| 'Num 1' | Decrease outer tessellation level 0 |
| 'Num 8' | Increase outer tessellation level 1 |
| 'Num 2' | Decrease outer tessellation level 1 |
| 'Num 9' | Increase outer tessellation level 2 |
| 'Num 3' | Decrease outer tessellation level 2 |
| 'Num +' | Increase outer tessellation level 3 |
| 'Num -' | Decrease outer tessellation level 3 |

*Continued on next page*

*Continued from previous page*

| Key | Action |
|---|---|
| PN Triangles demo controls: | |
| ’M’ | Switch between PN/flat tessellation |
| ’Up’ | Increase mesh resolution |
| ’Down’ | Decrease mesh resolution |
| Terrain demo controls: | |
| ’Up’ | Increase mesh resolution |
| ’Down’ | Decrease mesh resolution |
| ’Num 7’ | Increase terrain size |
| ’Num 3’ | Decrease terrain size |
| ’Num 8’ | Increase input mesh resolution |
| ’Num 2’ | Decrease input mesh resolution |

# Appendix C.

# Configuration file options

| Option | Value | Description |
|---|---|---|
| renderer | str | Set the used Rendering system (OpenGL or cuRE) |
| └ module | str | Set the used demo |
| └ simple_demo | | |
|   └ model_path | str | Path to the OBJ-file to render |
|   └ renderer | | |
|     └ mode | int [0, 1] | Quad (1) or triangle (0) rendering |
|   └ tessellation | | |
|     └ mode | int [0,1,2] | Spacing mode for tessellation |
|     └ inner_(0,1) | float | Inner tessellation levels |
|     └ outer_(0-3) | float | Outer tessellation levels |
| └ terrain | | |
|   └ size_(x,y) | float | Terrain size |
|   └ tile_size_(x,y) | float | Size of each quad composing the terrain |
|   └ triSize | float | Refinement factor for LOD calculation |
| └ pn_triangles | | |
|   └ model_path | str | Path to the OBJ-file to render |

# Bibliography

[AL09] Timo Aila and Samuli Laine. "Understanding the efficiency of ray traversal on GPUs." In: *Proceedings of the HPG 2009: Conference on High-Performance Graphics 2009*. 2009. ISBN: 9781605586038. DOI: 10.1145/1572769.1572792 (cited on page 10).

[AS19] Kurt Akeley and Mark Segal. "The OpenGL Graphics System : A Specification (Version 4.6 (Core Profile))." In: (2019). [Online; accessed 16-March-2020]. URL: https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf (cited on page 1).

[Bly06] David Blythe. "The Direct3D 10 system." In: *ACM SIGGRAPH 2006 Papers, SIGGRAPH '06*. 2006. ISBN: 1595933646. DOI: 10.1145/1179352.1141947 (cited on page 1).

[BS05] Tamy Boubekeur and Christophe Schlick. "Generic mesh refinement on GPU." In: *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*. [Online; accessed 24-March-2020]. 2005. ISBN: 1595930868. DOI: 10.1145/1071866.1071882. URL: https://hal.inria.fr/file/index/docid/260838/filename/GenericMeshRefinementOnGPU.pdf (cited on page 12).

[BS08] T. Boubekeur and C. Schlick. "A flexible kernel for adaptive mesh refinement on GPU." In: *Computer Graphics Forum* (2008). [Online; accessed 24-March-2020]. ISSN: 14678659. DOI: 10.1111/j.1467-8659.2007.01040.x. URL: https://hal.inria.fr/inria-00260825/document (cited on page 12).

[FH08] Kayvon Fatahalian and Mike Houston. "A closer look at GPUs." In: *Communications of the ACM* (2008). ISSN: 00010782. DOI: 10.1145/1400181.1400197 (cited on page 7).

[Ken+18] Michael Kenzel, Bernhard Kerbl, DIeter Schmalstieg, and Markus Steinberger. "A high-performance software graphics pipeline architecture for the GPU." In: *ACM Transactions on Graphics* 37.4 (2018). ISSN: 15577368. DOI: 10.1145/3197517.3201374 (cited on pages 11, 12, 28, 30).

[Khr10] Khronos Group. *ARB Tessellation Shader*. [Online; accessed 07-May-2020]. 2010. URL: https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB%7B%5C_%7Dtessellation%7B%5C_%7Dshader.txt (cited on page 12).

[Khr20] Khronos Group. "Vulkan 1.1.132 - A Specification." In: (2020). [Online; accessed 16-March-2020]. URL: https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html (cited on page 1).

[Kub18] Christoph Kubisch. *Introduction to Turing Mesh Shaders — NVIDIA Developer Blog*. [Online; accessed 16-June-2020]. Sept. 2018. URL: https://devblogs.nvidia.com/introduction-turing-mesh-shaders/#toc2 (cited on pages 13, 14, 63).

[Liu+10] Fang Liu, Meng Cheng Huang, Xue Hui Liu, and En Hua Wu. "FreePipe: A programmable parallel rendering architecture for efficient multi-fragment effects." In: *Proceedings of I3D 2010: The 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. [Online; accessed 08-May-2020]. 2010. ISBN: 9781605589381. DOI: 10.1145/1730804.1730817. URL: https://dl.acm.org/doi/epdf/10.1145/1730804.1730817 (cited on page 9).

[LK11] Samuli Laine and Tero Karras. "High-performance software rasterization on GPUs." In: *Proceedings - HPG 2011: ACM SIGGRAPH Symposium on High Performance Graphics*. 2011. ISBN: 9781450308960. DOI: 10.1145/2018323.2018337 (cited on page 9).

[Llo+07] D. Brandon Lloyd, Naga K. Govindaraju, Steven E. Molnar, and Dinesh Manocha. "Practical logarithmic rasterization for low-error shadow maps." In: *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*. 2007. ISBN: 9781595936257 (cited on page 3).

[MB05] Tom McReynolds and David Blythe. *Advanced Graphics Program. Using OpenGL*. 2005. ISBN: 9781558606593. DOI: 10.1016/B978-1-55860-659-3.X5000-8 (cited on page 26).

[Mic15] Microsoft. *Direct3D 11.3 Functional Specification*. [Online; accessed 07-May-2020]. 2015. URL: https://microsoft.github.io/DirectX-Specs/d3d/archive/D3D11%7B%5C_%7D3%7B%5C_%7DFunctionalSpec.htm (cited on page 12).

[Nic+08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. "Scalable parallel programming with CUDA." In: *Queue* (2008). ISSN: 15427730. DOI: 10.1145/1365490.1365500 (cited on page 7).

[Nie+16] M. Nießner, B. Keinert, M. Fisher, M. Stamminger, C. Loop, and H. Schäfer. "Real-Time Rendering Techniques with Hardware Tessellation." In: *Computer Graphics Forum*. [Online; accessed 25-March-2020]. 2016. DOI: 10.1111/cgf.12714. URL: https://graphics.stanford.edu/%7B~%7Dmdfisher/papers/realtimeRendering.pdf (cited on page 13).

[NVI18] NVIDIA Corporation. *NVIDIA Turing GPU Architecture*. [Online; accessed 16-June-2020]. 2018. URL: https://www.industry-era.com/images/pdf/NVIDIA-Turing-Architecture-Whitepaper.pdf (cited on pages 13, 63).

[NVI19] NVIDIA Corporation. "CUDA C Programming Guide." In: *Programming Guides* September (2019). [Online; accessed 16-March-2020], pages 1–346. ISSN: 15284972. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (cited on pages 1, 4–6, 8).

[Pat+15] Anjul Patney, Stanley Tzeng, Kerry A. Seitz, and John D. Owens. "Piko: A framework for authoring programmable graphics pipelines." In: *ACM Transactions on Graphics*. 2015. DOI: 10.1145/2766973 (cited on page 9).

[SGS10] John E. Stone, David Gohara, and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems." In: *Computing in Science and Engineering* (2010). ISSN: 15219615. DOI: 10.1109/MCSE.2010.69 (cited on page 1).

[SS09] Michael Schwarz and Marc Stamminger. "Fast GPU-based adaptive tessellation with CUDA." In: *Computer Graphics Forum* (2009). [Online; accessed 24-March-2020]. ISSN: 14678659. DOI: 10.1111/j.1467-8659.2009.01376.x. URL: http://research.michael-schwarz.com/publ/files/cudatess-eg09.pdf (cited on page 12).

[Sta19] Pascal Stadlbauer. "Streaming Primitive Tessellation for High-Performance Software Rendering Pipelines." PhD thesis. Graz University of Technology, 2019 (cited on pages 15, 16, 18, 21, 23, 24).

[Ste+14] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. "Whippletree: Task-based scheduling of dynamic workloads on the GPU." In: *ACM Transactions on Graphics*. [Online; accessed 25-March-2020]. 2014. DOI: 10.1145/2661229.2661250. URL: https://arbook.icg.tugraz.at/schmalstieg/Schmalstieg_286.pdf (cited on page 10).

[TPO10] Stanley Tzengy, Anjul Patney, and John D. Owens. "Task management for irregular-parallelworkloads on the GPU." In: *High-Performance Graphics - ACM SIGGRAPH / Eurographics Symposium Proceedings, HPG*. 2010. ISBN: 9783905674262 (cited on page 10).

[Vla+01] A. Vlachos, J. Peters, C. Boyd, and J. L. Mitchell. "Curved PN triangles." In: *Proceedings of the Symposium on Interactive 3D Graphics*. 2001. DOI: 10.1145/364338.364387 (cited on page 42).

[Wik19] OpenGL Wiki. *Rendering Pipeline Overview — OpenGL Wiki,* [Online; accessed 16-March-2020]. 2019. URL: http://www.khronos.org/opengl/wiki_opengl/index.php?title=Rendering_Pipeline_Overview&oldid=14511 (cited on pages 1, 2).