DEGREE PROJECT, IN COMPUTER SCIENCE , SECOND LEVEL
STOCKHOLM, *SWEDEN 2015*

Surface data on dynamic topologies

A REAL TIME TECHNOLOGY FOR TOPOLOGY INDEPENDENT STORAGE OF SURFACE DATA

ANDREAS TARANDI

KTH ROYAL INSTITUTE OF TECHNOLOGY

COMPUTER SCIENCE AND COMMUNICATION

Surface data on dynamic topologies

ANDREAS TARANDI
taran@kth.se

JANUARY 2015

Master of Science in Engineering
Computer Science and Technology

School of Computer Science and Communication
Royal Institute of Technology
Supervisor at CSC was Mario Romero
Examiner was Olle Bälter

Problem was provided by EA DICE
Supervisor at DICE was Jan Schmid

Ytdata på dynamiska topologier

Abstract

We implement, combine, and evaluate the work by H.Schäfer (2012) in *"Multiresolution attributes for tessellated meshes"* and C.Yuksel (2010) in *"Mesh Colors"* for storing surface data such as colors, normals and displacement directly on the surface for 3D geometry in a topology-independent way. We provide implementation details missing in the original papers and propose and evaluate possible optimizations to the techniques, with a focus on usage in real-time applications.

The technique of storing data directly on the surface provides advantages over the commonly used texture mapping such as the eliminations of discontinuities and seams. The proposed technique is also useful on dynamic topologies where texture mapping is difficult.

We also evaluate the performance of these techniques in a real time application where they perform resonably well, but not better than traditional techniques, leading to the conclusion that they are only suitable to use in situation where traditional techniques fall short.

Referat

Ytdata på dynamiska topologier

Vi implementerar, kombinerar och evaluerar algoritmerna föreslagna av H.Schäfer (2012) i *"Multiresolution attributes for tessellated meshes"* och av C.Yuksel (2010) i *"Mesh Colors"* för att lagra ytdata så som färg, normaler och deformering direkt på ytan för 3D geometri oberoende av dess topologin. Vi tillför detaljer kring implementeringen som saknades i de ursprungliga texterna och föreslår och evaluerar möjliga optimeringar av dem, med fokus på realtidsapplikationer.

Att lagra datan obeorende av geometrins topologi har vissa föredelap getemot den traditionella metoden texturmappning, bland annat elimineringen av sömmar och diskontinuiteter. Den föreslagna algoritmen är också användbar på dynamiska topologier där texurmappning är svårt.

Vi utvärderar också algoritmernas prestanda i en realtidsapplikation där de presterar acceptabelt, men inte bättre än de traditionella metoderna, vilket leder till slutsatsen att de endast är rimliga att använda i de fall då det inte är rimligt att använda traditionella metoder.

Preface

This is a Master's Thesis at the School of Computer Science and Communication at KTH. The problem was provided by EA DICE and the project was executed there.
I would like to thank my supervisor at KTH, Mario Romero for his support throughout my work with the thesis. I would also like to thank my supervisors at DICE; Jan Schmid, Torbjörn Söderman and Christopher Birger, as well as the other employees at DICE who made me feel welcome. It has truly been a great experience.

Contents

Chapter 1

Introduction

This chapter will give an introduction to the problem, as well as providing an overview of the field and previous work done on the problem.

1.1 Background

1.1.1 Texture mapping

Mapping of textures onto surfaces have since the early days of computer graphics been the preferred nonprocedural way of providing detail to a geometric surface [Catmull, 1974]. This technique has the inherent problem of requiring a map for a 3D surface onto the 2D domain. If one allows disjunct islands and nonuniform distribution in the 2D mapping, the problem can be solved for all 3D geometries with only minimal seams where the islands connect [Akenine-Möller et al., 2008].

1.1.2 Tessellation

A relatively recent feature in graphics hardware is programmable tessellation. It provides a way of subdividing the geometry on the GPU (*graphics processing unit*) with full control of the location of the newly generated vertices. Tessellation affords CPU (*central processing unit*) algorithms to run on a coarser base mesh and reduces the load on the memory bus to the GPU, while maintaining or improving the granularity of the rendered mesh.

Tessellation adds two programmable shader stages to the graphics pipeline: the hull-shader and the domain-shader. In addition to this a fixed stage, the actual tessellator, is added. The hull-shader is run per patch (a polygonal face) and controls the input parameters to the tessellator and does per-patch calculation. The domain-shader is run once for each point on the output patch and its purpose is to generate a vertex for the point [Microsoft, 2013a].

A common way of encoding the extra detail in the mesh geometry is storing it as a texture (commonly known as a displacement map), and mapping it to the surface in the same way as a color texture [NVIDIA, 2010]. Due to the intrinsic problems with seams in texture mapping, this may result in holes or ridges in the mesh, which in many cases is unacceptable.

1.1.3 Barycentric coordinates

Barycentric coordinates are triples of numbers $\{\alpha, \beta, \gamma\}$ corresponding to weights placed at the vertices of a triangle [Weisstein, 2014]. The sum of the weights α, β, and γ is 1.0. They are used in tessellation to provide the relative position of a newly generated vertex within a triangle.

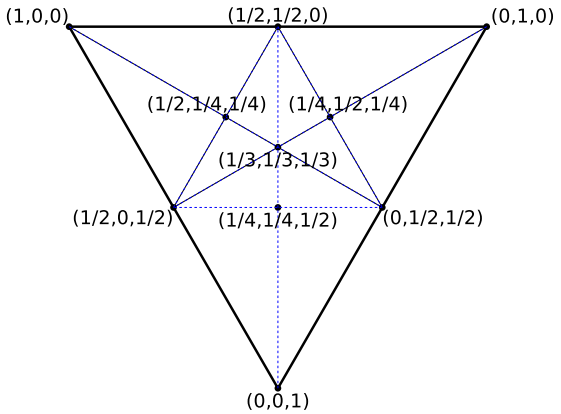

Figure 1.1: Barycentric coordinates $\{\alpha, \beta, \gamma\}$ on a equilateral triangle [Wikipedia, 2008].

1.1.4 Related work

Multiple solutions to handling seams in displacement maps have been previously explored.

Piponi and Borshukov proposed a technique for finding an optimal and intutive mapping called *"pelting"*, combined with a method for blending texture mappings together, to solve the problem [Piponi and Borshukov, 2000]. Nießner and Loop proposed a technique using a tiled texture format and a displacement function, but it relies on a static topology and thus doesn't solve the issue of texture mapping a dynamic topology [Nießner and Loop, 2013].

Sander et al. proposed a solution to texture mapping progressive meshes by using texture atlases, which solves some of the texture stretch issues, but not the problem with dynamic topologies [Sander et al., 2001].

Yuksel et al. proposed a way of storing surface color details directly associated with the 3D geometry surface, avoiding entirely the problem of mapping between texture and model space [Yuksel et al., 2010]. Schäfer et al. later extended this, proposing a similar technique for handling displacement data [Schäfer et al., 2012]. Both these methods provide a one to one mapping between surface data and topology, and are thus feasable to use on dynamic topologies. We will further explore the techniques presented in these papers in this thesis.

1.2 Problem definition

In this thesis we will go through the process of implementing and combining the algorithm by Yuksel et al. with the algorithm by Schäfer et al. to provide a technique that is completely free from texture mapping while still providing both color and geometry details [Yuksel et al., 2010] [Schäfer et al., 2012]. The original papers lack important details required to implement the algorithms. We will provide the details of our implementation here.

Schäfer et al. proposed in their *"Limitations and Future Work"* section that if the hardware tessellator could provide a unique index per generated vertex, the costly index retrieval algorithm could be removed [Schäfer et al., 2012]. We have experimented with removing the index retrieval algorithm by taking advantage of the hardware available today, namely Compute Shaders, that can perform general purpose computing on GPUs (GPGPU). This also eliminates another limitation mentioned by Schäfer et. al in *"Limitations and Future Work"*, that current hardware only supports tessellation factors up to 64. It may also increase performance on AMD hardware where tessellation performance in general is not on par with NVIDIA hardware, while usually outperforming NVIDIA hardware when it comes to GPGPU (General Purpose GPU).

1.2.1 Limitations

We will not consider the problem of compressing the data on disk or the actual generation of the surface data. For testing purposes, we will convert texture mapped surfaces to our data format, but we will not be concerned with creating a signal optimal approach such as the one used by H.Schäfer et al., since our primary purpose for the algorithm is to directly work with the new data format, and not conversion from existing textures.

1.3 Purpose

The main purpose behind evaluating these techniques, beside the obvious advantage of eliminating discontinuities and distortions caused by texture mapping, is the usage on a dynamic topology.

Simulated or hand-crafted dynamic topology mesh morphing can be used to improve aesthetics and realism. For these techniques to have applications in games or other real-time systems, a replacement for texture mapping is required. Animation of the texture mapping is infeasible when it contains discontinuities, as is the case in complex models.

Since these techniques stores the surface data directly related to the topology, moving triangles around on the surface does not require additional work to keep the surface data intact. Furthremore, adding geometry does not raise further challenges, given that the only required update is the insertion of the data for the new faces.

A secondary advantage of these techniques is that they could allow artist to directly paint on

the mesh. This could be useful even if one end up baking the data to a traditional texture in the end.

In addition to these advantages, these techniques could also be useful in the case of procedurally-generated meshes and other cases of programmatically processed meshes.

We have chosen to evaluate these techniques since they have the potential to be used in real-time applications, and work well for dynamic topologies.

1.4 Performance measurement

We carry out all performance measurements in this thesis using DirectX Queries with results in milliseconds. This metric provides the actual process time on the GPU.

Chapter 2

Implementation

Here we provide the details of the implementation of the techniques, including problems and solution for overcoming them.

2.1 The tessellator pattern

One of the main problems we faced implementing H.Schäfer's multiresolution algorithm was to imitate the hardware tessellator on the CPU. This is required to get the exact barycentric coordinates for the vertices generated by the hardware tessellator. In turn, this precision supports the generation of correct surface data and provides the base for our Compute Shader implementation of tessellation.

Unfortunately, Microsoft and the hardware providers are restrictive with the implementation details of their tessellators, and the best description found was the one provided by F. Giesen in *"A trip through the Graphics Pipeline 2011, part 12: Tessellation"* [Giesen, 2011]. From Giesen we deduced the following method, which exactly imitates the hardware tessellator for even tessellation factors.

For edges, the tessellation is obvious; for a tessellation factor T_E the edge is divided into T_E equally large parts. For faces, on the other hand, the subdivision is less obvious; the face tessellation first divides the triangle into a number of rings. For a tessellation factor T the number of rings are $\frac{T}{2}$. For even tessellation factors the innermost ring is a degenerate triangle.

Then, for each ring, the three corner vertices are calculated using θ_r, φ_r (equation (2.1)) (for ring r where $1 \leq r \leq rings$).

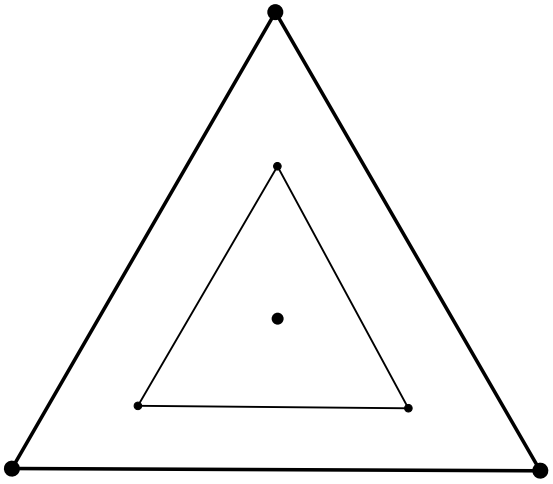

Figure 2.1: A triangle with tessellation level 4 with two rings; one normal and one degenerate triangle.

$$\begin{aligned} \theta_r &= \frac{1}{3} + (\frac{2}{3} * \frac{1}{rings})(rings - r) \\ \varphi_r &= \frac{1 - \theta_r}{2} \end{aligned} \tag{2.1}$$

The three vertices $\{\alpha, \beta, \gamma\}$ building the inner triangle are:

$$\begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = \begin{pmatrix} \theta_r & \varphi_r & \varphi_r \\ \varphi_r & \theta_r & \varphi_r \\ \varphi_r & \varphi_r & \theta_r \end{pmatrix} \tag{2.2}$$

That is, each corner vertex is built from one primary component (θ_r) and two smaller components (φ_r) of equal size. Since the barycentric coordinates must sum up to one, the two smaller components are easily calculated from the primary one.

The primary component is calculated as the distance in rings from the center of the triangle $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$. The distance from an outer triangle

corner to the center is $\frac{2}{3}$ (since the center is at component value $\frac{1}{3}$), and all rings are equally spaced from each other, thus the formula for θ_r.

Each edge on the ring is then subdivided into $T - 2r$ equal big parts. The hardware tessellator takes the tessellation factors for the edges in the order $u == 0, v == 0, w == 0$. That is in terms of our vertices $\beta \to \gamma, \gamma \to \alpha, \alpha \to \beta$, so to ensure consistency we always process the edges in this order. Thus we generate the barycentric coordinates for the vertices accordingly.

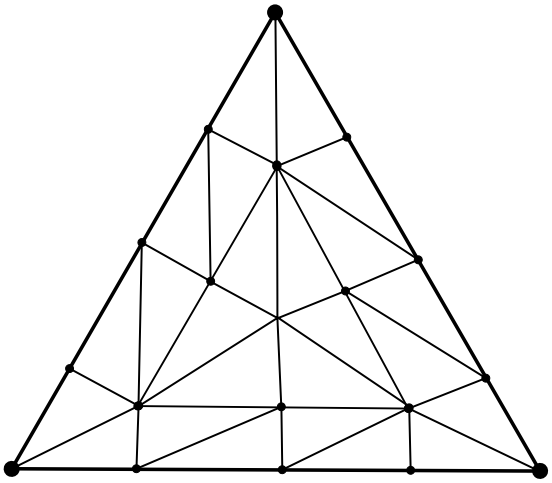

Figure 2.2: A triangle tessellated with tessellation level 4 on all edges and its face.

2.2 Displacement surface data

We have implemented *"Multiresolution attributes for tessellated meshes"* by Schäfer et. al. but limited to triangle primitives [Schäfer et al., 2012]. The technique generates data for the same positions generated by the hardware tessellator, thus eliminating the need for filtering the data.

Since the fixed GPU tessellator only outputs the barycentric coordinates for the generated vertices, the first step of the domain shader is to convert these coordinates into something more useful. Schäfer et. al proposes the use of a ring-, edge- and vertex-index (henceforward known as REV), since these are relatively painless to calculate from the barycentric coordinates. This conversion is well covered by H.Schäfer et al. and is thus not repeated here. The REV coordinates are each relative to their current scope, that is ring to the current face, edge to the current ring, and vertex to the current edge.

2.2.1 On face index given REV

The next step is to find the location in the point data buffer for the current REV. For outer edges and the three original vertices this is a straightforward usage of the V-index (for outer edges) or E-index (for original vertices), but the vertices on the face requires a bit more attention. Moreover, the original paper lacks the details of this implementation.

From the V-formula in the original paper (equation (2.3)), we can deduce that the number of indices on any edge on ring R is $(T-2R)$, giving the formula in equation (2.4) for the number of indices on a ring R (T is the tessellation factor, in this case the inner).

$$V = \frac{b'_E}{\alpha + \beta + \gamma - 3b_{min}}(T - 2R) \tag{2.3}$$

$$indices(R) = 3(T - 2R) \tag{2.4}$$

The unique index relative to the face for any vertex is then the number of vertices in all the previous rings (starting at R_1, since R_0 is the edge ring) plus the number of vertices in all the previous edges on the current ring, plus the current vertex index (equation (2.5)).

$$\begin{aligned} I(R, E, V) = &\sum_{r=1}^{R-1} indices(r) \\ &+ E(T - 2R) + V \end{aligned} \tag{2.5}$$

By using the formula for the sum of an arithmetic progresssion, followed by reduction this can be simplified to equation (2.6) [Wikipedia, 2014a].

$$\begin{aligned} I(R, E, V) = &3(R - 1)(T - R) \\ &+E(T - 2R) + V \end{aligned} \tag{2.6}$$

Precalculating the data for the point buffer

Next, we need to have the actual displacement data in the point buffer. For this, we calculate the barycentric coordinates for all vertices with the method described in section 2.1. These values are sent to an algorithm generating data for that position, either through texture lookup or some procedural algorithm.

We precompute and store the new normals for the vertex. Care should be taken when rendering to use the original normal when applying the displacement, and not the generated one. Since generating the data takes time for complex meshes, we generate and store it in a simple file structure in a preprocess step.

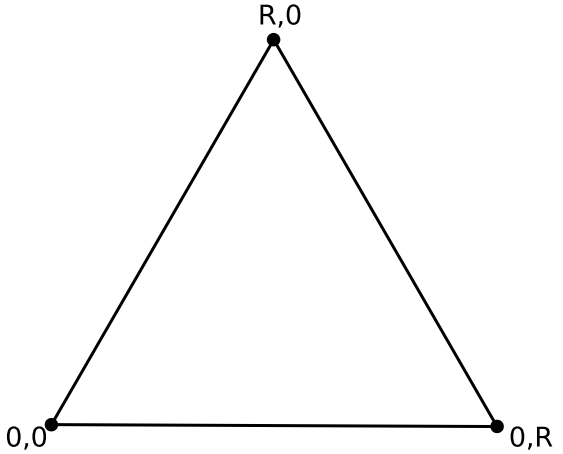

Figure 2.3: The coordinate B_{ij} for the vertices of a triangle.

2.3 Mesh colors

The original paper explains in detail the mesh coloring algorithm [Yuksel et al., 2010]. Thus, we will not describe it here. The paper does not provide implementation details surrounding the index lookup given the coordinate B_{ij} (figure 2.3) (from section 4.1 *"2D Filtering"*). Those details we will describe here.

2.3.1 Index calculation

Starting with B_{ij}, we have three cases; vertex, edge, or face. If B_{ij} exactly matches one vertex coordinate we are on a vertex. If either of B_i or B_j are zero, or the sum of the two is equal to the resolution R, we are on an edge. In all other cases we are on the face.

The formulas for calculating the indices and offset in the data buffer are given in the sections below.

Vertex

$$V_{index} = \begin{cases} 0 & \text{if } B_j = R \\ 1 & \text{if } B_i = 0 \, and \, B_j = 0 \\ 2 & \text{if } B_i = R \end{cases} \quad (2.7)$$

Edge

$$E_{index} = \begin{cases} 0 & \text{if } B_i = 0 \\ 1 & \text{if } B_j = 0 \\ 2 & \text{if } B_i \neq 0 \, and \, B_j \neq 0 \end{cases} \quad (2.8)$$

$$E_{offset} = \begin{cases} R - B_j & \text{if } E_{index} = 0 \\ B_i & \text{if } E_{index} = 1 \\ B_j & \text{if } E_{index} = 2 \end{cases} \quad (2.9)$$

Face

For faces, the data offset is the sum of all the previous columns plus the row offset. The row and column indices both start at one, since index 0 is contained within the edges.

$$\begin{aligned} F_{offset} &= \sum_{k=1}^{i-1} (R - (k+1)) + (j-1) \\ &= \frac{(2R - 2 - i) * (i-1)}{2} + (j-1) \end{aligned} \quad (2.10)$$

2.3.2 Differences in our implementation

Given the improvements in hardware since the original paper, we have used a structured array buffer instead of using the original workarounds with 2D textures [Yuksel et al., 2010]. We also combine the data lookup with the displacement lookup. We must still use per vertex data in the fragment shader, but we can use integers without interpolation between stages, a feature that was not widely available in 2010 hardware.

2.4 LOD-levels and MIP-mapping

We have support for storing and rendering different LOD and MIP-level, but the selection of which level to use is outside of the scope of this thesis, and thus in our implementation the selection is done manually for the whole mesh.

For both displacement and color data, we pregenerate $m = log_2(T)$ LOD-levels (for colors this is equivalent to MIP-levels). We store the LOD-levels contiguously in memory. The data for a given LOD-level can be accessed by adding an offset to the list's header memory address.

Note that LOD-level 0 is the highest resolution, and LOD-level m has lowest resolution. The tessellation factor T for a LOD-level lod (and equivalently resolution for mesh colors) is $T = 2^{m-lod}$.

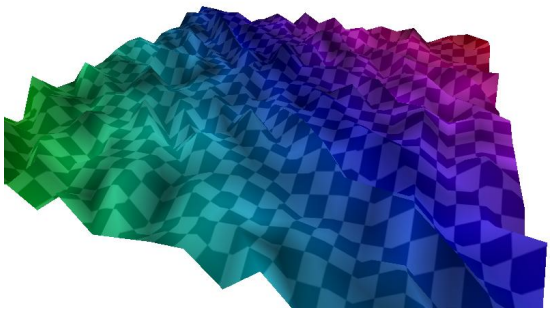

Figure 2.4: Lowest LOD resolution.

The following sections will cover how to generate the offset in the memory buffer for a given LOD-level, with LOD-level 0 at offset 0.

For the corner vertices, the offset is simply the LOD-level, since there is only one data point for each LOD-level, but for the edges and face it is not as simple.

2.4.1 Displacement

Edges

The first thing needed to calculate the offset is the number of data points (P) for any given edge and LOD-level. The number of vertices

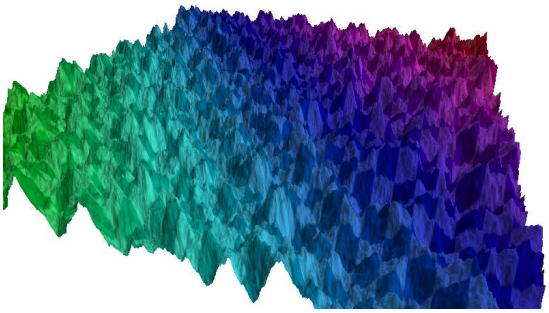

Figure 2.5: Highest LOD resolution.

on an edge is $T + 1$, but neither corner vertex is included in the edge buffer. The number of data points is thus equation (2.11).

$$P(lod, m) = 2^{m-lod} - 1 \tag{2.11}$$

The offset (O) for a LOD-level is the sum of all the data points in the previous LOD-levels (equation (2.12)).

$$\begin{aligned} O(m, lod) &= \sum_{k=0}^{lod-1} P(k, m) \\ &= 2^m + 2^{m-1} + ... + 2^{m-(lod-1)} \end{aligned} \tag{2.12}$$

This can be rewritten as a geometric progression with start value $a = 2^{(m-lod+1)}$, ratio $r = 2$, and $n = lod$ (equation (2.13)) [Wikipedia, 2013].

$$\begin{aligned} O(m, lod) &= \sum_{k=0}^{lod-1} (2^{(m-lod+1)} * 2^k) - lod \\ &= 2^{m+1} - 2^{m-lod+1} - lod \end{aligned} \tag{2.13}$$

Faces

The same operations for faces are more complicated. The number of points on a face is given by equation (2.14).

$$P(T) = 3(\frac{T^2}{4} - \frac{T}{2}) + 1 \tag{2.14}$$

From equation (2.14) we derive equation (2.15) and (2.16).

$$P(lod, m) = 3(\ 2^{2(m-lod-1)} - 2^{m-lod-1}) + 1 \tag{2.15}$$

$$O(m, lod) = \sum_{k=0}^{lod-1} P(k, m) \tag{2.16}$$

This can be written with two geometric progressions, one with $r = 4$ and one with $r = 2$ (equation (2.17), (2.18) and (2.19).

$$\begin{aligned} a_0 &= 2^{2(m-lod)} = 4^{m-lod} \\ r_0 &= 4 \end{aligned} \tag{2.17}$$

$$\begin{aligned} a_1 &= 2^{m-lod} \\ r_1 &= 2 \end{aligned} \tag{2.18}$$

$$\begin{aligned} &O(m, lod) = \\ &3(\sum_{k=0}^{lod-1} 4^{m-k-1} - \sum_{k=0}^{lod-1} 2^{m-k-1}) + lod = \\ &4^m - 4^{m-lod} - 3(2^m - 2^{m-lod}) + lod \end{aligned} \tag{2.19}$$

2.4.2 Mesh colors

Edges

For edges, we similarly compute the offset to our method for displacement edges (section 2.4.1).

Faces

We calculate the number of data points for one face using equation (2.20) and (2.21).

$$\begin{aligned} P(T) &= \frac{(T-2)}{2}(T-1) \\ &= \frac{T^2}{2} - \frac{3T}{2} + 1 \end{aligned} \tag{2.20}$$

$$\begin{aligned} P(m, k) &= \frac{2^{2(m-k)}}{2} - 3\frac{2^{m-k}}{2} + 1 \\ &= 2^{2(m-k)-1} - 3 * 2^{m-k-1} + 1 \end{aligned} \tag{2.21}$$

This offset calculation can too be written as a geometric progression (equation (2.22)).

$$\begin{aligned} &O(m, lod) \\ &= \sum_{k=0}^{lod-1} 2^{2(m-k)-1} - 3 \sum_{k=0}^{lod-1} 2^{m-k-1} + lod \\ &= \frac{1}{3}(2^{2m+1} - 2^{2(m-lod)+1}) \\ &\quad - 3(2^m - 2^{m-lod}) + lod \end{aligned} \tag{2.22}$$

2.5 Implementation considerations

2.5.1 Reversed edges

Since edge data is shared between two half-edges, one of them will need to render the content of the shared buffer backwards to represent it correctly. In our implementation, we encode this by using a negative sign on the edge index, indicating that it should be read backwards.

2.6 Theoretical performance

2.6.1 Memory usage

Compared to storing raw texture data in GPU memory our technique would consume approximatly the same amount of memory assuming that we use equivalent resolutions. It can even yield better results, since it support variable resolution per face. Textures compressed with DXT with hardware decompression though obviously takes up less memory, since we don't have any compression scheme for our hexagonal data pattern.

2.6.2 Time

Time analysis of GPU tasks are seldom useful, since what's more important on the GPU is cache locality and memory access.

2.7 Compute shader tessellation

We have implemented a version of the integer subdivision tessellation pattern in a compute shader. The primary reason for this was to see if performance could be improved by having the tessellator yield a unique index instead of only a barycentric coordinate. To contain complexity, our implementation is limited to work only with even integer tessellation factors.

2.7.1 Buffer size calculation

Since compute shaders is unable to allocate global memory or issue draw calls a buffer to store all the generated vertices and indices must be generated before the compute shader can run. Thus, we need to know the exact number of vertices and indices that would be generated.

For inner tessellation, given the factor T the number of vertices is equation (2.25).

$$nV(R) = 3(R-1)(T-R)+1 \tag{2.23}$$

$$nR(T) = \frac{3}{2}T \cdot \frac{1}{3} = \frac{T}{2} \tag{2.24}$$

$$\begin{aligned} nV(T) &= 3(\frac{T}{2}-1)(T-\frac{T}{2})+1 \\ &= 3(\frac{T^2}{4}-\frac{T}{2})+1 \end{aligned} \tag{2.25}$$

The reason for the single +1 in the formula is that the innermost triangle is a degenerate triangle (single vertex) for even tessellation factors (which is the only one we support).

The number of edge segments are the number of vertices without the last degenerate triangle. Thus:

$$nE(T) = nV(T) - 1 = 3(\frac{T^2}{4}-\frac{T}{2}) \tag{2.26}$$

The number of faces are twice the number of edges, since for each edge segment one triangle is added outwards, and one inwards:

$$nF(T) = 2 * nE(T) = 6(\frac{T^2}{4}-\frac{T}{2}) \tag{2.27}$$

For the outer edges, these numbers are simply the edge tessellation factor (both for number of vertices and number of faces).

For triangle list rendering, the number of indices are three times the number of faces. The actual number of indices drawn is controlled by an indirect draw call to support controlling LOD-levels from the GPU.

2.7.2 Triangle pattern generation

Generation of the vertices follow the method described in section 2.1. We then need to connect them by generating indices. In this section, we will cover the method for generating them for triangle lists, but in the optimization section we will show how to calculate indices for triangle strips.

Each ring on the face is processed separately and, since the edges are special, they are handled separately from the inner rings. For each line segment on the ring, a face is generated connecting inwards. For the inner segments, another face is added connecting outwards.

Inwards connecting point

The first value needed to calculate the index for the vertex on the next ring inwards is the ratio between the inner and outer ring, the *"triangle factor" TF* (equation (2.28)), where ES is the number of line segments on the inner and outer edge.

$$TF = \frac{ES_I + 1}{ES_O} \tag{2.28}$$

$$\begin{aligned} I_C = &(ES_I * I_E \\ &+ \lfloor TF * (I_V + 1) \rfloor) \bmod 3ES_I \end{aligned} \tag{2.29}$$

Equation (2.29) show how to calculate the connect index inwards, where I_E and I_V is the index of the edge and vertex relative to their context (ring and edge respectively). The formula is only valid for $ES_I > 0$ since modulo 0 is undefined. The index calculated is relative to the inner ring.

Outwards connecting point

In the same way as we calculate the inwards connecting point, we need to calculate a point

for connecting the outwards face from the line segment. These points must be selected so that the triangles do not overlap with the inwards faces, thus the triangle factor is the inverse of the triangle factor for the inwards face, and we round up instead of down.

$$TF = \frac{ES_O}{ES_I + 1} \tag{2.30}$$

$$I_C = \lceil TF * (I_V + 1) \rceil \tag{2.31}$$

The index calculated in equation (2.31) is as opposed to the inwards index not relative to the ring, but to the edge. This is due to the fact that for the outermost ring, the size of the edges are not consistent throughout the ring and needs to be handled separately. Care must still be taken to be sure to keep the index inside the ring, in the same way as the modulo in the inward case.

2.7.3 Per-face buffer offset

For each face we need to find where in the vertex and index buffer to start writing our data. We do this by keeping a buffer with the next write offsets in and reserving the size of the current face (found by using the formulas described in section 2.7.1) with atomic operations.

The offset cannot be calculated per-face since the tessellation level may vary and, thus, a single face cannot calculate the offset without knowing the previous faces' tessellation factors.

2.8 Optimizations

Both of these optimizations are only applicable to the compute shader tessellator.

2.8.1 Data compression

The segment consuming the majority of the time in the compute shader is the sequence of writes to global memory for indices and vertices. To reduce this, as many of the calculations as possible is moved to the vertex shader. This reduces the amount of data needed to be written from the compute shader.

The data compression optimization that we have accomplished stores the barycentric coordinate $\{\alpha, \beta, \gamma\}$ in one 32 bit integer by storing two of the components (α, β) as 16-bit integers and calculating the last one from these ($\alpha + \beta + \gamma = 1$).

2.8.2 Triangle strips

As writing to memory takes up the mayor part of the tessellation and there are many more indices than vertices, performance can be increased by reducing the number of indices by using triangle strips instead of triangle lists [Microsoft, 2013b].

Index structure

How the indices are generated needs some care for triangle strips so as to get winding order right, connecting the different rings and even more so for connecting the different triangles in the mesh (which may not be next to each other). The special cases are handed using degenerate triangles which are not rendered.

When processing the original triangles we start and end each with double vertices, this to make sure that they can always connect with the next and previous triangle group and that there wont be any problems with the first and last triangle in the mesh. Each ring inside the triangle also starts and ends with double vertices. The double start vertices are the same as the triangle start vertices for the outer edge. Figure 2.6 illustrates the double lines with triangle strips.

For each edge section, two indices are generated (as opposed to the six indices generated for triangle lists). Of these indices, one is on the current edge and one, on the next ring inwards. The formula for the inwards facing index is almost the same as before (equation (2.32)).

$$I_C = (ES_I * I_E + \lfloor TF * (I_V) \rfloor) \bmod 3ES_I \tag{2.32}$$

Number of indices

The number of indices have obviously changed. The new formulas for calculating the number of indices are equation (2.33) and (2.34).

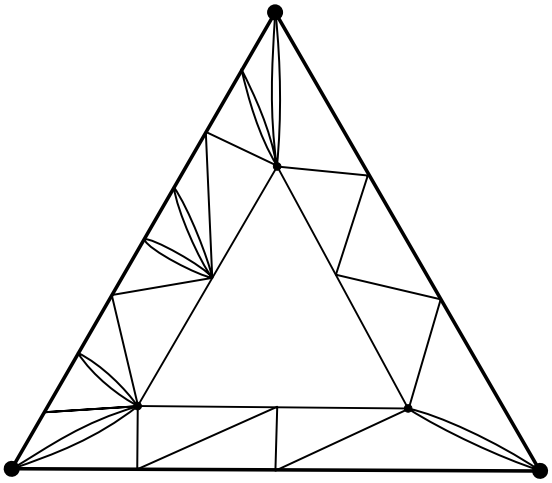

Figure 2.6: Triangle strips for edges with varying tessellation levels: The bent lines are for illustration purposes only, and shows where the lines doubles over.

$$nI_{Inner}(T) = 6*(\frac{T^2}{4} - \frac{T}{2}) + 4*(\frac{T}{2} - 1) \quad (2.33)$$

$$nI_{Edge}(T_{edge}, T_{inner}) = \quad (2.34)$$
$$2 * max(T_{edge}, T_{inner} - 2)$$

Comparing this to the number of indices for triangle lists we get the following difference (for face tessellation):

$$\begin{aligned} nI_{lists}(T) - nI_{strips}(T) &= \quad (2.35) \\ &= 9(\frac{T^2}{4} - \frac{T}{2}) + 3 \\ &\quad - (6 * (\frac{T^2}{4} - \frac{T}{2}) + 4 * (\frac{T}{2} - 1)) \\ &= \frac{3}{4}T^2 - \frac{7}{2}T + 4 \end{aligned}$$

This is approximately 30% fewer indices than with triangle lists (figure 2.7).

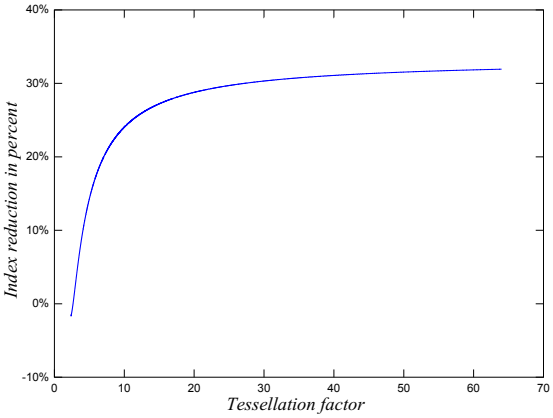

Figure 2.7: The index reduction in percent of using triangle strips over triangle list.

Chapter 3

Results

This section covers the results of our implementation.

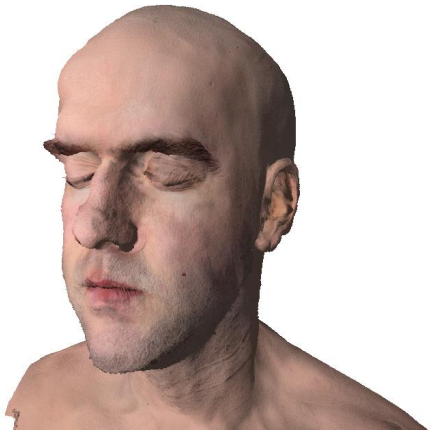

Figure 3.1: The head model rendered with our algorithm, using tessellation level 32, and color resolution 64 [McGuire, 2011].

Our method produces visual equivalent results of traditional methods, with the added advantage of elimitating the need for UV-mapping, which is as stated in the purpose section (section 1.3) useful for mesh morphing and procedually-generated meshes.

3.1 Performance tests

We have run performance tests to investigate the usability of these techniques in real time applications. The tests were run on both a NVIDIA GeForce GTX 680 and a AMD Radeon HD 7950.

3.1.1 Tests

The tests were run with the head model, consisting of 8842 faces. The tessellation level is 32, and the color resolution 64. All performance measurements is done using DirectX Queries with results in milliseconds. This metric provides the actual process time on the GPU.

The following are the different parameters we have tested:
Color: Only pixel color with phong shading. No tessellation.
Tessellation: Only tesselation, no colors.
Full: Tessellation and colors.

3.1.2 Baseline test

To put our test in context, we have tested shaders that use traditional textures for color and displacement. The displacement shader includes normal map lookup for normals on the new vertices. The results for this test are are in table 3.1. All tests uses hardware accelerated bilinear filtering for texture sampling.

3.1.3 Mesh color

Tests with only mesh colors, with different filters. The results for this test are are in table 3.2. These should be compared to the *Color* baseline test, and as such the NVIDIA results are especially bad (7 times worse performance). On AMD hardware the render times are only approximatly twice as long as the baseline test.

3.1.4 Hardware-fixed tessellation pipeline

Tests using the hardware tessellator. Full version uses mesh colors. The results for this test are are in table 3.3. Here the NVIDIA results are on par with the baseline tests, while on AMD our technique is slightly better with no coloring in the picture, but with coloring it performce much worse.

3.1.5 Compute shader tessellator

Tests using our compute shader tessellator with different optimizations. Full version uses mesh colors. The format for the numbers here are total render time, followed by time spent in the computer shader. The results for this test are are in table 3.4. Without color the performance on AMD hardware is equivalent to the fixed pipeline versions, but with color it performs much better. The NVIDIA performance is never really close to the fixed pipeline versions.

3.1.6 LOD and MIP-levels

The results for the different LOD- and MIP-levels are found in figure 3.2, figure 3.3 and figure 3.4. These results are in general a exponential decrese in render time, as expected.

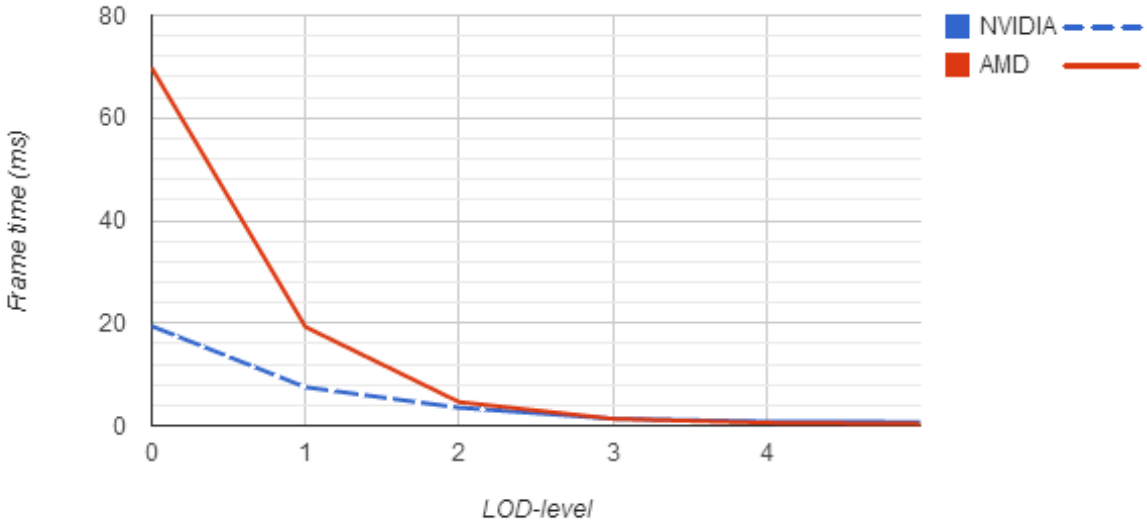

Figure 3.2: Performance of LOD-level. Both hardwares shows an expected performance increase on decreased level of detail.

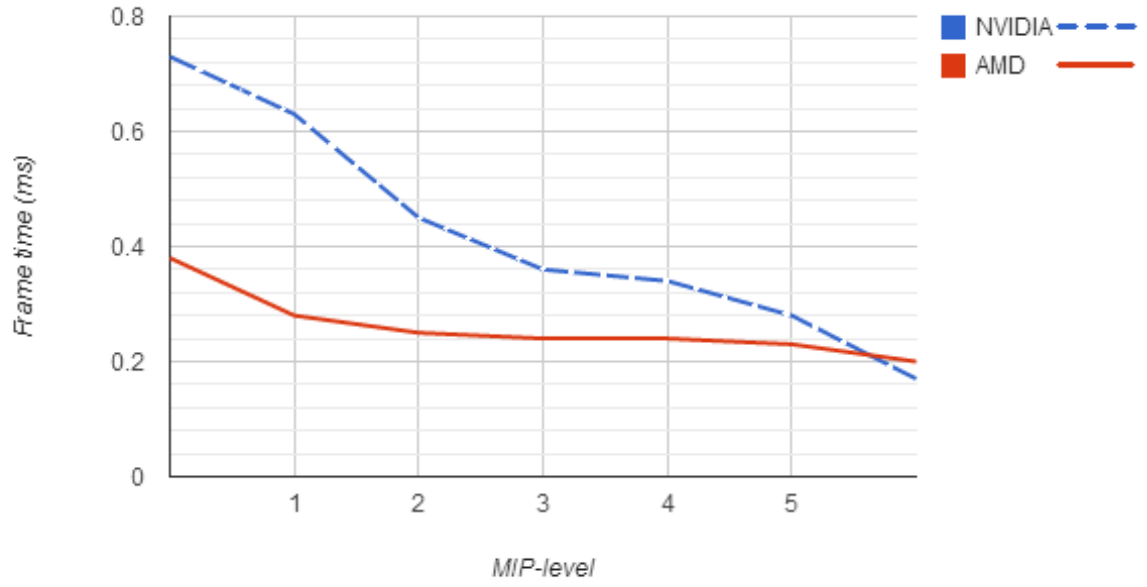

Figure 3.3: Performance of MIP-levels. Relatively small impact, probably due to number of actual samples not decreasing (but cache locality improves performance)

Test	AMD	NVIDIA
Color	0.11 ms	0.11 ms
Tessellation	30.6 ms	9.65 ms
Full	30.7 ms	9.75 ms

Table 3.1: Baseline test: Performance of textured techniques. Used as a baseline when evalutating the other tests.

Test	AMD	NVIDIA
Nearest	0.18 ms	0.40 ms
Bilinear	0.26 ms	0.70 ms

Table 3.2: Performance of mesh colors. On AMD hardware this technique is approximatly twice as slow as the baseline. On NVIDIA hardware the performance is even worse, approximatly 7 times as slow.

Test	AMD	NVIDIA
Tessellation	25.77 ms	9.65 ms
Full	70.80 ms	9.75 ms

Table 3.3: Performance of surface data tessellation with fixed tessellator. On NVIDIA hardware this technique performs identical with the baseline test, while AMD hardware falls short on *Full* (ie with mesh colors).

Test	AMD	NVIDIA
No optimizations	38.0 ms / 17.0 ms	51.0 ms / 35.0 ms
Compression	38.0 ms / 17.6 ms	47.0 ms / 30.0 ms
Compression and triangle strips	25.0 ms / 7.8 ms	28.0 ms / 15.0 ms
Compression, triangle strips and mesh colors	48.4 ms / 7.8 ms	40.0 ms / 15.0 ms

Table 3.4: Performance of surface data tessellation with compute shader tessellator. First number is total time, second is time spent in compute shader. On AMD hardware this techniques provides better times than the fixed pipeline version.

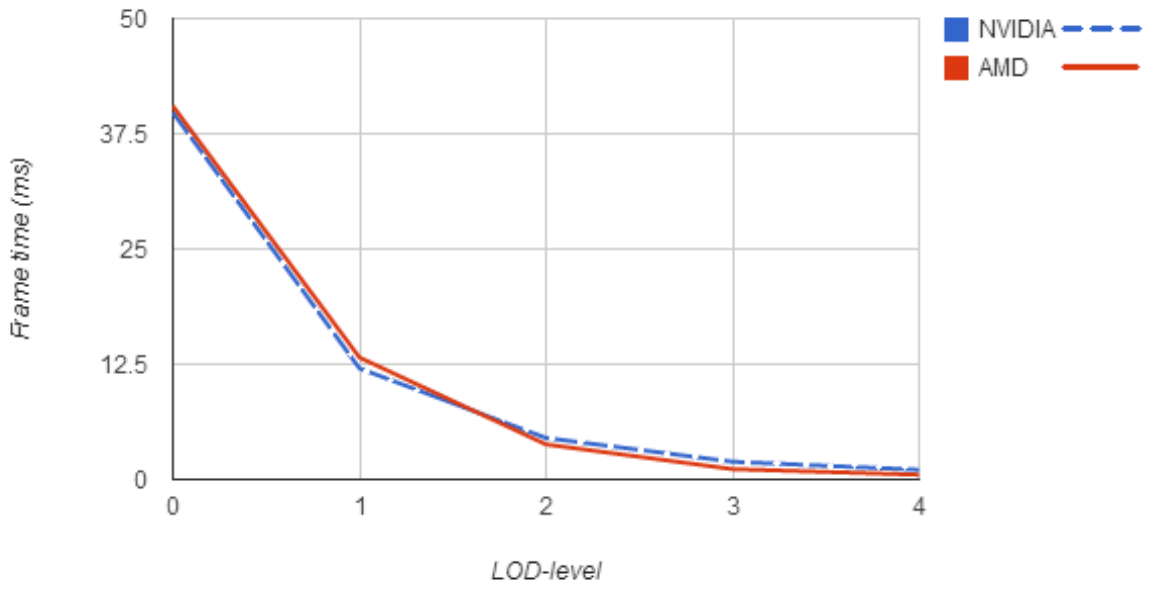

Figure 3.4: Performance of LOD in Compute Shader. Shows a expected exponential curve.

Chapter 4

Discussion

This section covers our reasoning surrounding the results and the future possibilities for it.

4.1 Analysis

As can be seen from the performance tests the hardware tessellator on AMD have some issues. Whats interesting though is how much worse the AMD card performed in the full test versus how it performed in the tessellator only test, especially since the times for mesh colors alone is in the context negligible. We have found that the cause for this performance hit is the extra data that needs to be sent from the domain shader to the vertex shader for mesh colors to work. AMD cards have issues with too much data being sent between shader stages. This is a good point for future optimizations.

On the other hand the AMD card outperforms the NVIDIA card when it comes to compute shaders (this is especially clear if one look at the compute shader times only, 7.8 ms on AMD versus 15.0 ms on NVIDIA), and that makes it possible for us to perform our tessellation there, for better results.

The other notable difference between the cards is in mesh colors where the AMD card is better. The reason for this might be the higher memory bandwidth of the HD7950 since mesh colors require a lot of memory lookups, especially for the bilinear filter [HWCompare, 2012].

The performance improvement of reducing the LOD-level is no surprise, the tessellation reduction is in power-of-twos and the results reflects this. For the MIP-levels the improvement of reducing the MIP-levels is not very big, but this is not the primary reason for MIP-mapping. The number of lookups are not reduced either, since that is controlled by the number of rendered fragments. The reason for the performance improvement is probably due to better cache locatity of the samples.

Worth noticing on the overall performance is that the mesh used in the tests have a high poly count and that an unnecessarily high tessellation level was used. The performance times for our algorithms should be considered in comparison with the baseline tests.

There are also the consideration of storing the displacement data in a different pattern than that for colors. Doing so saves filtering since the storage pattern always exactly matches that from the tessellator, but on the other hand this adds complexity to the implementation, and if in the future compression is concidered, two different schemes, or a scheme that works on both patterns must be deviced. Thus it could be worth using the mesh color pattern for both data types.

4.2 Conclusion

We have implemented and evaluated techniques for storing color and displacement data directly in a 3D-topology. From the performance results one can conclude that these methods could be usable in a real application, but since they obviously do not outperform the traditional methods they should only be used in cases where they are more suitable than the traditional methods or when the traditional methods is infeasible.

Even if the color data and displacement data is stored in a similar way they should still be handled separately, since the desired resolution for colors is often higher than the desired resolution for displacement. It could though be worth concidering using the same pattern for both displacement and colors, to save implementation complexity.

4.3 Future work

To work well with meshes with dynamic topology animation of the data is probably needed. Animation could probably be solved relatively easy, given that support for effectively streaming data to the GPU exists, but to properly render dynamic topology that would probably already be a requirement.

Another issue that needs to be addressed is compression of the data, primary when stored on disk. One could also explore the possibility to compress the color in the GPU mesh color buffer to reduce the memory footprint. Another point of optimization is to reduce the amount of sent between shader stages for mesh colors, especially to improve performance on AMD cards.

Bibliography

[Akenine-Möller et al., 2008] Akenine-Möller, T., Haines, E., and Hoffman, N. (2008). *Real-Time Rendering 3rd Edition*, chapter 6. Texturing, pages 147–199. A. K. Peters, Ltd., Natick, MA, USA.

[Catmull, 1974] Catmull, E. E. (1974). *A Subdivision Algorithm for Computer Display of Curved Surfaces.* PhD thesis. AAI7504786.

[Giesen, 2011] Giesen, F. (2011). A trip through the graphics pipeline 2011, part 12. `http://fgiesen.wordpress.com/2011/09/06/a-trip-through-the-graphics-pipeline-2011-part-12/`. [Online; Accessed 2013-10-10].

[HWCompare, 2012] HWCompare (2012). Geforce gtx 680 vs radeon hd 7950. `http://www.hwcompare.com/12350/geforce-gtx-680-vs-radeon-hd-7950/`. [Online; Accessed 2014-01-21].

[McGuire, 2011] McGuire, M. (2011). Computer graphics archive. `http://graphics.cs.williams.edu/data`. [Online; Accessed 2014-01-21].

[Microsoft, 2013a] Microsoft (2013a). Tessellation overview. `http://msdn.microsoft.com/en-us/library/windows/desktop/ff476340(v=vs.85).aspx`. [Online; Accessed 2014-01-21].

[Microsoft, 2013b] Microsoft (2013b). Triangle strips. `http://msdn.microsoft.com/en-us/library/windows/desktop/bb206274(v=vs.85).aspx`. [Online; Accessed 2014-01-21].

[Nießner and Loop, 2013] Nießner, M. and Loop, C. (2013). Analytic displacement mapping using hardware tessellation. *ACM Trans. Graph.*, 32(3):26:1–26:9.

[NVIDIA, 2010] NVIDIA (2010). Directx 11 tessellation. `http://www.nvidia.com/object/tessellation.html`. [Online; Accessed 2014-01-21].

[Piponi and Borshukov, 2000] Piponi, D. and Borshukov, G. (2000). Seamless texture mapping of subdivision surfaces by model pelting and texture blending. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 471–478, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

[Sander et al., 2001] Sander, P. V., Snyder, J., Gortler, S. J., and Hoppe, H. (2001). Texture mapping progressive meshes. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 409–416, New York, NY, USA. ACM.

[Schäfer et al., 2012] Schäfer, H., Prus, M., Meyer, Q., Süßmuth, J., and Stamminger, M. (2012). Multiresolution attributes for tessellated meshes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 175–182, New York, NY, USA. ACM.

[Weisstein, 2014] Weisstein, E. W. (2014). Barycentric coordinates. [From MathWorld – A Wolfram Web Resource; Accessed 2014-01-24].

[Wikipedia, 2008] Wikipedia (2008). Triangle barycentric coordinates. `http://en.wikipedia.org/wiki/File:TriangleBarycentricCoordinates.svg`. [Online; Accessed 2014-02-08].

[Wikipedia, 2013] Wikipedia (2013). Geometric progression. `http://en.wikipedia.org/wiki/Geometric_progression`. [Online; Accessed 2014-01-21].

[Wikipedia, 2014a] Wikipedia (2014a). Arithmetic progression. `http://en.wikipedia.`

org/wiki/Arithmetic_progression. [Online; Accessed 2014-01-21].

[Wikipedia, 2014b] Wikipedia (2014b). Polygon mesh. http://en.wikipedia.org/wiki/Polygon_mesh. [Online; Accessed 2014-12-28].

[Yuksel et al., 2010] Yuksel, C., Keyser, J., and House, D. H. (2010). Mesh colors. *ACM Trans. Graph.*, 29(2):15:1–15:11.

Appendix A

Glosary

- **Barycentric coordinates** A three component vector describing a position on a triangle (see section 1.1.3).
- **Bilinear filtering** A filtering technique that samples all neighboring pixels and smoothly blends them.
- **CPU** Central Processing Unit
- **DirectX** A Graphics API on Microsoft systems.
- **DXT** A texture compression algorithm fit for real time decompression on the GPU
- **GPU** Graphics Processing Unit
- **GPGPU** General Purpose GPU. Refers to the ability to perform general computing tasks on a GPU, often refered to a Compute Shaders.
- **LOD** Level of Detail. A sequence of versions of the same mesh, with decreasing detail.
- **Mesh** A collection of vertices edges and faces that defines the shape of a three dimensional solid with flat faces and sharp edges. [Wikipedia, 2014b].
- **MIP** Or Mipmaps. A pre-calculated optimized sequence of lower resolution versions of a texture.
- **Nearest filtering** A filtering technique that always picks the color of the nearest neighboring pixel.
- **REV** Ring-, Edge-, Vertex-index. Acronym defined by H.Schäfer et al. refering to a three component vector of indices describing a discrete position on a triangle.
- **Tesselation** The subdivision of a mesh surface into smaller triangles.
- **Topology** In the scope of this thesis this refers to the surface of a mesh.
- **UV-mapping** Or texture mapping. The mapping of a 2d dimensional texture onto the surface of a 3d dimensional mesh.