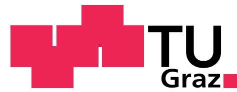


Pascal Stadlbauer, BSc

Streaming Primitive Tessellation for High-Performance Software Rendering Pipelines

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

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

Institute for Computer Graphics

Graz, January 2019

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

A need for mesh refinement at run time led to tessellation stages being added to hardware pipelines. Applications that require specific rendering techniques, that are not implementable or efficient on hardware pipelines, often use software rendering pipelines. These software rendering pipelines usually consist of various stages, that in the end produce an image. This thesis shows an implementation of a Primitive Tessellation stage for a Streaming Software Rendering Pipeline executed on the GPU. The pipeline this implementation is designed for uses a megakernel with dynamic load balancing to increase GPU utilization. Therefore this thesis shows a Tessellation stage, that is able to handle batches of data and redistribute work in between. To get a comparison to hardware rendering, the tessellation procedure orients itself on the OpenGL specification.

Contents

Abstract v

1. Introduction 1
1.1. NVIDIA GPU 3

2. Related Work 7
2.1. Software Rendering Pipeline 7
2.2. Tessellation 9

3. Tessellation Procedure 11
3.1. Edge Tessellation Level Conversion 12
3.2. Edge division 12
3.3. Primitive Tessellation Sections 14
3.4. Quad Inner Tessellation 16
3.4.1. Specific Implementation 17
3.5. Triangle Inner Tessellation 19
3.5.1. Specific Implementation 20
3.6. Outer Tessellation 22
3.6.1. Specific Implementation 22

4. Implementation 25
4.1. A look at openGL 25
4.2. Stage Implementation 27
4.3. Tessellation Control Stage 28
4.3.1. Number of Vertices and Triangles 28
4.3.2. Subpatch division 28
4.4. Tessellation Evaluation Stage 31
4.4.1. Global index calculations 31

5. Evaluation **35**
5.1. Results 35
5.1.1. Tessellation Results 35
5.1.2. Subpatches 38
5.1.3. Models 39
5.2. Comparison 44
5.3. Testing 46
5.4. Conclusion 47

A. Algorithms **51**

B. Lists **65**

Bibliography **71**

1. Introduction

Nowadays graphics applications need to perform various specific tasks. Traditional implementations rely on the hardware rendering pipeline to generate graphics. These pipelines are interacted with via an API like OpenGL, Direct3D or Vulkan. Primitive data is handed to the pipeline and then processed by multiple sequential stages. Due to the sequential processing by fixed stages and the user sending instructions through an API, adaption to special rendering needs is often not possible.

Because of these drawbacks the focus shifts to software rendering pipelines, when specific applications are needed. Software rendering pipelines have the advantage of giving the developer more possibilities to implement applications. Their whole pipeline can be designed to fit a specific purpose. Software rendering pipelines can be implemented to exploit both benefits of the CPU and the GPU. The GPU is used in compute mode to process high amount of data in a parallelized way. The main aspect to consider is that data needs to be efficiently processed on hardware, that was traditionally designed for hardware pipelines.

One of the stages in hardware rendering pipelines is called the Tessellation stage. This stage is used to divide the primitives to smaller primitives during run time. Because of this stage memory consumption is kept lower and a finer subdivision is only created, when it is intended and needed. Because of more subdivisions, it is possible to make the model seem smoother, and give it more detail for example through a displacement map.

To create these smaller primitives, vertices are interpolated and then connected. For the interpolation and connections multiple techniques exist. The technique this thesis takes a closer look at is the one specified by OpenGL. An OpenGL tessellation example can be seen in Figure 1.1

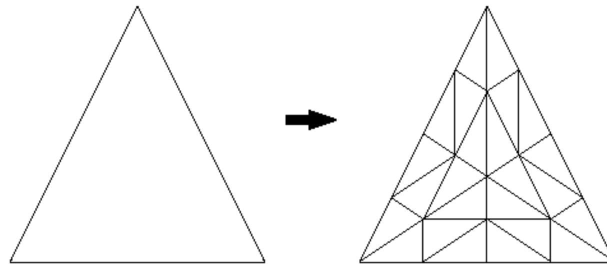


Figure 1.1.: Tessellation illustration

For this thesis a tessellation stage is created, that can be integrated into a software rendering pipeline, that is written in CUDA [4]. The software rendering pipeline is assumed to have a streaming architecture and therefore a technique will be shown to divide the workload and distribute the work on the GPU. An illustration of how individual subpatches build a tessellated quad primitive can be seen in Figure 1.2.

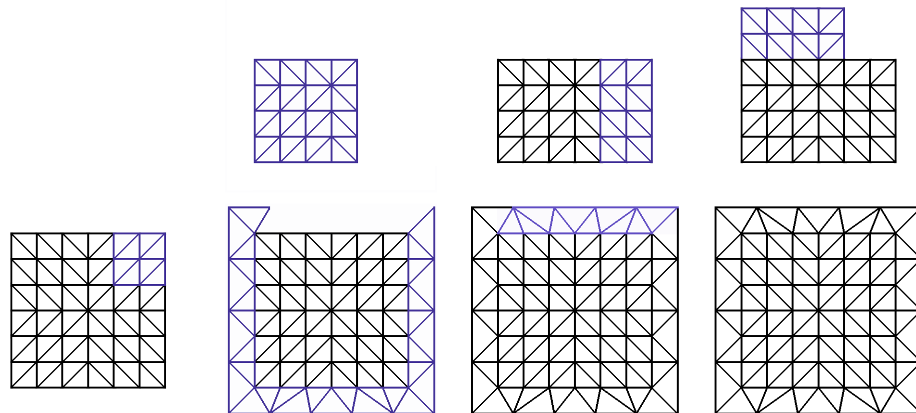


Figure 1.2.: Subpatching illustration

To get acquainted with the targeted pipeline and the work it builds on, the previous and related work are shown first. After that the tessellation procedure and the needed requirements to be conform to the OpenGL specification are explained in detail. The implementation, in terms of specific adaption to the streaming pipeline is mentioned afterwards. At last results of the tessellation process are shown in the evaluation section and a conclusion is drawn.

1.1. NVIDIA GPU

The concept of parallelism on the GPU consists of multiple parallel threads calling the same kernel function. These threads are grouped together into blocks. Individual threads in these blocks can be identified by a one-, two- or three-dimensional identifier. An example of this would be, one using a two-dimensional id for image processing, or a three-dimensional id for real world simulations. Figure 1.3 shows an illustration of two-dimensional identification.

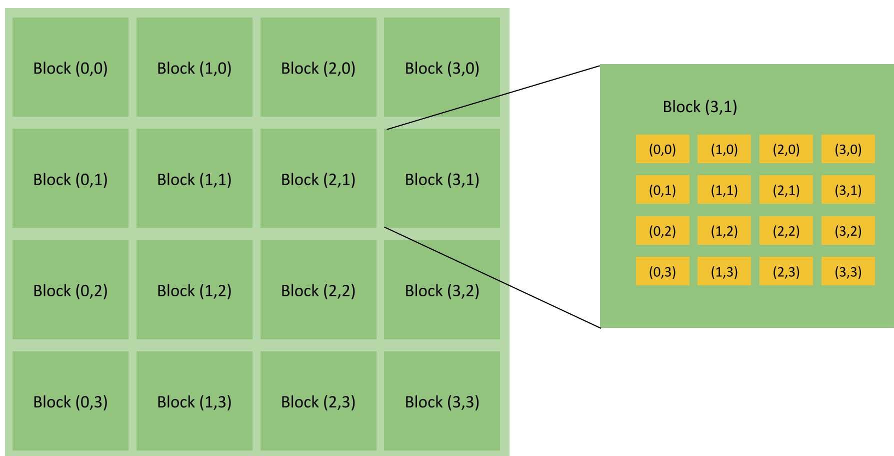


Figure 1.3.: Nvidia Thread-Block Structure

Thread blocks are given to an array of streaming multiprocessors (SM) for execution. In order for a single SM to manage its assigned blocks, it groups it into warps of 32 threads. Warps are handled by the warp scheduler, which schedules and executes them.

SM are built as a SIMT(Single Instruction - Multiple Thread) architecture, which is based on the SIMD (Single Instruction - Multiple Data) architecture. This means efficiency is high when all threads execute the same code at the same time. If threads within a warp choose different code paths, then the multiprocessor is forced to execute these paths serially. Only when all taken code paths are finished, the threads converge back together. This is why branching inside a warp should be kept at a minimum.

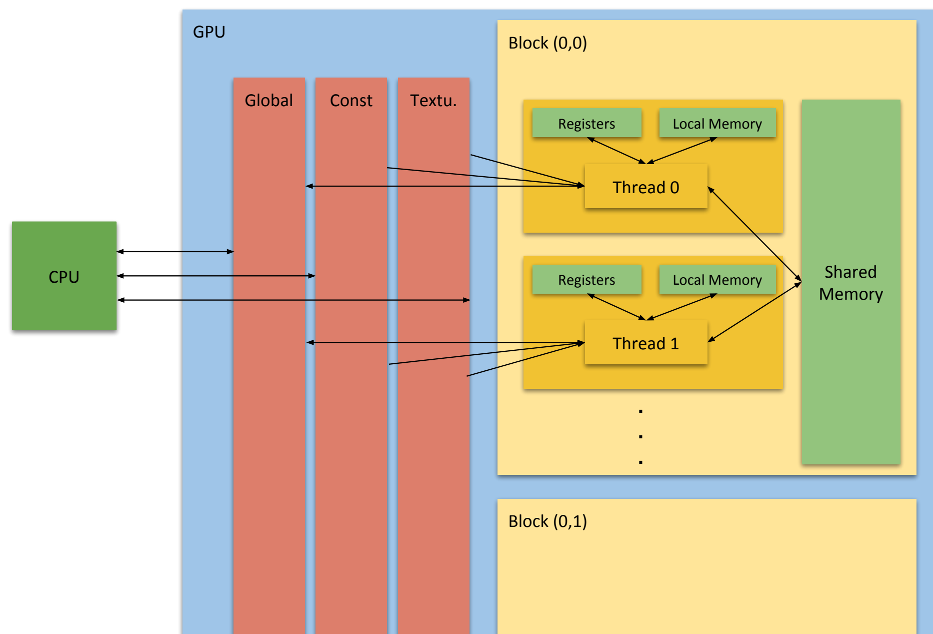


Figure 1.4.: Nvidia Memory Model

Another major influence on efficiency is data transfer. The time to transfer data can outweigh the time for computation. Especially transfers between Host and Device have an large impact on performance. This is why there are

multiple memory spaces, with different storage capabilities and access times. Threads have registers and local memory for their own. These memory spaces can only be accessed by the thread its owned by. A whole block owns shared memory, which can be accessed by all threads in the block and can be for example used to exchange data between individual threads. For its lifetime the application itself has persistent global, texture and constant memory. Memory access is slower the farther the memory is away from the actual thread.

2. Related Work

2.1. Software Rendering Pipeline

In Software Rendering Pipelines various stages process data in order to generate an image. The first pipelines used the CPU to process data, which has limited efficiency, due to the limited parallelism of the CPU. Parallelism is a key part for this kind of graphics processing, because a lot of data has to be processed in the same manner in a short amount of time. After a compute mode for GPUs became available, implementations for the GPU started to emerge. These implementations benefit from the parallel processing that is done on the GPU.

One of the early software rendering architectures was Larrabee [15]. It used multiple x86 CPUs augmented by a wide vector processor unit. An early GPU based architecture was FreePipe [7]. Problems with this approach are the non-linear sorting complexity, that is getting higher with depth complexity.

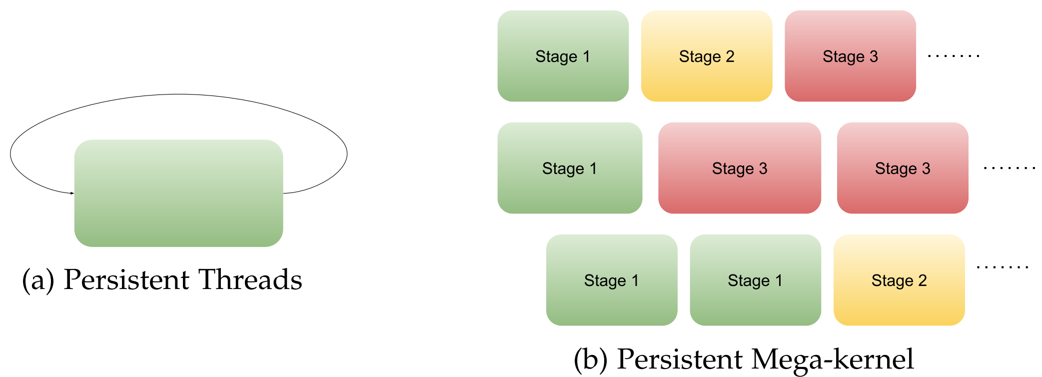
CudaRaster [6] then showed high-performance highly-optimized Rasterization on GPUs. It executes the pipelines stages in a sequential order. One of the latest architectures, that also uses sequential stages, is Piko [13]. Piko uses spatial locality to implement a binning approach. Primitives are grouped into bins, to be further processed.

These traditional approaches consist of simply executing one stage after the other. This means that every multiprocessor executes the same stage, at the same time, as seen in 2.1. As a result the workload is not evenly distributed over time and high computational needs lead to a bottleneck.

Figure 2.1.: Traditional Pipe

The persistent threads approach as introduced by Aila et al. [1] creates a more efficient work distribution. To bypass the normal scheduler, the whole GPU is filled with threads once. Every thread fetches work from a global work queue. This global work queue can also be a bottleneck.

A persistent mega kernel, as described by Steinberger et al. [16], uses the same concept of filling up the GPU. But in this approach every multiprocessor chooses individually, which stage to execute next. The work is fetched from a procedure specific queue.



(a) Persistent Threads

(b) Persistent Mega-kernel

Figure 2.2.: Scheduling approaches

Target Pipeline

The target pipeline, introduced in "A high-performance software graphics pipeline architecture for the GPU" [5], that the tessellation could be integrated into, implements a persistent mega-kernel. As explained above scheduling of the processing steps is done by the multiprocessor according to a specific pattern. The primitives need to go through the following steps in this order. First Geometry Processing projects the triangles and sorts out the ones, that are hidden. After that the triangles are redistributed for Rasterization, based on which screen region they cover. Each rasterizer then processes the triangles in its screen bin. At last the fragments are transferred into the framebuffer.

2.2. Tessellation

There are only a few general Tessellation approaches, that only depend on the GPU. To overcome the hardware limitation of non-existent geometry generation in 2005, Tamy Boubekeur and Christophe Schlick [3] used the GPU to create a general simple mesh refinement. After the geometry shader was introduced, "Generic mesh refinement on GPU."[2] showed a more flexible and adaptive mesh refinement. This mesh refinement adapts to the distance to the camera.

A parallelized tessellation on the patch level was implemented as CudaTess introduced by Schwarz et al. [14] . For CudaTess to work efficiently on the GPU many patches, with a low number of subdivisions are needed to exploit the parallelized architecture.

Sometimes a combination of CPU, GPU computation and hardware is used to create a tessellation. A mixture of these is used by Matthias Nießner et al [11] to create adaptive Catmull-Clark surfaces. Pixar's OpenSubdiv API [17] uses this algorithm and became a popular approach for these problems.

As Tessellation became more popular, it was integrated as a new stage in hardware pipelines. This stage was introduced with DirectX11[10] and OpenGL4.0[8].

After hardware tessellation became available, the focus of the work on tessellation shifted from the concrete implementation itself, to using hardware tessellation efficiently. An overview over the various techniques is given in "Real-Time Rendering Techniques with Hardware Tessellation" [12].

3. Tessellation Procedure

The goal of tessellation is to divide primitives, in this case quads and triangles, into smaller sections. Both primitive types are divided into triangles, so that further processing by the rendering pipeline, can be done uniformly.

The tessellation process is done according to the OpenGL specification [9], to produce similar results. Following sections will summarize the procedure given by OpenGL and explain parts, that are implementation specific.

Customization of the tessellation is achieved by choosing values and a type. These parameters define the number of subdivisions, that are made and are referred to as tessellation levels and tessellation type.

To divide quads and triangles, it is first shown how tessellation levels are converted. After that the individual edge subdivision, that is consistent for every edge, is discussed. Then it is shown how the inner quads and triangles are tessellated. Due to the similarities of the outer sections of quads and triangles, the tessellation for the outer section is summarized in the last section.

3.1. Edge Tessellation Level Conversion

Tessellation levels and the spacing type are usually chosen based on a specific purpose. Sometimes these levels are chosen at run time, to adapt to, for example the distance to the camera. This means that the levels calculated may not match the requirement of the tessellation spacing type and may not be processable by the succeeding tessellation stages. Therefore a conversion is needed to make sure, that the requirements are met. This is done as specified in the OpenGL documentation [9].

First all chosen tessellation levels(f) are clamped. After this these values are rounded and stored as the converted levels(n). The specific clamping boundaries and the rounding target are chosen, based on the spacing type as seen in 3.1.

type	clamp	round
equal spacing	1,max	up to n, $n \in \mathbb{N}$
fractional even spacing	2,max	up to n, $\frac{n}{2} \in \mathbb{N}$
fractional odd spacing	1,max-1	up to n, $\frac{n+1}{2} \in \mathbb{N}$

Table 3.1.: Clamping and rounding specification

3.2. Edge division

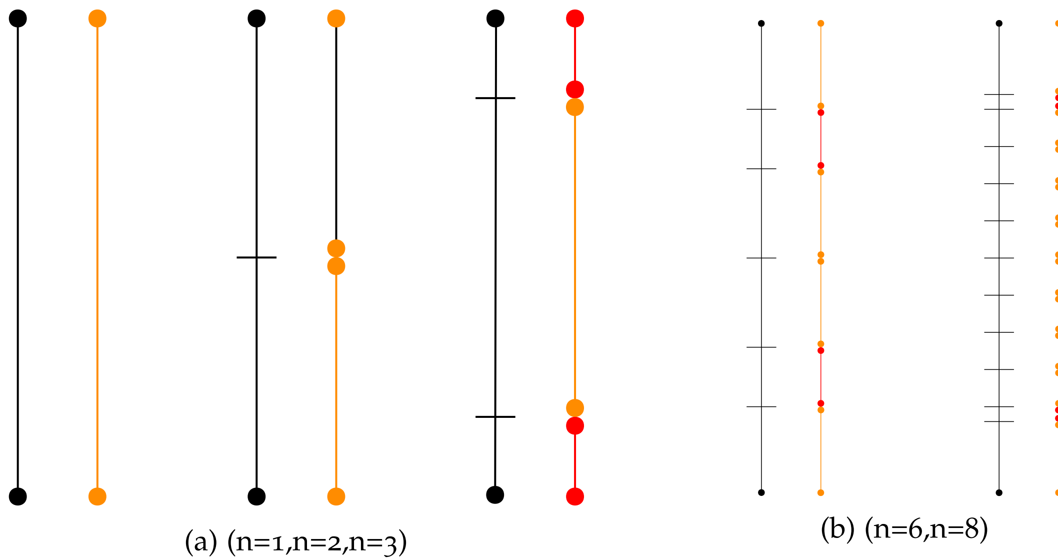
Each edge of the original primitive has its own two tessellation levels. One level (f) is chosen and the other one (n) is computed from f. The tessellation level n indicates in how many segments the edge is divided. While the levels vary between edges, the procedure stays the same.

In the simplest case, the level is one and therefore the edge will not be divided. If it is two, two segments with equal length exist. Otherwise the edge is split into $n - 2$ segments of equal length. The remaining two segments, referred to as "short segments", have equal length and are placed symmetrically.

In this implementation short segments are placed on the second outermost position, on each side. Except, when n equals 3, then the shorter segments are placed on both sides of the single normal segment. The lengths of the segments, for this implementation, are computed as follows,

$$l_{normal} = \frac{1}{n}$$

$$l_{short} = \frac{n - f}{2} * \frac{1}{n}$$



(a) (n=1,n=2,n=3) (b) (n=6,n=8)

Figure 3.1.: Special edge division levels are shown in a, and some normal cases in b, . Red indicates short segments and orange, normal segments

3.3. Primitive Tessellation Sections

Different primitives can have different tessellation levels. To get a uniform transition between primitives with different tessellation levels, the outer edges have to have the same number of vertices. Otherwise this would lead to a non-manifold mesh and possible gaps in the model. Therefore there exists an inner and an outer tessellation level, such that a smooth transition is ensured and different tessellation levels for primitives are made possible. This results in two sections for a primitive, as seen in Figure 3.2 and Figure 3.3.

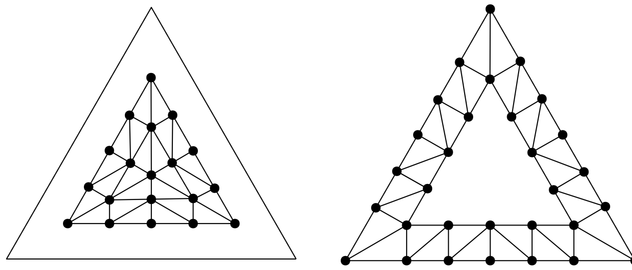


Figure 3.2.: Inner and outer section of a triangle

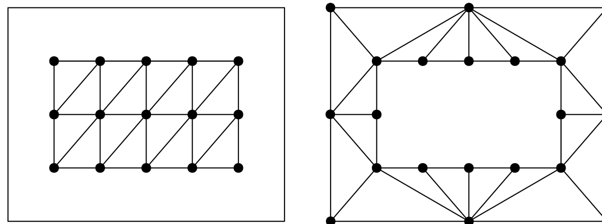


Figure 3.3.: Inner and outer section of a quad

The inner tessellation will be discussed individually for each primitive type in Section 3.4 and Section 3.5 . Each outer edge, with its inner section edge counterpart, can be viewed as an individual section, as seen in Figure 3.4. Due to the similarities of the individual outer sections across the primitive

types, the generation procedure is the same and can be found in Section 3.6.

In addition to coordinates, vertices can have multiple properties. This is why in the following sections, the interpolation values, rather than the coordinates, are discussed.

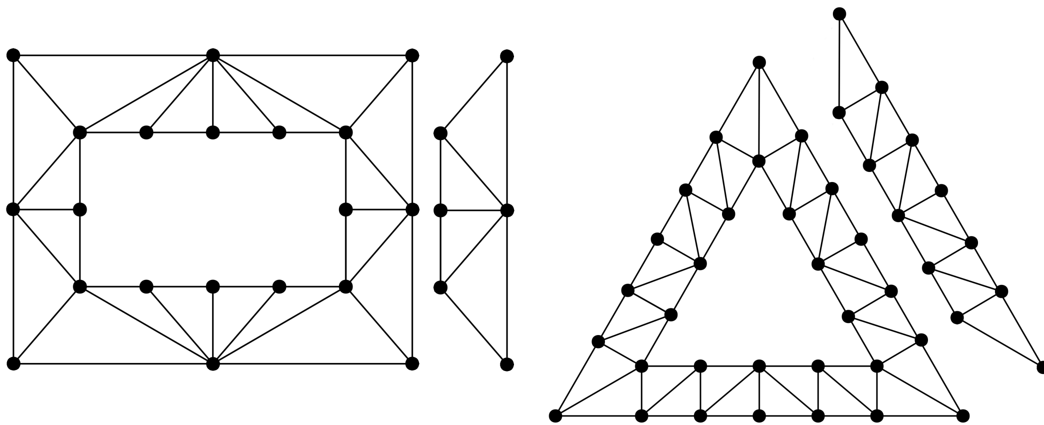
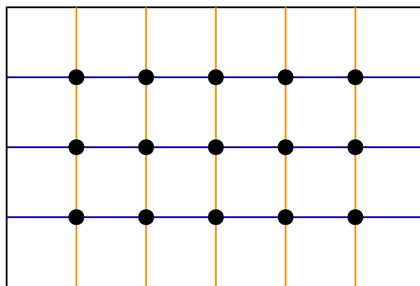


Figure 3.4.: Visualization of an outer section part of a quad and a triangle

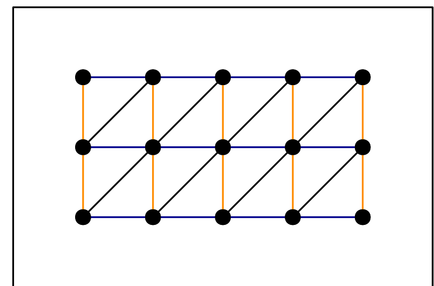
3.4. Quad Inner Tessellation

To construct the inner section of a quad, one can imagine the outer edges divided by the inner tessellation levels. Inner level two is used for outer edge one and three, and inner level one for outer edge two and four.

Perpendicular lines are then drawn from the edge segment borders. At the intersections the inner vertices are created, as seen in Figure 3.5a. The inner triangles are then created by taking two vertices of one row and one vertex of the next/previous row as seen in Figure 3.5b. This results in a simple grid like structure.



(a) Inner vertices

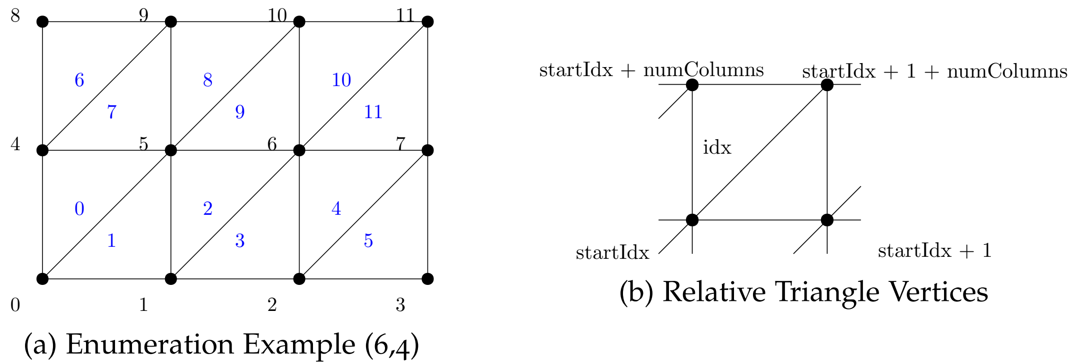


(b) Inner triangles

Figure 3.5.: Example of inner quad tessellation (inner1 = 6, inner2 = 4)

3.4.1. Specific Implementation

The enumeration for vertices and triangles is done in a procedural row-like manner. Generated vertices from the first to the second vertex, of the original primitive, are defined as the first row.



(a) Enumeration Example (6,4)

(b) Relative Triangle Vertices

Figure 3.6.: Triangle and Vertex Enumeration

Triangle indices

The indices calculation for the quad triangles in the inner section is trivial. Row and column index are calculated and based on these values the orienation of the triangle can be determined. Then the indices of the vertices on the surrounding vertex rows can be easily computed.

Algorithm 1 CalculateQuadInnerTriangle

1: $pointingDown \leftarrow idx \mod 2$
2: $row \leftarrow \frac{idx}{numColumns*2}$
3: $startIdx \leftarrow \frac{idx}{2} + row$
4: $vertex.x \leftarrow startIdx$
5: $vertex.y \leftarrow startIdx + (numColumns + 1) + 1 - pointingDown * (numColumns + 1)$
6: $vertex.z \leftarrow startIdx + (numColumns + 1) + pointingDown$

Vertices

For the new vertices the interpolation value is calculated by, first computing the normal and the short edge segment step size. After that the number of short and normal steps are determined. Adjustments are then made for the two special cases, where no outer tessellation exists on a side. The new vertices are computed by a simple linear interpolation between the original ones.

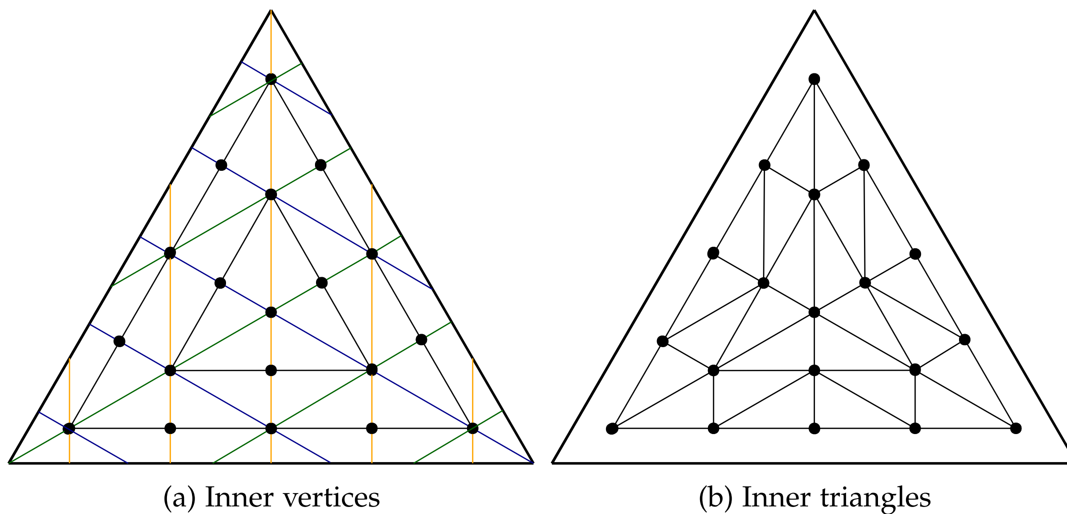
Algorithm 2 CalculateQuadEdgeFactor

1: $normalStep \leftarrow \frac{1.0}{inner}$
2: $shortStep \leftarrow normalStep * \frac{2-(convertedInner-inner)}{2}$
3: $numShorts \leftarrow (1 - (int)\frac{convertedInner-columnIdx}{convertedInner}) + (int)\frac{columnIdx+1}{convertedInner-1}$
4: $edgeStep \leftarrow normalStep * columnIdx + normalStep + numShorts * (shortStep - normalStep)$
5: **if** $convertedInner == 3$ **then**
6: $\quad edgeStep \leftarrow normalStep * columnIdx + shortStep$
7: **end if**
8: **if** $convertedInner == 1$ **then**
9: $\quad edgeStep \leftarrow columnIdx$
10: **end if**

3.5. Triangle Inner Tessellation

The inner section of a triangle is in principal done like the quad inner section. All outer edges are divided by the only inner tessellation level. Orthogonal lines are drawn from the division points. When we view the inner section as triangle rings, the corners of these inner triangle rings are given by the first intersection of the lines, that are drawn from the i-th outer subdivision, as seen in Figure 3.7a. The rest of the vertices are generated by the intersection of the lines drawn from the outer edge, with its corresponding inner ring triangle edge.

In this thesis, triangles are placed as seen in Figure 3.7b. On each of the three sides, triangles are placed symmetrically and the edges are oriented towards the center, when possible.



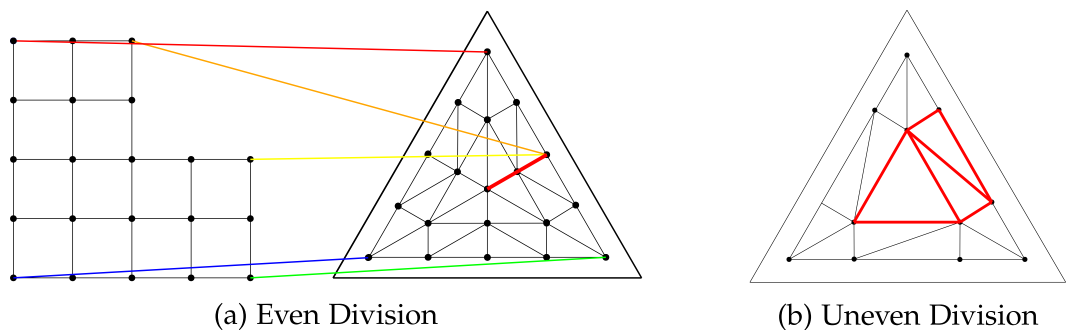
(a) Inner vertices

(b) Inner triangles

Figure 3.7.: Example for inner triangle tessellation (inner = 6)

3.5.1. Specific Implementation

For more efficient computability on the GPU, the inner section is viewed as a L-shaped grid, as shown in Figure 3.8a. The upper left vertex is the vertex closest to the third original vertex and the lower right vertex is the vertex closest to the second original vertex. This restructuring to a L-shaped-grid has benefits, because of data locality, when building subpatches. Subpatching will be further discussed in 4.3.2.



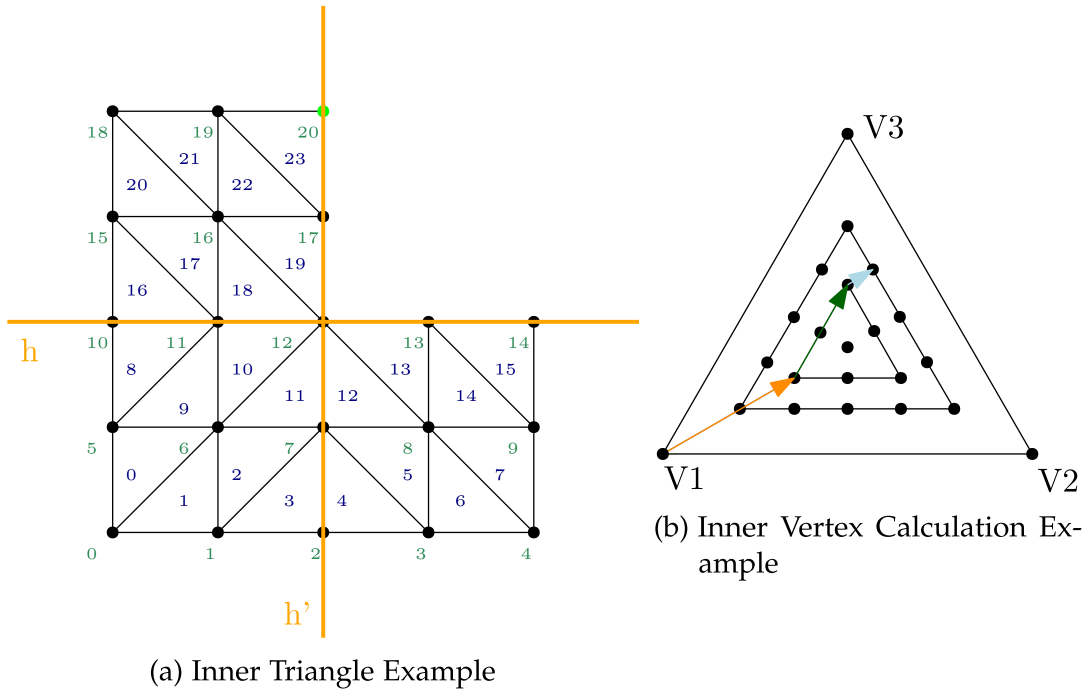
(a) Even Division (b) Uneven Division

Figure 3.8.: Vertex-Mapping

If the tessellation level is uneven, the L-form is preserved by splitting off a strip. This strip includes triangles in the middle of the side located between original vertex one and original vertex two, as seen in Figure 3.8b. In this case the implementation for the outer section is handling this strip. The enumeration of the vertices and the triangles starts at the lower left vertex, continuous to the lower right one and proceeds row-wise.

Triangle Indices (Algorithm: 3)

For the triangle primitive the procedure to generate inner triangle indices is only marginally more complex, than for quads. The difference is caused by the L-shape of the inner triangle tessellation, which results in a different calculation for the start index of the subdivided quad. If the subdivided quad is outside of the lower left section, either above h or to the right of h', the first triangle, in that quad, gets two vertices on the lower vertex row.



(a) Inner Triangle Example

(b) Inner Vertex Calculation Example

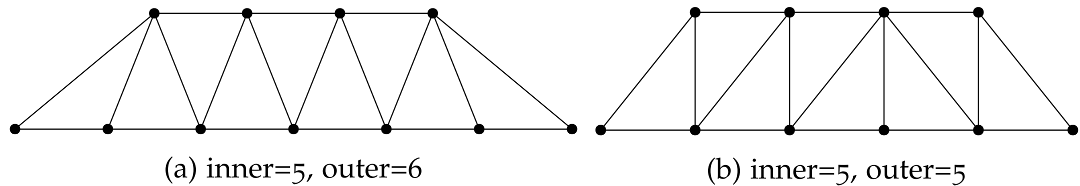
Figure 3.9.: Inner Tessellation Examples

Vertices (Algorithm: 4)

For the vertex interpolation values the calculation starts at $v1$ with steps to the center, as seen in Figure 3.9b. After that, steps on the current "ring" are added, either to the left$(v3 - v1)$ or to the right$(v2 - v1)$, until the line between $v2$ and the center or $v3$ and the center is reached. At last steps along the $\frac{v2+v3}{2} - center$ direction are taken.

3.6. Outer Tessellation

For the outer tessellation all sides of the primitive are generated in the same manner. The outer edge is split according to the corresponding outer tessellation level. Triangles with two vertices on the outer side (a), are connected symmetrically and equally across the inner vertices (b). Each inner vertex then has at least $k = floor(\frac{a}{b})$ triangles connected to it. Then the left and rightmost $\frac{l}{2}$ inner vertices have an additional triangle connected to it, where $l = a - k * b$. If l is uneven, the first inner vertex from the center to the right has another additional triangle connected to it. Triangles with two inner vertices fill the gaps, in a way such that triangles do not intersect and the whole area is covered.



(a) inner=5, outer=6 (b) inner=5, outer=5

Figure 3.10.: Outer Tessellation Examples

3.6.1. Specific Implementation

Triangle Indices (Algorithm: 5)

The outer tessellation is split into two symmetrical parts and a center part, that contains triangles, that could not be placed symmetrically. The special case of the inner tessellation level beeing one, is handled first. After that the middle section is computed.

The majority of the cases, where the triangle is on one of the sides, are computed afterwards. Therefore the triangle's side and its placement on that side are calculated. At last the triangles are generated, based on their orientation and the values calculated before.

Vertices (Algorithm: 6)

To avoid precision errors between corresponding vertices on edges shared between two primitives, the interpolation for the outer vertices is done in a coherent manner. The UVW calculation for the triangle is shown here. For the quad it is a similar process, but with four sides.

The calculation starts by determining on which edge the vertex is placed. After that the edge vertices are ordered coherently. This is done by flipping the edge, if v1's coordiantes are greater than v2's. The ordering could be based on different attributes, but coordinates are sure to exist.

Then the edge interpolation value, which interpolates between the edges vertices, and its opposite are calculated. If the edge was flipped before, then these values are swapped. At last, based on the edge interpolation values, the general interpolation values are calculated.

4. Implementation

This chapter gives an overview over the implementation and explains steps necessary to realize the tessellation procedure, explained in the previous chapter.

To show the differences between openGL and the CUDA implementation presented here, the first section will explain how tessellation in openGL works from the developers perspective. The stage implementation is explained afterwards, before looking at the individual stages.

4.1. A look at openGL

OpenGL lets the developer control the tessellation through the Tessellation Control Shader (TCS) and the Tessellation Evaluation Shader (TES). These two shaders are non-mandatory and can be skipped if tesselation is not required.

Figure 4.1.: OpengGL Tessellation Primitive Generation

In the TCS the developer chooses the tessellation levels. This shader is executed for every vertex of a triangle. An example can be seen in Figure 4.2. In this example the levels are set through uniform variables, that were

set outside the shader. Also the levels are only set if it is the first vertex of the triangle, that calls this shader.

```
#version 430


layout(vertices = 3) out;

uniform float TessellationInner;
uniform float TessellationOuter1;
uniform float TessellationOuter2;
uniform float TessellationOuter3;


void main()
{
  if( gl_InvocationID == 0 )
  {
    gl_TessLevelInner[0] = TessellationInner;
    gl_TessLevelOuter[0] = TessellationOuter1;
    gl_TessLevelOuter[1] = TessellationOuter2;
    gl_TessLevelOuter[2] = TessellationOuter3;
  }
 gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
}
```

Figure 4.2.: TessellationControlShader

The TES is then invoked for every old and every newly created vertex. Through the tessellation coordinate input the developer is able to interpolate the vertex attributes. An example for position interpolation can be seen in Figure 4.3. The shader also specifies the spacing type and the primitives, that are used.

```
#version 430

layout (triangles, equal_spacing, cw) in;

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

Figure 4.3.: TessellationEvaluationShader

This system limits the developer in several aspects. The triangulation and exact vertex placement can not be chosen by the developer. If we consider tessellations, that do not conform to the openGL specification many alterations would be possible.

4.2. Stage Implementation

Before the tessellation can be done, preceding stages process the primitives. These primitives are then handed to the Tessellation phase. The Tessellation phase is split into two main stages, the Tessellation Control Stage, and the Tessellation Evaluation Stage. One can see in Figure 4.4, the distribution of primitives across multiprocessors and after that the work redistribution, after the Tessellation Control Stage.

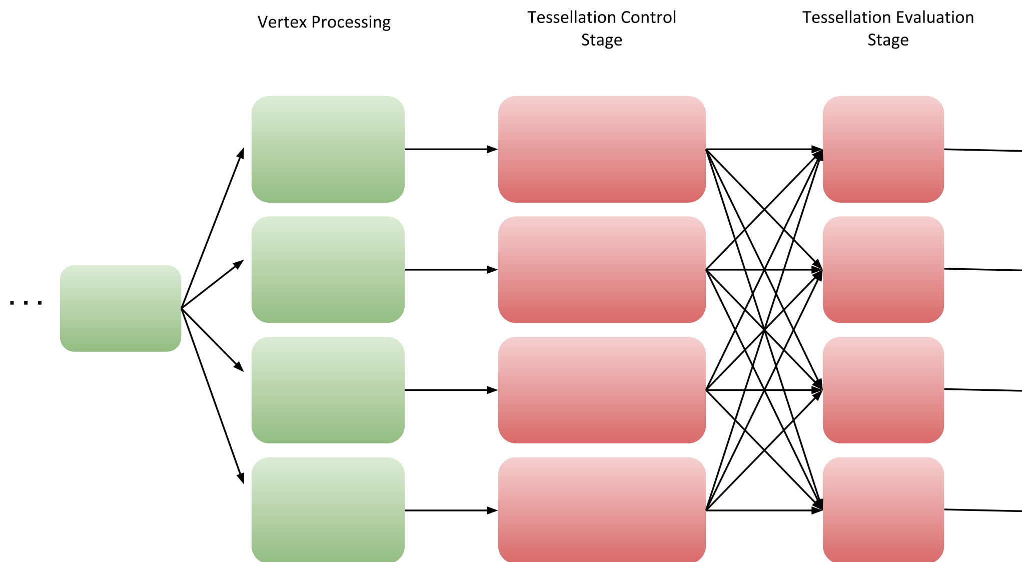


Figure 4.4.: Tessellation steps (red)

4.3. Tessellation Control Stage

The purpose of Tessellation Control Stage is to determine the tessellation levels and distribute work for the next stage, that constructs the new triangles.

In the control stage, a Tessellation Control Shader - function (TCS) is executed, ones for every primitive. The TCS usually sets the edge tessellation levels, based on a specific purpose. These parameters are then converted according to the edge tessellation conversion 3.1, to get processable levels. In addition, the number of subpatches is computed, to determine how much work is needed and therefore to how many multiprocessors the workload can be distributed. Further the number of vertices and triangles, that need to be generated, are calculated.

4.3.1. Number of Vertices and Triangles

Every thread theoretically executes the same code at the same time. Therefore one thread has in general no knowledge of how much data is generated by other threads. To store data in an array the place to store it needs to be determined, before the thread generates the data. Because of this the Tessellation Control Stage calculates the number of vertices and triangles. Prefix sums then serve to produce the offsets into the array for the kernel, that calculates the data, in the Tessellation Evaluation Stage.

4.3.2. Subpatch division

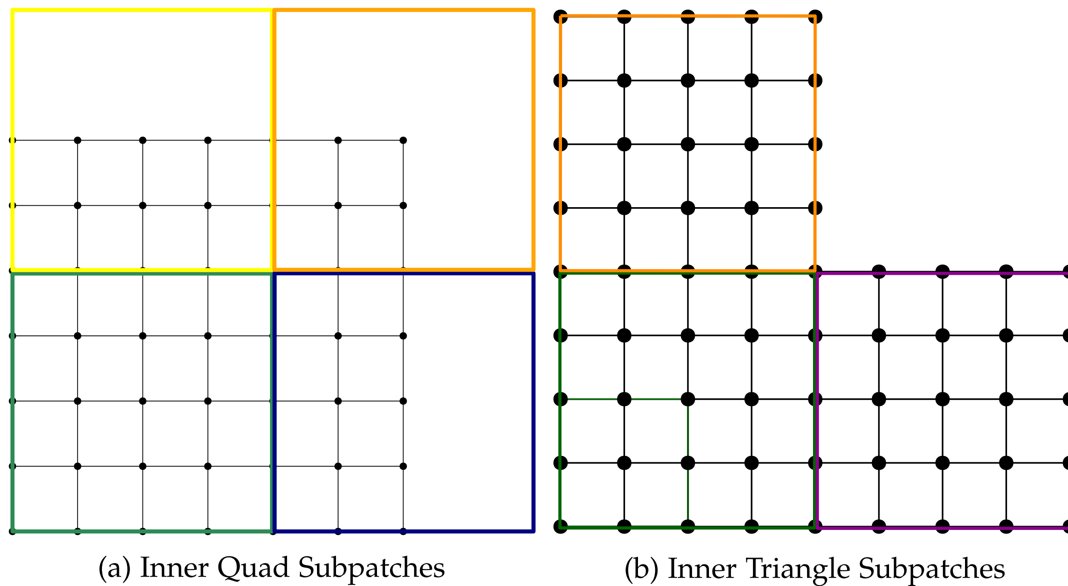
To efficiently compute the vertex coordinates and triangle indices after the Tessellation Control Stage, the workload is redistributed across the multiprocessors. It is necessary to create smaller patches, that can be computed individually. A maximum subpatch size is defined, to keep the number vertices/triangles, that need to be processed lower or equal to the number of threads handled by a multiprocessor. Subpatches are formed in a way, that all vertices and triangles, belonging to a subpatch can be computed on

a multiprocessor, the amount of triangles is maximized and no unessential vertex is calculated.

The division into subpatches is done by dividing the inner and outer section individually. This is because of a different layout and thus a different calculation procedure.

Inner Subpatch division

The inner segment is split into rectangles, because of a good triangle to vertex ratio. If the subpatch size is large enough, the number of triangles is the limiting factor. Figure 4.5a shows how the inner quad is divided and Figure 4.5b shows how the inner L-structure of the triangle is divided.



(a) Inner Quad Subpatches (b) Inner Triangle Subpatches

Figure 4.5.: Inner Subpatching

Outer subpatch division

Outer subpatches are formed by simply splitting the triangle strip. The limiting factor to the subpatch size are the number of vertices. Three vertices

are needed to form a basic triangle. If we add another triangle, we also need to add another vertex. This implies $numVertices = 2 + numTriangles$, which indicates a outer subpatch size of SUBPATCHMAX - 2.

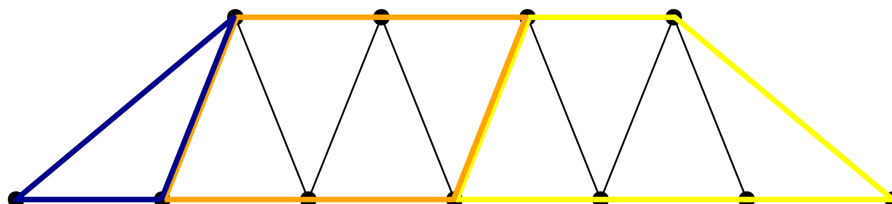


Figure 4.6.: Outer Subpatches

For triangles with uneven inner tessellation the leftover triangle strip, that was not covered with the L-shaped grid, is also handled like an outer subpatch. Because of this, the worst case for outer triangle subpatches are two interrupted triangle strips as seen in Figure 4.7 . Which concludes $numVertices = 4 + numTriangles$, and therefore a subpatch size of SUB-PATCHMAX - 4.

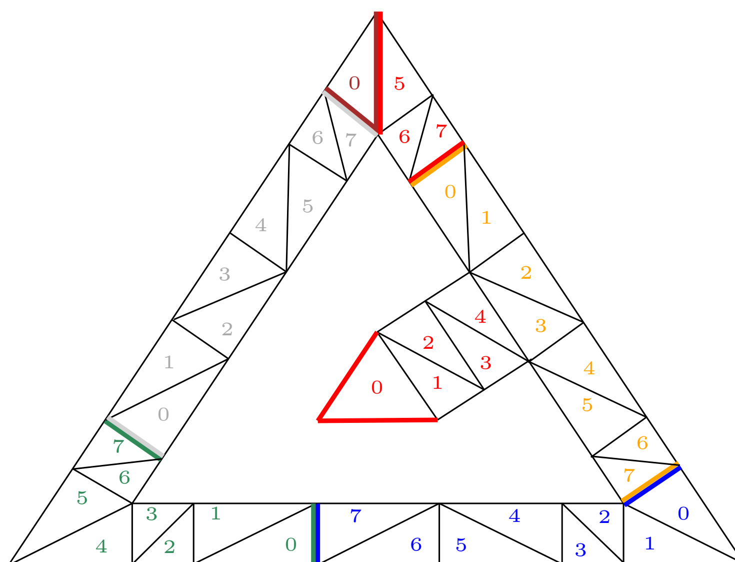


Figure 4.7.: Outer Triangle Subpatches

4.4. Tessellation Evaluation Stage

Each block in the tessellation evaluation stage generates a subpatch. Because of this, the global vertex index and the global triangle index are calculated first. Based on these indices, the vertex coordinates and the indices for the triangles are then calculated, according to the specification defined in the previous chapter. Following are the calculations of the global indices for the triangle indices and the vertices.

4.4.1. Global index calculations

Subpatch id, local(thread) id and the converted parameters are handed to the functions to determine the global indices. "Out of bounds"-checks are done to ensure, the validity of those parameters. The index calculations are mainly split into an outer and and inner part for both the triangle indices and the vertices, which are discussed below.

Global Triangle Index

To determine the triangle index, the subpatch offset is calculated and then added to the local offset. For the inner subpatches this means computing the position of the subpatch in the grid. From this position the offset of triangles for the subpatch is determined. After that, the same procedure is followed for the local id to produce the local offset. Local and global offset are then added and result in the global index.

The calculations for the triangle primitives are only marginally more complicated compared to the quad primitives. The L-structure causes a few more calculations to be needed, due to the changed subpatch grid, as seen in Figure 4.5b. Outer subpatches can be viewed as a triangle strip and therefore results in,

$$\begin{aligned} globalId = (subpatchId - numInnerPatches) * subpatchOuterSize \\ + id + numInnerTriangles \end{aligned} \tag{4.1}$$

For more detailed algorithms, see Algorithm 7 and Algorithm 9.

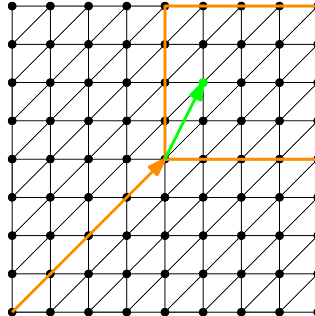


Figure 4.8.: Quad subpatch index calculation

Global Vertex Index

The calculation of the inner subpatch vertex indices is done similar to the triangle index calculation. The global index is a result of adding the local vertex offset to the subpatch vertex offset.

One needs to keep in mind, that a subpatch needs to generate all vertices for all triangles, that are produced by the subpatch. Outer subpatch indices are computed in a more sophisticated manner, than the inner ones, due to the reuse of inner vertices, as seen in Figure 4.9. Outer vertices are calculated first. After that, the outermost inner ring is traversed to generate the rest of the vertices.

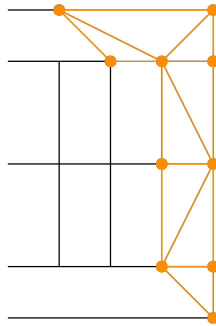


Figure 4.9.: Example of an outer subpatch

For both triangle and quad primitives, exist special cases, that need to be considered. If one of the inner tessellation levels is equal to one, one inner side of a quad vanishes. Therefore the global index for the inner vertices is calculated in a different way.

Unequal tessellation modes used on triangle primitives cause a strip beeing left out by the inner subpatches. The strip is generated by the first few outer subpatches, if necessary, which adds this special case to the vertex index generation. In this case, the left side and after that the right side of the strip are generated first.

For more detailed algorithms, see Algorithm 8 and Algorithm 10.

5. Evaluation

The test system used here, was running Windows10 and consisted of an Intel i7-4790K@4.00GHz CPU, 16 GB RAM and a Nvidia GTX1070. Cuda compute capability 6.1 and OpenGL version 4.6 were used.

The following sections show the results of the tessellation process. To get an overview over the differences to a specific openGL implementation a comparison is done afterwards. In the end the testing process is mentioned shortly.

5.1. Results

In this section the results of the tessellation process are shown. First the different inner tessellations for lower levels can be seen. Different models, at different tessellation levels, are listed after taking a look at subpatch examples.

5.1.1. Tessellation Results

Inner Triangle Tessellations

Figure 5.1 and Figure 5.2 show the inner tessellation from one to five, with a step size of 0.5 and outer tessellation level one, for the different tessellation types.

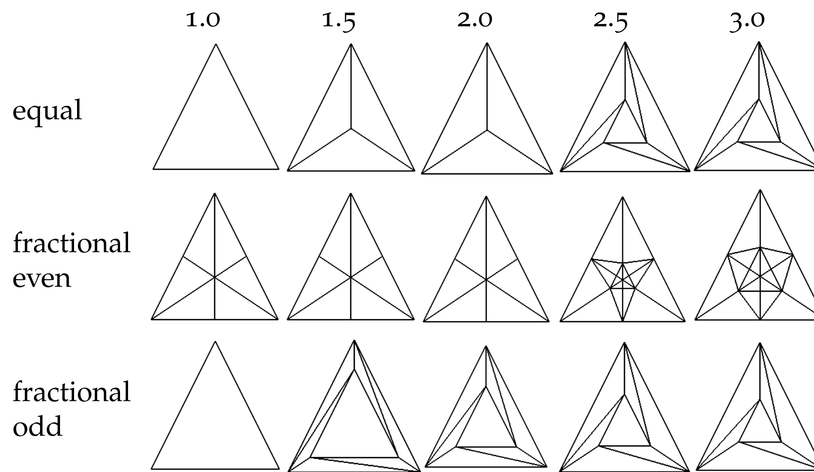


Figure 5.1.: Inner Tessellation 1 - 3 , 0.5 step

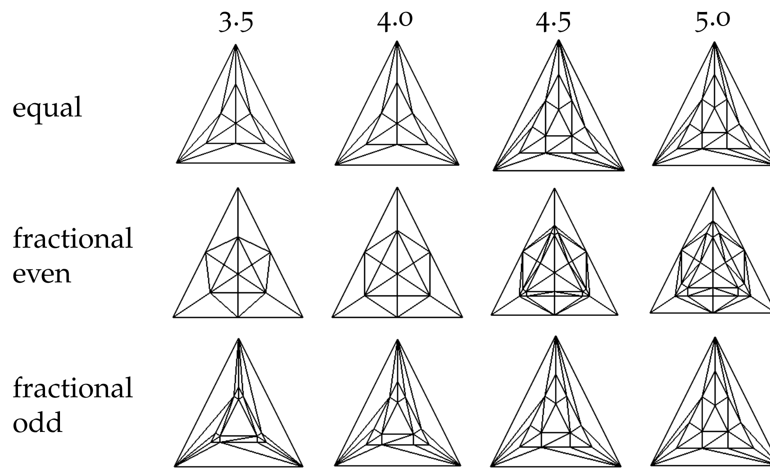


Figure 5.2.: Inner Tessellation 3.5 - 5 , 0.5 step

Inner Quad Tessellations

Shown here are inner quad tessellation for some lower tessellation levels. In Figure5.3 one can see equal tessellation and in Figure 5.4 fractional tessellation.

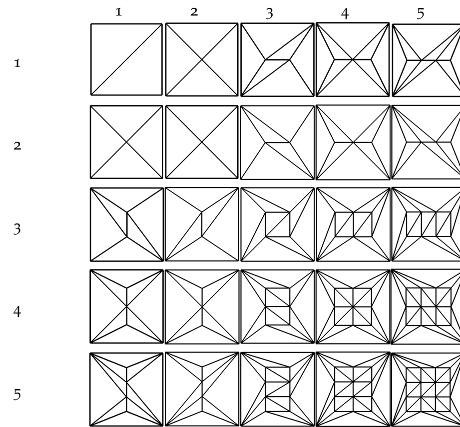


Figure 5.3.: Equal quad tessellation 1 - 5

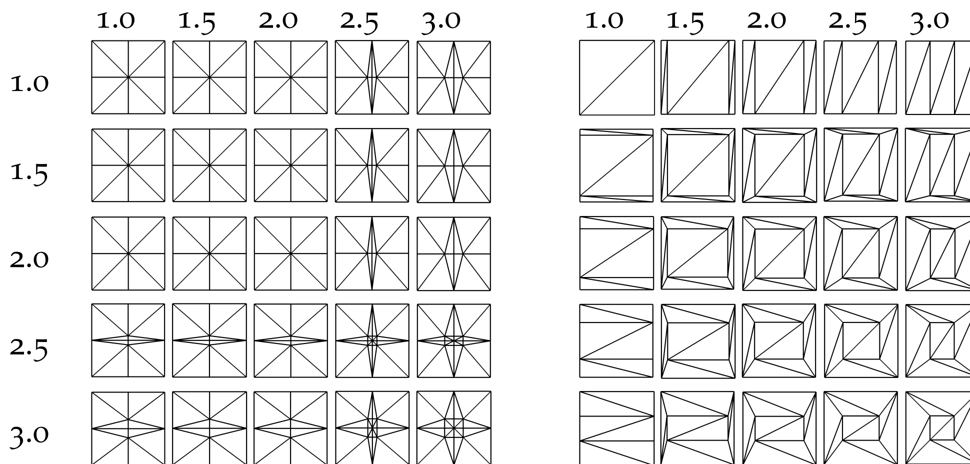
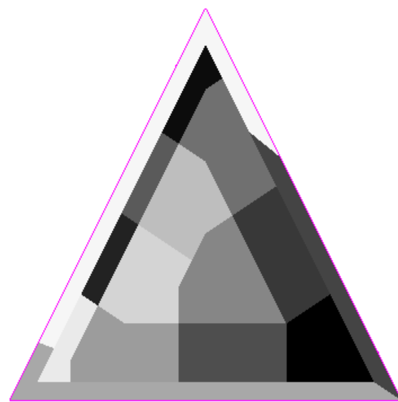


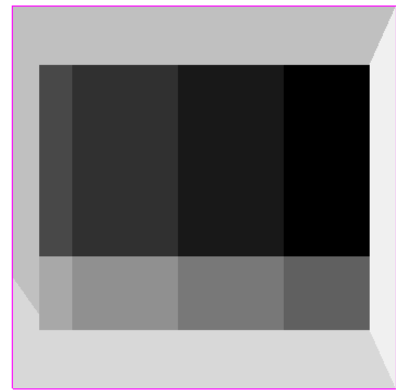
Figure 5.4.: Fractional even tessellation (left) and fractional odd tessellation (right) 1 - 3

5.1.2. Subpatches

Figure 5.5a and 5.5b show a visualization of the primitives being subdivided into subpatches. An example for the whole model can be seen in Figure 5.6a and Figure 5.6b.

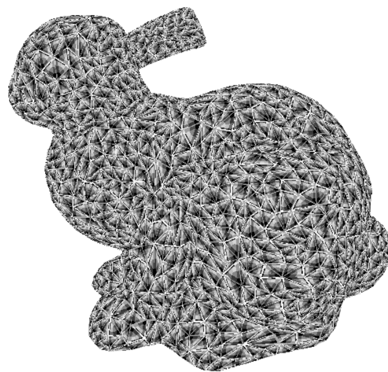


(a) Triangle Subpatches

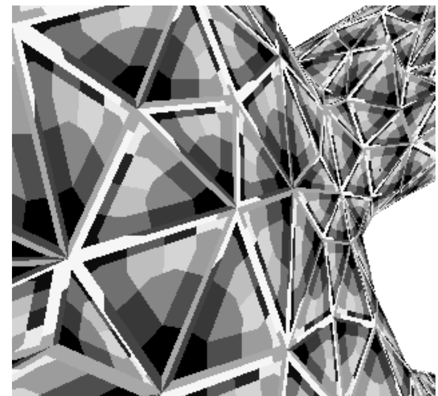


(b) Quad Subpatches

Figure 5.5.: Example Subpatches



(a) Subpatches shown for the bunny



(b) Close view of the subpatch

Figure 5.6.: Bunny Subpatches (inner=14.5,outers=8.5/8.25/9.5)

5.1.3. Models

Figure 5.8 shows the number of triangles, that are generated by some common test models shown in Figure 5.7. The execution time for these models is shown in Figure 5.9. One can see in figure Figure 5.10, that the execution time of the TES outweighs the execution time of the TCS, as expected. For models with a high number of triangles, tessellation level 64 was not possible, due to the memory consumption of the generated triangles and was therefore left out.

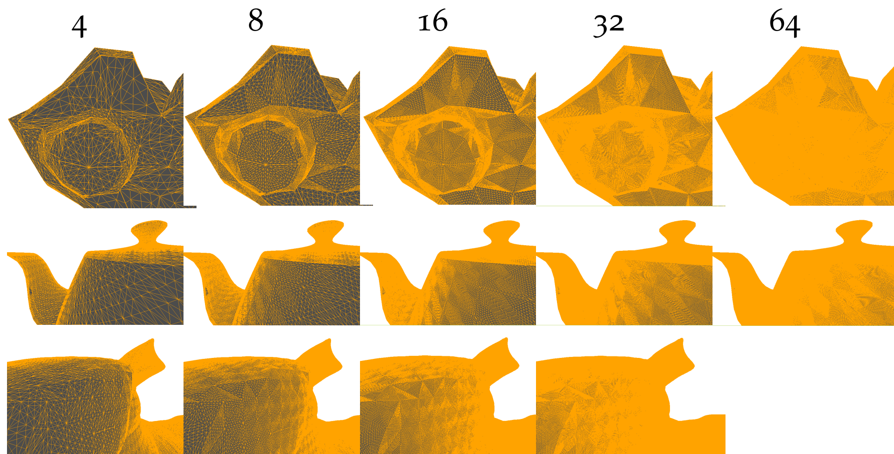


Figure 5.7.: Suzanne, Utah Teapot

	Suzanne	Utah Teapot	Stanford Bunny
No Tess	697	6320	69,630
TL 4	25,208	151,680	1,671,120
TL 8	92,832	606,720	6,684,480
TL 16	371,328	2,426,880	26,737,920
TL 32	1,485,312	9,707,520	106,951,680
TL 64	5,941,24	38,830,080	*427,806,720*

Figure 5.8.: Number of triangles for different tessellation levels. Note that TL 64 for the Stanford Bunny was not doable, due to memory limitation.

	Suzanne	Utah Teapot	Stanford Bunny
TL 4	3.4167	3.4155	11.5264
TL 8	3.5122	5.181	22.5825
TL 16	4.2880	9.4816	55.5980
TL 32	6.3266	25.0572	153.0488

Figure 5.9.: Execution time (ms) for common test models

	Suzanne	Utah Teapot	Stanford Bunny
TL 4	0.7947	0.4688	0.9889
TL 8	0.4644	0.5671	1.4340
TL 16	0.4681	0.8869	2.7621
TL 32	0.7871	1.3049	4.1788

Suzanne	Utah Teapot	Stanford Bunny
2.6220	3.1243	10.5375
3.0478	4.4863	21.1485
3.8200	7.6251	52.8359
5.5395	13.6864	148.8700

Figure 5.10.: Execution time (ms) for common test models. TCS(left), TES(right)

To get a better overview of how execution time behaves at different tessellation levels, with different amounts of triangles, sphere meshes are compared. Figure 5.11 shows the combined tessellation time, Figure 5.12 and Figure 5.12 the TCS and the TES execution times.

	S. 1	S. 2	S. 3	S. 4	S. 5	S. 6
Triangles	*960*	*3840*	*8640*	*15360*	*34560*	*61440*
TL 4	3.4566	3.6407	3.7252	4.8328	7.5002	10.6713
TL 8	3.6001	4.9040	5.8589	8.9251	12.9689	20.3741
TL 16	4.4136	6.8195	10.5898	15.4263	30.2714	47.9148
TL 32	6.5311	12.233	23.8071	37.9889	81.8656	135.0185

Figure 5.11.: Execution time (ms) for spheres with different numbers of triangles

	S. 1	S. 2	S. 3	S. 4	S. 5	S. 6
TL 4	0.8183	0.5622	0.4914	0.4892	0.5932	0.5797
TL 8	0.4839	0.5280	0.5044	0.5545	0.9212	1.2742
TL 16	0.4868	0.5453	0.5219	0.9753	1.4078	2.1398
TL 32	0.8454	0.9003	1.3219	1.5258	2.7776	3.2504

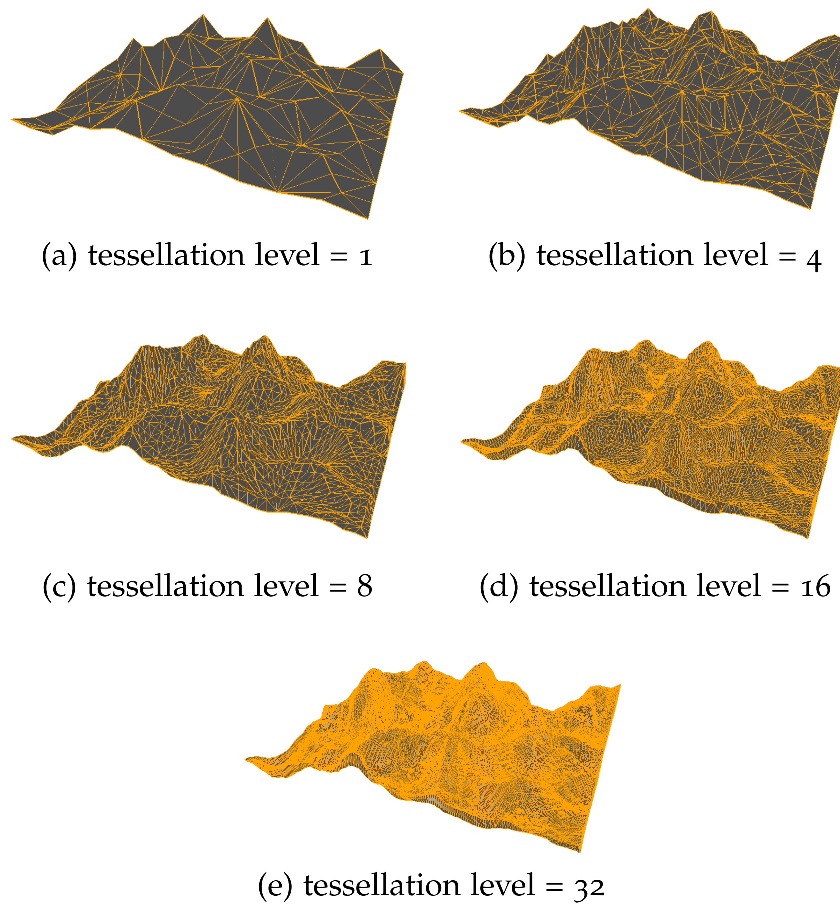
Figure 5.12.: TCS execution time (ms) for spheres

	S. 1	S. 2	S. 3	S. 4	S. 5	S. 6
TL 4	2.6382	3.0784	3.2338	4.3436	6.9070	10.0916
TL 8	3.1162	4.3760	5.3545	8.3706	12.0477	19.0999
TL 16	3.9268	6.2742	10.0679	14.4510	28.8636	45.7750
TL 32	5.6856	11.3333	22.4852	36.4631	79.0880	131.7680

Figure 5.13.: TES execution time (ms) for spheres

Height map

One practical example of Tessellation is a height map, as seen in Figure 5.14. Normals and texture coordinates are interpolated during the tessellation process along with the vertex coordinates. The texture coordinates are used to look up an offset value stored in a height map image. After that, the vertex coordinates are offset by the interpolated normal times the offset value.



(a) tessellation level = 1

(b) tessellation level = 4

(c) tessellation level = 8

(d) tessellation level = 16

(e) tessellation level = 32

Figure 5.14.: Heightmap Example

LOD

A very simple example for a LOD adaption is shown in 5.15. The TCS changes the inner tessellation level with the distance to a point, which is moved in from the camera to the lower vertex. In practical examples this point is usually the camera, to apply a finer tessellation only to the closest, most visible, regions.

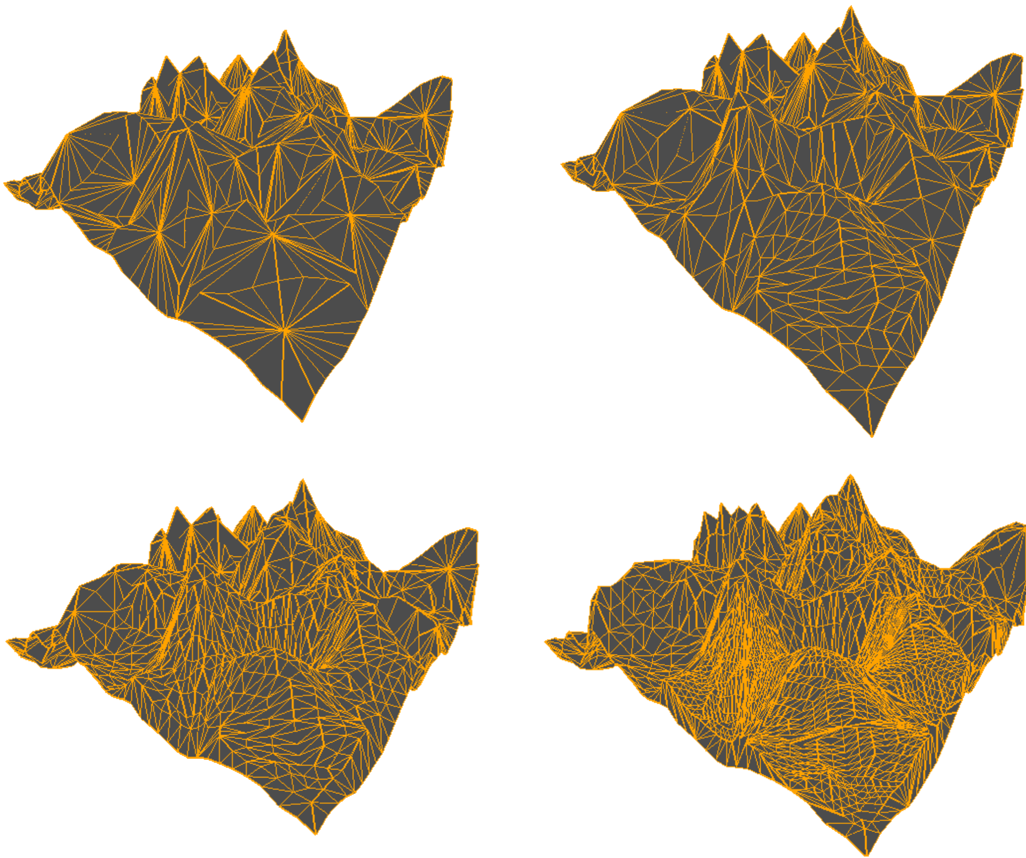


Figure 5.15.: LOD Example

5.2. Comparison

Due to the freedom that is given to the developer in some parts of the OpenGL specification, the results can vary between OpenGL implementations. This is why in general an exact comparison is difficult.

The main differences between this CUDA tessellation implementation and the OpenGL version, that is running on the test system are mentioned here. One difference is the edge placement. This OpenGL version starts placing short edges as the $k = floor(\frac{n}{2}) - 1$ edge, counted from the center outside. For every new placed edge, k is decremented by one until it reaches zero. After that the process starts anew, as seen in Figure 5.16.

2.5

4.5

6.5

8.5

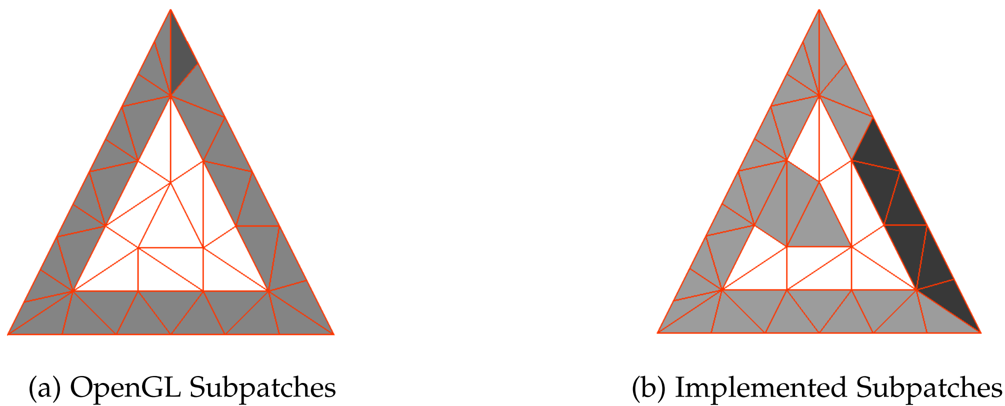
10.5

12.5

Figure 5.16.: Edge Tessellation Comparison

One can also see the difference in the edge lengths. The approach presented in this thesis uses simple division by the converted tessellation level to calculate the normal edge length, unlike the OpenGL version.

For efficiency reasons this behavior mentioned above was not imitated in this CUDA tessellation implementation. The same logic is the reason for the difference in the triangulation. Computing the vertices and triangle indices in a way to imitate OpenGL, would yield no obvious aesthetic improvement, but would complicate the program. This would obviously cause more branching and as an effect reduce efficiency for computations on the GPU.



(a) OpenGL Subpatches

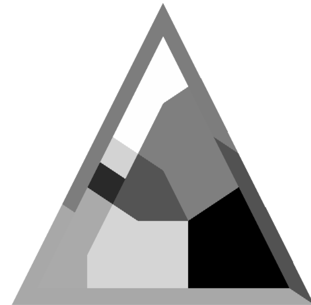
(b) Implemented Subpatches

Figure 5.17.: Subpatch Comparison

OpenGL also seems to split the subpatches for primitives into approximately the same sections. The difference is the strip that was added to the outer section in this implementation, when uneven tessellation levels are set, as described in Section 4.3.2. This allows OpenGL always to generate SUBPATCHMAX - 2 triangles for the outer subpatches. One can see in Figure 5.17 the different subpatching techniques for outer subpatches. The darker subpatch is the second outer subpatch, which is in the right case 5 triangles longer(3 for the strip + 2 for the subpatch length difference).

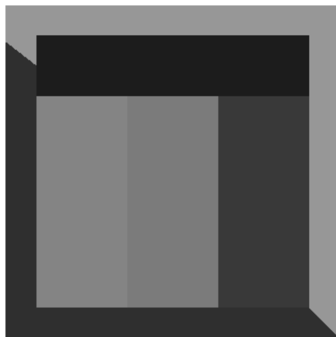
Figure 5.18 shows a comparison between triangles and Figure 5.19 between quads. For the subpatch visualization an extension(NV_shader_thread_group) of OpenGL was used. The streaming multiprocessor id is displayed as a grey value. The inner subpatching of primitives in OpenGL was not further analyzed in detail due to complexity. For the inner subpatches OpenGL chooses a maximum of 32 vertices (42 triangles).

(a) OpenGL Subpatches

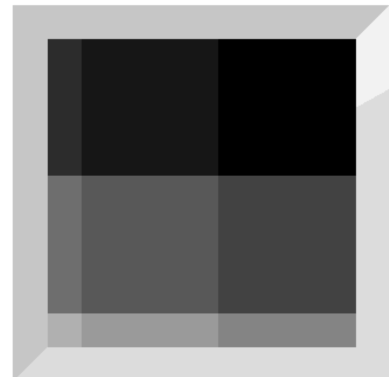


(b) Implemented Subpatches

Figure 5.18.: Subpatch Comparison for Triangles



(a) OpenGL Subpatches



(b) Implemented Subpatches

Figure 5.19.: Subpatch Comparison for Quads

5.3. Testing

To confirm that every triangle and every vertex are generated by the subpatches, all combinations of converted tessellation levels are tested and the result of all subpatches is checked for completeness.

Different operation execution orders cause different results in the last bits. Neighboring primitives share the same edge, but usually in inverted order,

due to the same winding order, as seen in Figure 5.20. Precision checks are done to confirm consistent calculation results.

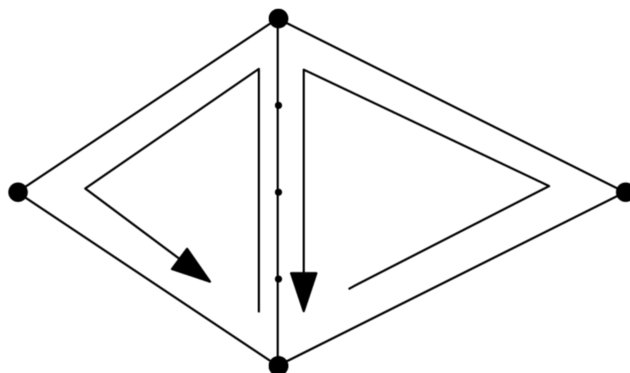


Figure 5.20.: Shared edge, different orientation

5.4. Conclusion

The thesis shows an implementation of a Tessellation stage, that can be integrated into a streaming software rendering pipeline. To make the process stream able, this thesis explains a way to subdivide and redistribute the work between stages.

This thesis also showed a way to implement the OpenGL tessellation procedure in CUDA. Because this implementation fulfills the OpenGL specification for tessellation, some not entirely efficient procedures for the GPU had to be implemented. Therefore some performance improvements could be made, if diverging from the OpenGL specification is acceptable.

The performance evaluation has shown, that this implementation is run-able in real time and therefore usable for the targeted pipeline. Some models, with a high number of primitives, show long run times, at high tessellation levels. One should keep in mind, that these cases generate a high amount of triangles, which may be not viable for real time applications.

This implementation for a streaming software rendering pipeline illustrates, that in general, stages, that implement standard hardware solutions, can be added to this kind of pipeline without loosing to much performance.

A measurement of the exact run time of the OpenGL tessellation yielded non conclusive results. In general due to the specific adaption of the hardware to the procedure, the API should outperform this solution.

Less performance is obviously not wanted, but this is compared to solutions for standard rendering. If we consider non-standard tasks, that require a special modification of the pipeline, the performance difference can be much different.

The next step would be to integrate this stage into the targeted pipeline. Tests on real real-time rendered scenes could be done. Different TES implementations could be tested on those scenes, to show the adaptability of software solutions to specific situations.

Appendix

Appendix A.

Algorithms

Algorithm 3 CalculateTriangleInnerTriangle

1: $pointingDown \leftarrow idx \mod 2$

2: $aboveH \leftarrow max(0, row - (numRings + 1))$
3: $inH \leftarrow min(numRings + 1, row)$
4: $aboveHPlusOne \leftarrow max(0, (row + 1) - (numRings + 1))$
5: $inHPlusOne \leftarrow min(numRings + 1, row + 1)$

6: $vStartLower \leftarrow inH * ((numRings + 1) * 2 - (1 - uneven))$
7: $\quad +aboveH * (numRings + 1) + column$
8: $vStartUpper \leftarrow inHPlusOne * ((numRings + 1) * 2 - (1 - uneven))$
9: $\quad +aboveHPlusOne * (numRings + 1) + column$

10: $vertex.x \leftarrow vStartLower$
11: $vertex.y \leftarrow vStartLower + 1$
12: $vertex.z \leftarrow vStartUpper + (!(row >= numRings || column >= numRings))$
13: **if** !PointingDown **then**
14: $\quad vertex.x \leftarrow vStartUpper$
15: $\quad vertex.y \leftarrow vStartUpper + 1$
16: $\quad vertex.z \leftarrow vStartLower + (row >= numRings || column >= numRings)$
17: **end if**

Algorithm 4 CalculateTriangleInnerVertexUVW

1: $u \leftarrow 1 + -\frac{2}{3} * (normalStepToCenter * (1 + numNormalStepsToCenterr)$
2: $\quad +shortStepToCenter * numShortStepsToCenter)$
3: $v \leftarrow \frac{1}{3} * (normalStepToCenter * (1 + numNormalStepsToCenterr)$
4: $\quad +shortStepToCenter * numShortStepsToCenter)$
5: $w \leftarrow \frac{1}{3} * (normalStepToCenter * (1 + numNormalStepsToCenterr)$
6: $\quad +shortStepToCenter * numShortStepsToCenter)$

7: $cap \leftarrow convertedInner - 2 - ringNr * 2$
8: $numRingShortSteps \leftarrow (\frac{1-(cap-ringSteps)}{cap} + \frac{ringSteps}{cap}) * (1 - min(1, ringNr))$
9: $numRingNormalSteps \leftarrow ringSteps - numRingShortSteps$
10: $u \leftarrow u + (-1 * (numRingNormalSteps * normalStep1$
11: $\quad +numRingShortSteps * shortStep1))$
12: $v \leftarrow v + ((numRingNormalSteps * normalStep1$
13: $\quad +numRingShortSteps * shortStep1)) * (!left_side)$
14: $w \leftarrow w + ((numRingNormalSteps * normalStep1$
15: $\quad +numRingShortSteps * shortStep1)) * (left_side)$

16: $numShortLeftSteps \leftarrow 0$
17: **if** $ringNr \geq 0$ **then**
18: $\quad numShortLeftSteps \leftarrow \frac{leftSteps}{ringNr}$
19: **end if**
20: $numNormalLeftSteps \leftarrow max(leftSteps - numShortLeftSteps, 0)$

21: $u \leftarrow u + (-\frac{2}{3} * (numNormalLeftSteps * normalStep1$
22: $\quad +numShortLeftSteps * shortStep1))$
23: $v \leftarrow v + (\frac{1}{3} * (numNormalLeftSteps * normalStep1$
24: $\quad +numShortLeftSteps * shortStep1))$
25: $w \leftarrow w + (\frac{1}{3} * (numNormalLeftSteps * normalStep1$
26: $\quad +numShortLeftSteps * shortStep1))$

Algorithm 5 CalculateOuterTriangle

1: $innerVerticesSide \leftarrow max(1, \downarrow (\frac{numberInnerVertices}{2}))$
2: $outerSegmentsSide \leftarrow \downarrow (\frac{convertedOuter-outerUneven}{2})$
3: $innerSegmentsSide \leftarrow \downarrow (\frac{numberInnerSegments-innerUneven}{2})$
4: $sec1EndIdx \leftarrow outerSegmentsSide + innerSegmentsSide$
5: $sec2StartIdx \leftarrow outerSegmentsSide + innerSegmentsSide + innerUneven + outerUneven$

// *Special cases*
6: **if** $(convertedInner1 == 1) \wedge (convertedInner2 == 1 || isTriangle)$ **then**

7: $facingSide \leftarrow 1$
8: $vertexIdx.x \leftarrow 0$
9: $vertexIdx.y \leftarrow 1$
10: $vertexIdx.z \leftarrow 0$
11: **return**
12: **end if**

// *Compute the middle section if it exists*
13: **if** $idx \geq sec1EndIdx \wedge idx < sec2StartIdx$ **then**
14: $facingSide \leftarrow 0$
15: $vertexIdx.x \leftarrow innerVerticesSide$
16: $vertexIdx.y \leftarrow innerVerticesSide - 1$
17: $vertexIdx.z \leftarrow outerSegmentsSide$
18: **if** $idx - sec1EndIdx \geq InnerUneven$ **then**
19: $facingSide \leftarrow 1$
20: $vertexIdx.x \leftarrow outerSegmentsSide$
21: $vertexIdx.y \leftarrow outerSegmentsSide + 1$
22: $vertexIdx.z \leftarrow innerVerticesSide - (convertedInner == 2)$
23: **end if**
24: **return**
25: **end if**

// *Left or right side*
26: $betaFac \leftarrow \frac{idx}{sec2StartIdx}$
27: $perInner \leftarrow \frac{outerVerticesSide}{innerVerticesSide}$
28: $perInnerAdditionals \leftarrow outerVerticesSide \mod innerVerticesSide$

// *position of the last additional on local side*
29: $lambda \leftarrow (perInner + 2) * perInnerAdditionals$ // *side idx*
30: $aidx \leftarrow ((innerSegmentsSide + convertedOuter) - idx - 1) * betaFac + idx * (1 - betaFac)$
31: $numLocalInnerVerts \leftarrow (innerVerticesSide) * betaFac$
32: $numLocalOuterVerts \leftarrow (convertedOuter + 1) * betaFac$
33: $lowIdx \leftarrow (aidx - lambda) \mod (perInner + 1)$
34: $sector \leftarrow \frac{aidx - lambda}{perInner + 1} + perInnerAdditionals$
35: $isLower \leftarrow (lowIdx == perInner)$

// *tri is placed before last additional*
36: **if** $aidx < lambda$ **then**
37: $lowIdx \leftarrow aidx \mod (perInner + 2)$
38: $sector \leftarrow \frac{aidx}{perInner + 2}$
39: $isLower \leftarrow lowIdx == (perInner + 1)$
40: **end if**

```
    // pointing outside
41: if isLower then
42:     facing ← 0
43:     vertIdx.x ← (numLocalInnerVerts − 1 − sector) ∗ betaFac + (sector + 1) ∗ (1 − betaFac)
44:     vertIdx.y ← (numLocalInnerVerts − 1 − (sector + 1)) ∗ betaFac
45:                 +(sector) ∗ (1 − betaFac)
46:     onLeftSide ← (sector + 1) ∗ perInner
47:     onLeftSide+ = fminf(sector + 1, perInnerAdditionals)
48:     vertIdx.z ← (numLocalOuterVerts − 1 − onLeftSide) ∗ betaFac
49:                 +onLeftSide ∗ (1 − betaFac)
50:     if vertIdx.z == numInnerVerts + numOuterVerts then
51:         vertIdx.z ← numInnerVerts
52:     end if
53: else
54:     facing ← 1
55:     lTIdx ← sector ∗ perInner + min(sector, perInnerAdditionals) + lowIdx
56:     vertIdx.x ← (numLocalOuterVerts − 1 − (lTIdx + 1)) ∗ betaFac + lTIdx ∗ (1 − betaFac)
57:     vertIdx.y ← (numLocalOuterVerts − 1 − lTIdx) ∗ betaFac + (lTIdx + 1) ∗ (1 − betaFac)
58:     vertIdx.z ← (numLocalInnerVerts − 1 − sector) ∗ betaFac + sector ∗ (1 − betaFac)
59: end if
```

Algorithm 6 CalculateOuterTriangleVertexUVW

```
    // Calclulate the outer index
 1: cidx ← minOuter + cur − numInnerTriangleVertices

    // if its the last its actually the first
 2: if cidx == numOuterVertices then
 3:     cidx ← 0
 4: end if
 5: cDir ← max(0, cidx − convertedOuter1 − convertedOuter2)
 6: bDir ← max(0, cidx − convertedOuter1 − cDir)
 7: aDir ← max(0, cidx − bDir − cDir)
 8: side ← (aDir == convertedOuter1) + (bDir == convertedOuter2)

    // which edge?
 9: eV1 ← v1 ∗ (side == 0) + v3 ∗ (side == 1) + v2 ∗ (side == 2)
10: eV2 ← v3 ∗ (side == 0) + v2 ∗ (side == 1) + v1 ∗ (side == 2)
11: eI ← aDir ∗ (side == 0) + bDir ∗ (side == 1) + cDir ∗ (side == 2)
12: maxE ← outer1 ∗ (side == 0) + outer2 ∗ (side == 1) + outer3 ∗ (side_= = 2)
13: maxEC ← convertedOuter1 ∗ (side == 0) + convertedOuter2 ∗ (side == 1) + convertedOuter3 ∗
    (side == 2)

    // Order the vertices coherently
```

```
14: if (eV1.x > eV2.x then
15:     swap(eV1, eV2)
16:     eI ← maxEC − eI
17: end if
18: if (eV1.y > eV2.y) then
19:     swap(eV1, eV2)
20:     eI ← maxEC − eI
21: end if

    // Calculate number of shorts
22: dirShorts ← (eI − 1 > 0)
23: dirShorts+ = (eI − maxEC + 1 + 1 > 0)
24: if maxEC == 3 ∧ eI > 0 ∧ eI < 3 then
25:     dirShorts ← 1
26: end if

    // Calculate the value and the opposite one
27: val ← (2.0−(maxEC−maxE))/2.0 ∗ dirShorts + max(0, eI − dirShorts)
28: if maxE == 1 then
29:     val ← eI
30: end if
31: val ← val/maxE
32: antiVal ← 1 − val
33: if !swappedVertices then
34:     swap(val, antiVal)
35: end if


36: u ← (side == 0) ∗ val + (side == 2) ∗ antiVal
37: v ← (side == 0) ∗ antiVal + (side == 1) ∗ val
38: w ← (side == 1) ∗ antiVal + (side == 2) ∗ val
```

Algorithm 7 getGlobalQTriangleID

1: $globalId = (subpatchId - numInnerPatches) * subpatchOuterSize$
2: $\quad +id + numInnerTriangles$
3: **if** $globalId \geq numTriangles \vee (id \geq subpatchSize)$ **then**
4: **return** -1
5: **end if**
6: **if** $subpatchId \geq numInnerPatches \wedge (id \geq subpatchOuterSize)$ **then**
7: **return** -1
8: **end if**

// *If its an inner subpatch*
9: **if** $subpatchId < numInnerPatches$ **then**
10: $subpatchI \leftarrow subpatchRow * subpatchesSize2 * numInnerQuadSegmentsOnSide * 2$
11: $\quad +subpatchColumn * subpatchesSize1 * 2$
12: $localI \leftarrow localRow * numInnerQuadSegmentsOnSide * 2 + localColumn$
13: $globalId \leftarrow subpatchI + localI$
14: **if** $globalColumn \geq numInnerQuadSegmentsOnSide * 2$ **then**
15: **return** -1
16: **end if**
17: **if** $globalRow \geq numInnerQuadSegmentsOnSidePlusOne$ **then**
18: **return** -1
19: **end if**
20: **if** $(localColumn \geq subpatchSizeX * 2)$
21: $\quad \vee(localRow \geq subpatchSizeY)$ **then**
22: **return** -1
23: **end if**
24: **return** $globalId$
25: **end if**

Algorithm 8 getGlobalQVertexID

1: **if** $id > subpatchSize$ **then**
2: **return** -1
3: **end if**

// *First handle outer subpatch*
4: $globalId \leftarrow minOuterVertex + id$

5: **if** $id > numPatchOuterVerts$ **then**
6: $tessSide1Exists \leftarrow 1$
7: $tessSide2Exists \leftarrow 1$
8: **if** $convertedInner1 == 1$ **then**
9: $tessSide1Exists \leftarrow 0$

10: **end if**
11: **if** $convertedInner2 == 1$ **then**
12: $tessSide2Exists \leftarrow 0$
13: **end if**

14: $convId \leftarrow minInnerIdx + id - numPatchOuterVerts - 1$

15: $dDir \leftarrow max(0, convId - innerSegs1 * 2 - innerSegs2)$
16: $cDir \leftarrow max(0, convId - innerSegs1 - innerSegs2 - dDir)$
17: $bDir \leftarrow max(0, convId - innerSegs1 - cDir - dDir)$
18: $aDir \leftarrow max(0, convId - bDir - cDir - dDir)$

19: $globalId \leftarrow aDir + bDir * inVerts1 - cDir - dDir * inVerts1$

20: **if** $(!tessSide1Exists)||(!tessSide2Exists)$ **then**
21: $globalId \leftarrow$
22: $((convId \geq 0) * (innerVerts1 - 1)$
23: $+(convId > 0) * innerVerts1$
24: $-(convId > 1) * innerVerts1 - 1)$
25: $-(convId > 2) * innerVerts1) * (1 - tessSide2Exists)+$
26: $((convId > 0)$
27: $+(convId > 1) * ((innerVerts2) * (innerVerts1) - 2)$
28: $-(convId > 2)) * (1 - tessSide1Exists)$
29:
30: **end if**

31: **if** $globalId < 0$ **then**
32: $globalId \leftarrow -1$
33: **end if**

34: **else**
35: **if** $globalId == numInnerVertices + numOuterVertices$ **then**
36: $globalId \leftarrow numInnerVertices$
37: **end if**
38: **end if**

39: **if** $(globalId \geq numVertices) \wedge (subpatchId \geq numInnerPatches)$ **then**
40: $globalId \leftarrow -1$
41: **end if**

// Handle inner subpatch
42: **if** $(subpatchId < numInnerPatches)$ **then**
43: $subpatchI \leftarrow subpatchRow * subpatchesSize2 * numInnerQuadVerticesOnSide$
44: $+subpatchColumn * subpatchesSize1$

45: $localIndex \leftarrow localRow * numInnerVerticesSide + localColumn$

46: $globalId \leftarrow localIndex + subpatchI$

47: **if** $globalColumn > numInnerQuadVerticesOnSide$ **then**
48: **return** -1
49: **end if**
50: **if** $globalRow > numInnerQuadVerticesOnSidePlusOne$ **then**
51: **return** -1
52: **end if**
53: **if** $(localColumn > (subpatchesSize1 + 1)) \vee (localRow > (subpatchesSize1 + 1))$ **then**
54: **return** -1
55: **end if**
56: **end if**

57: **return** $globalId$

Algorithm 9 getGlobalTTriangleID

1: $mSizeX \leftarrow numRingSegments * 2 + uneven$
2: $mSizeY \leftarrow numRingSegments$
3: $hX \leftarrow\uparrow \frac{mSizeX}{subpatchSizeX}$
4: $hY \leftarrow\uparrow \frac{mSizeY}{subpatchSizeY}$
5: $left \leftarrow mSizeX - hY * subpatchSizeY$
6: $nX \leftarrow\uparrow \frac{mSizeY}{subpatchSizeX}$
7: $nY \leftarrow\uparrow \frac{left}{subpatchSizeY}$

8: $localRealColumn \leftarrow \frac{localColumn}{2}$

9: $comRow \leftarrow subpatchRow * subpatchSizeY + localRow$
10: $comColumnPatch \leftarrow subpatchColumn * subpatchSizeX * 2 + localColumn$
11: $comColumnReal \leftarrow subpatchColumn * subpatchSizeX + localRealColumn$

12: $aboveH \leftarrow max(0, comRow - numVertexRings)$
13: $inH \leftarrow min(numVertexRings, comRow)$
14: $aboveTri \leftarrow max(0, comRow - numRingSegments)$
15: $inTru \leftarrow min(numRingSegments, comRow)$

16: **if** $subpatchId < numInnerSubpatches$ **then**
17: **if** $comColumnReal \geq mSizeX \vee comRow \geq mSizeX$ **then**
18: **return** -1

```
19:     end if
20:     if comRow ≥ mSizeY ∧ comColumnReal ≥ mSizeY then
21:         return −1
22:     end if
23:     return inTru ∗ mSizeX ∗ 2 + aboveTri ∗ mSizeY ∗ 2 + comColumnPatch
24: else
25:     if id ≥ subpatchesOuterSize then
26:         return −1
27:     end if

28:     cidx ← (subpatchId − numInnerSubpatches) ∗ subpatchesOuterSize + id
29:     if cidx ≥ numOuterTriangles then
30:         return −1
31:     end if


32:     return numInnerTriangles + cidx
33: end if
```

Algorithm 10 getGlobalTVertexID

1: $mSizeX \leftarrow numRingSegments * 2 + uneven$
2: $mSizeY \leftarrow numRingSegments$
3: $hX \leftarrow\uparrow \frac{mSizeX}{subpatchSizeX}$
4: $hY \leftarrow\uparrow \frac{mSizeY}{subpatchSizeY}$
5: $left \leftarrow mSizeX - hY * subpatchSizeY$
6: $nX \leftarrow\uparrow \frac{mSizeY}{subpatchSizeX}$
7: $nY \leftarrow\uparrow \frac{left}{subpatchSizeY}$

8: $comColumn = subpatchStartX + localColumn$
9: $comRow = subpatchStartY + localRow$

```
10: if subpatchId < numInnerSubpatches then
11:     if convertedInner1 == 3 ∧ id < 3 then
12:         return id
13:     end if
14:     if comColumn ≥ (2 ∗ numVertexRings − (!uneven))∨
15:                 comRow ≥ (2 ∗ numVertexRings − (!uneven)) then
16:         return −1
17:     end if
18:     if comRow ≥ numRingVertices ∧ comColumn ≥ numVertexRings then
19:         return −1
20:     end if
```

// *Number of steps below h + steps above h (L-structure)*

21: **return** $min(numVertexRings, comRow) * (numVertexRings * 2 - (!uneven))$
22: $+max(0, comRow - numRingVertices) * numVertexRings + comColumn$
23: **else**

// *Calculate the leftover part if we have uneven tessellation*
24: $innerLeft \leftarrow uneven * numSectorRings * 2 + uneven$
25: $k \leftarrow numInnerTriangles + innerLeft - 1$
26: $toDoNow \leftarrow min(max(k - startTriangle + 1, 0), subpatchOuterSize) * uneven$
27: $doneBefore \leftarrow startTriangle - numInnerTriangles$
28: $toDoVertices \leftarrow (toDoNow + 2) * uneven$
29: $toDoVertices \leftarrow toDoVertices$
30: $+(startTriangle == numInnerTriangles \wedge toDoNow > 0)$
31: $toDoVertices \leftarrow toDoVertices + (doneBefore > 0) * uneven$
32: $doneVerticesSideL \leftarrow \frac{doneBefore}{2} * (doneBefore > 0)$
33: $doneVerticesSideR \leftarrow \frac{doneBefore}{2} * (doneBefore > 0)$
// *Height of L-Structure*
34: $o \leftarrow (numVertexRings * 2 - (!uneven))$

// *Generate the leftover part if necassary*
35: **if** $toDoNow > 0 \wedge id < toDoVertices$ **then**
36: $gidx \leftarrow o * numRingSegments + numVertexRings - 1$
37: **if** $id == 0 \wedge startTriangle == numInnerTriangles$ **then**
38: **return** $gidx$
39: **end if**
40: **if** $convertedInner1 == 3$ **then**
41: **if** $id < 3$ **then**
42: **return** id
43: **end if**
44: **return** -1
45: **end if**
46: **if** $convertedInner1 == 2$ **then**
47: **if** $id! = 0$ **then**
48: **return** -1
49: **end if**
50: **return** 0
51: **end if**

// *Generate left and right leftover side*
52: **if** $id \leq \frac{toDoVertices-1}{2}$ **then**
53: $rIdx \leftarrow doneVerticesSideL + id$
54: $gidx \leftarrow gidx + rIdx$

```
55:         return gidx
56:     else
57:         rIdx ← doneVerticesSideR + id
58:         gidx ← gidx(numVertexRings − (!uneven))
59:                 +(rIdx − (toDoVertices−1)/2) ∗ numVertexRings
60:         return gidx
61:     end if
62: end if

// Generate the normal outer ring

63: if toDoNow > 0 then // set the startTriangle to the first outer triangle
64:     startTriangle ← numInnerTriangles + innerLeft − 1
65: end if

66: cur ← id − toDoVertices

67: if toDoNow == 0 then
68:     cur ← id
69: end if
70: numHVertices ← (numVertexRings ∗ 2 − (!uneven)) ∗ numVertexRings

// Generate outer vertices
71: if cur <= numPatchOuterVertices then
72:     cidx ← minOuter + cur − numInnerTriangleVertices
73:     cidx ← cidx − (convertedInner1 == 2 ∧ convertedOuter1 == 1
74:                 ∧convertedOuter2 == 1 ∧ convertedOuter3 == 1)
75:     if cidx == numOuterVertices  then
76:         cidx ← 0
77:     end if

78:     if cidx < numOuterVertices then
79:         return cidx + numInnerTriangleVertices
80:     end if
81:     return −1
82: else

// Generate inner vertices
83:     cur ← cur − numPatchOuterVertices + 1

84:     if convertedInner1 == 3 then
85:         if id < 3 then
86:             return id
87:         end if
```

88: **return** -1
89: **end if**
90: **if** $convertedInner1 == 2$ **then**
91: **if** $id! = 0$ **then**
92: **return** -1
93: **end if**
94: **return** 0
95: **end if**

96: $inA \leftarrow (startColumn == 0) \wedge (minInnerVertex! = o - 1)$
97: $inC \leftarrow (startRow == 0) * (!inA)$
98: $inB \leftarrow (!inA) * (!inC)$
99: $bSteps \leftarrow startColumn$
100: **if** $startColumn == o - 1$ **then**
101: $bSteps \leftarrow (numVertexRings - startRow) + numVertexRings - 2$
102: **end if**

// How far am I on the current side
103: $inIdx \leftarrow startRow * inA + bSteps * inB + (o - startColumn - 1) * inC$
104: $cidx \leftarrow inIdx + cur$
105: $crossed \leftarrow \frac{cidx}{o-1}$

// Calc new side after current thread correction
106: $rSide \leftarrow inB + inC * 2 + crossed$
107: $cidx \leftarrow cidx \mod (o - 1)$
108: **if** $rSide == 3 \wedge cidx == 0$ **then**
109: $rSide \leftarrow 0$
110: **end if**
111: **if** $rSide > 2 \wedge cidxS! = 0$ **then**
112: **return** -1
113: **end if**

// Determine the steps in each direction
114: $aDir \leftarrow (rSide == 1) * (o - 1) + (rSide == 12 * (o - 1)$
115: $+(rSide == 0) * cidx$
116: $bDir \leftarrow (rSide == 2) * (o - 1) + (rSide == 1) * cidx$
117: $cDir \leftarrow (rSide == 2) * cidx$
118: $aFac \leftarrow \frac{aDir}{o-1}$
119: $bFac \leftarrow \frac{bDir}{o-1}$
120: $cFac \leftarrow \frac{cDir}{o-1}$

121: $b2StepsF \leftarrow max(0, cidx - numRingSegments$
122: $+(!uneven) * aFac * bFac * (!cFac))$
123: $b1StepsF \leftarrow cidx - b2StepsF$

```
124:        b2StepsFFac ← b2StepsF > 0

    // Calculate index by adding side offset to the "on side steps"
125:        inCorr ← aFac * numInnerSegments + bFac * (numRingSegments * o
126:                   +(numRingSegments − 1 + uneven) * numVertexRings + 1)
127:        gidx ← inCorr+
128:                  cidx * (!aFac) * (!bFac) * (!cFac)
129:                  +b1StepsF * o * aFac * (!bFac) * (!cFac)
130:                  +(b2StepsFFac * numRingSegments * (numVertexRings − (!uneven))
131:                  +b2StepsFFac − b2StepsF
132:                  −(!uneven) * b2StepsFFac) * aFac * (!bFac) * (!cFac)
133:                  −b1StepsF * numVertexRings * aFac * bFac * (!cFac)
134:                  −b2StepsF * o * aFac * bFac * (!cFac)
135:        if gidx >= numInnerVertices + numOuterVertices then
136:            return −1
137:        end if
138:        return gidx
139:    end if
140: end if
```

Appendix B.

Lists

List of Algorithms

1. CalculateQuadInnerTriangle 17
2. CalculateQuadEdgeFactor 18

3. CalculateTriangleInnerTriangle 51
4. CalculateTriangleInnerVertexUVW 52
5. CalculateOuterTriangle 52
6. CalculateOuterTriangleVertexUVW 54
7. getGlobalQTriangleID 56
8. getGlobalQVertexID 56
9. getGlobalTTriangleID 58
10. getGlobalTVertexID 59

List of Figures

1.1. Tessellation illustration 2
1.2. Subpatching illustration 2
1.3. Nvidia Thread-Block Structure 3
1.4. Nvidia Memory Model 4

2.1. Traditional Pipe 8
2.2. Scheduling approaches 8

3.1. Special edge division levels are shown in a, and some normal cases in b, . Red indicates short segments and orange, normal segments 13
3.2. Inner and outer section of a triangle 14
3.3. Inner and outer section of a quad 14
3.4. Visualization of an outer section part of a quad and a triangle 15
3.5. Example of inner quad tessellation (inner1 = 6, inner2 = 4) . 16
3.6. Triangle and Vertex Enumeration 17
3.7. Example for inner triangle tessellation (inner = 6) 19
3.8. Vertex-Mapping 20
3.9. Inner Tessellation Examples 21
3.10. Outer Tessellation Examples 22

4.1. OpengGL Tessellation Primitive Generation 25
4.2. TessellationControlShader 26
4.3. TessellationEvaluationShader 26
4.4. Tessellation steps (red) 27
4.5. Inner Subpatching 29
4.6. Outer Subpatches 30
4.7. Outer Triangle Subpatches 30
4.8. Quad subpatch index calculation 32

4.9. Example of an outer subpatch 33

5.1. Inner Tessellation 1 - 3 , 0.5 step 36
5.2. Inner Tessellation 3.5 - 5 , 0.5 step 36
5.3. Equal quad tessellation 1 - 5 37
5.4. Fractional even tessellation (left) and fractional odd tessellation (right) 1 - 3 37
5.5. Example Subpatches 38
5.6. Bunny Subpatches (inner=14.5,outers=8.5/8.25/9.5) 38
5.7. Suzanne, Utah Teapot 39
5.8. Number of triangles for different tessellation levels. Note that TL 64 for the Stanford Bunny was not doable, due to memory limitation. 39
5.9. Execution time (ms) for common test models 40
5.10. Execution time (ms) for common test models. TCS(left), TES(right) 40
5.11. Execution time (ms) for spheres with different numbers of triangles 40
5.12. TCS execution time (ms) for spheres 41
5.13. TES execution time (ms) for spheres 41
5.14. Heightmap Example 42
5.15. LOD Example 43
5.16. Edge Tessellation Comparison 44
5.17. Subpatch Comparison 45
5.18. Subpatch Comparison for Triangles 46
5.19. Subpatch Comparison for Quads 46
5.20. Shared edge, different orientation 47

Bibliography

[1] Timo Aila and Samuli Laine. "Understanding the efficiency of ray traversal on GPUs." In: *Proceedings of the conference on high performance graphics 2009*. ACM. 2009, pp. 145–149 (cit. on p. 8).

[2] Tamy Boubekeur and Christophe Schlick. "A flexible kernel for adaptive mesh refinement on GPU." In: *Computer Graphics Forum*. Vol. 27. 1. Wiley Online Library. 2008, pp. 102–113 (cit. on p. 9).

[3] Tamy Boubekeur and Christophe Schlick. "Generic mesh refinement on GPU." In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. ACM. 2005, pp. 99–104 (cit. on p. 9).

[4] NVIDIA Corporation. *CUDA C Programming Guide*. 2018. URL: `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf` (cit. on p. 2).

[5] Michael Kenzel et al. "A high-performance software graphics pipeline architecture for the GPU." In: *ACM Transactions on Graphics (TOG)* 37.4 (2018), p. 140 (cit. on p. 9).

[6] Samuli Laine and Tero Karras. "High-performance software rasterization on GPUs." In: *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. ACM. 2011, pp. 79–88 (cit. on p. 7).

[7] Fang Liu et al. "FreePipe: a programmable parallel rendering architecture for efficient multi-fragment effects." In: *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*. ACM. 2010, pp. 75–82 (cit. on p. 7).

[8] Kurt Akeley Mark Segal. *The OpenGL Graphics System: A Specification (Version 4.0 (Core Profile) - March 11, 2010)*. 2010. URL: `https://www.khronos.org/registry/OpenGL/specs/gl/glspec40.core.pdf` (cit. on p. 9).

[9] Kurt Akeley Mark Segal. *The OpenGL Graphics System: A Specification (Version 4.6 (Core Profile) - July 30, 2017)*. 2017. URL: http://https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf (visited on 04/03/2018) (cit. on pp. 11, 12).

[10] Microsoft. *Direct3D 11 Graphics*. 2009. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ff476342(v=vs.85).aspx (visited on 04/03/2018) (cit. on p. 9).

[11] Matthias Nießner et al. "Feature-adaptive GPU rendering of Catmull-Clark subdivision surfaces." In: *ACM Transactions on Graphics (TOG)* 31.1 (2012), p. 6 (cit. on p. 9).

[12] Matthias Nießner et al. "Real-Time Rendering Techniques with Hardware Tessellation." In: *Computer Graphics Forum*. Vol. 35. 1. Wiley Online Library. 2016, pp. 113–137 (cit. on p. 10).

[13] Anjul Patney et al. "Piko: A Design Framework for Programmable Graphics Pipelines." In: *arXiv preprint arXiv:1404.6293* (2014) (cit. on p. 7).

[14] Michael Schwarz and Marc Stamminger. "Fast GPU-based adaptive tessellation with CUDA." In: *Computer Graphics Forum*. Vol. 28. 2. Wiley Online Library. 2009, pp. 365–374 (cit. on p. 9).

[15] Larry Seiler et al. "Larrabee: a many-core x86 architecture for visual computing." In: *ACM Transactions on Graphics (TOG)* 27.3 (2008), p. 18 (cit. on p. 7).

[16] Markus Steinberger et al. "Whippletree: task-based scheduling of dynamic workloads on the GPU." In: *ACM Transactions on Graphics (TOG)* 33.6 (2014), p. 228 (cit. on p. 8).

[17] Pixar Animation Studios. *OpenSubdiv*. 2013. URL: http://graphics.pixar.com/opensubdiv/docs/intro.html (cit. on p. 9).