Alexander Komar, BSc

Real-Time Gaussian-Product Subdivision on the GPU

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Dipl.-Ing. Dr.techn. Bakk.rer.soc.oec. Reinhold Preiner

Institute for Computer Graphics and Vision
Head: Univ.-Prof. Dr.rer.nat. M.Sc. Tobias Schreck

Graz, September 2020

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

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

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Abstract

This thesis proposes a method for rendering Gaussian-Product subdivision surfaces in real-time. It is achieved using an existing algorithm for performing an approximation to subdivision surfaces on the GPU. The algorithm was adapted to handle covariance meshes and the larger per-vertex data. The approximation is done using Bézier patches, which are organized in a quad tree data-structure to ease the handling on the GPU. The OpenGL tessellation shaders are used to compute the control points for the Bézier patches and to evaluate the surface, described by these patches. The geometric properties of the method are analyzed and performance of the system is evaluated using difference images, timing analysis and normal comparisons.

Content

Abstract v

1 Introduction 1

2 Background and Related Work 5
2.1 Subdivision Surfaces 5
2.2 Gaussian-Product Subdivision 7
2.2.1 Weights of the Gaussian-Product Subdivision 10
2.2.2 Transformation of the Input 10
2.3 Real-Time Rendering of Subdivision Surfaces 12
2.3.1 Algorithm overview 12
2.3.2 Mesh Preprocessing 13
2.3.3 Computation of the Control Polygon 14
2.3.4 Evaluation of the Control Polygon 16

3 Gaussian-Product Subdivision on the GPU 17
3.1 OpenGL Tessellation 17
3.1.1 Tessellation Control Shader 18
3.1.2 GPU Tessellator 20
3.1.3 Tessellation Evaluation Shader 20
3.2 Implementation of Real-Time Subdivision in the OpenGL Pipeline 21
3.2.1 Tessellation Control Shader 21
3.2.2 Tessellation Evaluation Shader 23
3.3 Buffer Setup 23
3.4 Normal Vector Computation 26
3.5 Implementation Details 28

4 Results **29**
4.1 Material Capture Shading 29
4.2 Visual Comparision 29
4.3 Performance Tests 33
4.4 Normals Comparison 37
4.5 Conclusion 38

5 Geometric Discussion **39**
5.1 Continuity of Gaussian-Product Subdivision 39
5.2 Continuity Properties in our Real-Time Pipeline 40

6 Outlook **43**

Bibliography **45**

List of Figures

1.1 Rendering of a Gaussian Product Subdivision surface. The slightly gray lines represent the base mesh, the blue ellipsoids illustrate the covariance matrices of the vertices. The anisotropic extend represents a probabilistic indicator for the surface alignment around a vertex 1

2.1 The first subdivision steps (left to right: control mesh, 1 step, 2 steps) and the limit mesh. 5
2.2 The product of the two black Gaussians Θ_i and Θ_j is the red Gaussian Θ_{ij}. The height lines denote lines of equal probability density. The blue lines, running along the Gaussians is the probability ridge. Image taken from [17]. 8
2.3 Visualization of the steps in the rendering pipeline. 12
2.4 Node types example. Left an internal node, with 4 extraordinary vertices. An internal node divides a face into the four smaller faces of the next subdivision step. Right a terminal node, with one extraordinary vertex (bottom right). Terminal nodes store the information to evaluate the surface patch increasing the resolution of Bézier patches bordered by extraordinary vertices. 14
2.5 Rules for calculating the control polygon for a new point inside the patch 2.5a, on an edge 2.5b and on a vertex 2.5c. The numbers at the vertices represent the weights and the variable v represents the valency of the vertex. 15

3.1 The full OpenGL pipeline with all optional stages in light blue 18
3.2 Tessellation example of a quad and triangle patch 19
3.3 The differences between even (left) and odd (right) fractional spacing with different subdivision levels (middle) 21

3.4 Interactions between the tessellation shader stages and the GPU memory. 22
3.5 Visualization of the structure of the *Neighbors* and *Linked Faces* buffer. $i0$, $i1$ being the indices in this buffer of the 0th and 1st vertex. Framed in red is the vertex information, starting with the size of the slice (e.g. 3), followed by 3 vertex/face indices. The green frame contains the information of the next vertex. . 24

4.1 The 3 MatCap texture used in the results section. 30
4.2 Preprocessing times (blue) and render times (orange) for the valence test meshes. 30
4.3 All valency test meshes, from valency 3 to valency 10 rendered with real time covariance rendering (top) and their base mesh and covariances (bottom). 31
4.4 The anvil mesh rendered by our GPU subdivision (left), by the reference CPU subdivision (middle) and the difference image between the two (right), equal pixels are black and pixel that differ are colored. The difference image was enhanced by factor 200. 32
4.5 The wirecube mesh subdivided and rendered by the GPU. . . 32
4.6 Render time analysis for different meshes 34
4.7 Plots of memory consumption over render time for all performance evaluation meshes. The markers highlight the measurement points at the tessellation levels (1-6). 36
4.8 The normal vectors of the distorted cube mesh encoded in the colors of the mesh, for the GPU render (left), the CPU subdivision (middle) and their difference image (right), equal pixels are black and pixel that differ are colored. The difference image was enhanced by factor of 5000. 37

1 Introduction

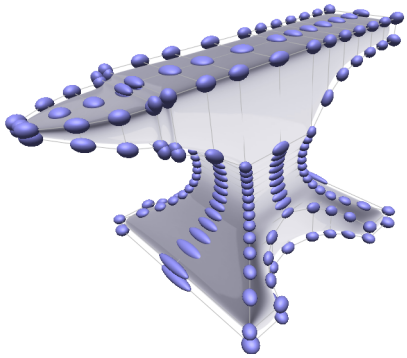


Figure 1.1: Rendering of a Gaussian Product Subdivision surface. The slightly gray lines represent the base mesh, the blue ellipsoids illustrate the covariance matrices of the vertices. The anisotropic extend represents a probabilistic indicator for the surface alignment around a vertex

Subdivision Surfaces are a widely used concept in computer graphics and Computer Aided Design (CAD) that was introduced in the late 1970. The principle is simple but very powerful: Take all faces of a base mesh and divide them into smaller faces, while introducing new vertices as linear combinations of the surrounding old ones and updating the old ones as a linear combination of the old vertex to update and the newly introduced surrounding ones. With this rather simple technique a low resolution base mesh, which is easy to construct and keep in memory, can be transformed into a well defined and smooth limit mesh. Further, this limit mesh has good continuity (C^2) everywhere, except at extraordinary vertices, where it is C^1 [1].

Although a quite simple and elegant scheme, the topology of the base mesh needs to conform to the scheme. There are different schemes proposed by different research groups for different purposes and topologies. Some are

made especially for triangle meshes [13], some for quadrilateral meshes [3], and others for polygons with any number of sides [5]. Moreover, we can differentiate between approximating schemes [3, 13, 5], and interpolating schemes [8, 11]. All these schemes work on the same principle, but use different weights for the linear combination of the surrounding vertices.

One perk of this method is also one of its weaknesses: it always generates smooth surfaces. There is no simple way of generating a sharp edge or a pointy peak with these schemes. There has been work on including the possibility to generate such features with extensions to the way the mesh is subdivided [4] that work well and are widely used, especially in the cinematic setting. Another way of introducing sharper edges and pointier peaks to these methods is to increase the granularity of the base mesh at such edges/vertices. However, this leads to a more complex shape information and requires more memory. A general solution to this problem does not exist since the limit surface will always be smooth by definition.

This calls for a whole different approach to generate subdivision surfaces. Preiner et al. [17] introduced such a new probabilistic surface definition that extends the vertices of the base mesh by Gaussian covariance matrices and applies a new nonlinear subdivision called *Gaussian-Product Subdivision* (GPS). Each subdivision step now takes into account not only the position of a vertex but also a covariance matrix information defining the Gaussian probability distribution of possible surface locations appearing around it. This makes the computation of the subdivision step non-linear and more involved. However, the authors show that the input vertices can be transformed into a higher-dimensional space where the subdivision scheme becomes linear, achieving the same result. This makes the implementation almost as computationally expensive as the linear schemes above, except it requires the transformation step before and after the subdivision and more memory for storing the covariance information for each vertex. With such a scheme the sharp edges or pointy peaks can be better modeled, by changing a few covariance matrices. Furthermore, the method also allows for much more freedom in designing more complex surfaces without increasing base mesh complexity.

In this thesis another problem of subdivision surfaces is tackled for this exciting new method: The real-time computation of the limit surface.

With the decrease in cost for high GPU computing power came a new desire to compute the limit surface of a subdivision scheme in real-time on the GPU. This is not a trivial computation, since the subdivision is a inherently recursive process, which can not be computed in parallel out of the box. However, Jos Stam developed a direct evaluation technique for subdivision, first for quadrilateral meshes [20], then for triangular meshes [19]. First advances in the field of real-time subdivision where done by Shiue, Jones and Peters [18] and Patney and Owens [16], which used particular optimizations on the GPU using a special texture memory layout for the data or prefix sums and reduce operations, respectively. Some mayor advances were done in recent years by Nießner et al. [15] and Brainerd et al. [2]. The authors managed to achieve a real-time evaluation of the limit surface by using the full OpenGL pipeline and powerful hardware. The performance and visual effects behind the proposal of Brainerd et al. [2] can be seen in the video game *Call of Duty Ghosts*. Nießner et al. and Brainerd et al. both heavily relied on the direct evaluation developed by Jos Stam [20].

This thesis proposes a method for rendering a Gaussian-Product Subdivision surface in real-time on the GPU, building from the basic principles mentioned above. The papers from Nießner et al. [15] and Brainerd et al. [2] do not discuss the full details of their approach and are thus difficult to reproduce. This thesis addresses this problem by providing documentation on how to implement these methods on the GPU correctly. Moreover, an exact performance evaluation of the render times and memory consumption is given.

2 Background and Related Work

In this chapter the related work and theoretical background of this thesis is laid out. It covers the bases of the implemented rendering pipeline and the method for real-time subdivision. The techniques utilized in this thesis are discussed and put into the context of subdivision surfaces.

2.1 Subdivision Surfaces

Subdivision surfaces for general mesh topologies were first introduced in 1978 by two independent researcher teams Catmull and Clark [3] and Doo and Sabin [6] based on the PHD thesis of Doo [5]. Both papers introduced a similar method for subdividing mesh faces into smaller mesh faces, only for different face types: Doo for arbitrary polygons and Catmull and Clark for quadrilateral polygons.

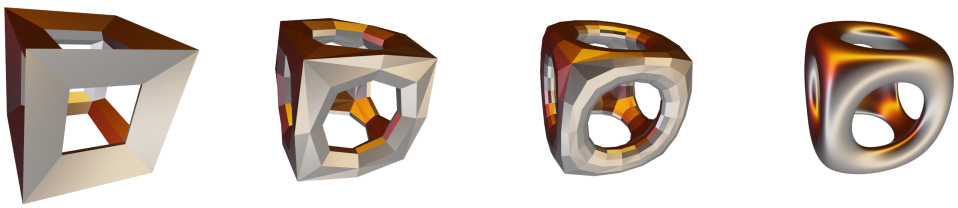


Figure 2.1: The first subdivision steps (left to right: control mesh, 1 step, 2 steps) and the limit mesh.

The principle of subdivision surfaces, as it was proposed by Catmull and Clark is as follows: starting with a low resolution control mesh, consisting of quadrilaterals, the mesh is iteratively refined, until the desired smoothness is

Classification	Primal	Dual
Approximating	Catmull-Clark[3], Loop[13]	Doo-Sabin[6], $\sqrt{3}$[11]
Interpolating	Butterfly[7], Mod. Butterfly[21]	Doo-Sabin for Quads[12]

Table 2.1: Classification of subdivision schemes.

obtained. This can be seen in Figure 2.1. These subdivisions are usually done in a preprocessing step, before displaying such a surface, since the algorithm is by definition recursive and the calculation time grows exponentially as the recursion depth increases.

After these initial papers, a whole "zoo" of subdivision schemes was introduced for many purposes and mesh topologies. The general classification of a subdivision algorithm is done in two major ways. First it can be either approximating or interpolating and second it can use either a primal or dual vertex generating method. An approximating scheme asymptotically approximates a Bézier surface with each iteration. Interpolating schemes interpolate the control mesh with each iteration into a smooth surface. A primal vertex generating scheme splits the faces of the mesh into smaller faces. In contrast, dual vertex generating schemes use corner cutting to subdivide the mesh, meaning they introduce new faces at the vertices of the base mesh. Table 2.1 illustrates the classification of some subdivision schemes.

The subdivision scheme introduced by Loop [13] uses a triangle mesh as a control mesh and then recursively subdivides each triangle face into four smaller triangles. During the subdivision new vertices are introduced, by computing a linear combination of surrounding old vertices. Moreover, old vertices are also updated using a linear combination of their surrounding vertices. Because of the change in the old vertices it is considered a approximating subdivision scheme. On the contrary, e.g. the modified butterfly subdivision scheme, introduced by Zorin et al. [21] does not update existing vertices and is therefore an interpolating subdivision scheme. The modified Butterfly subdivision scheme also works on triangle control meshes, but the neighborhood used to compute the new vertices is larger than the original

Butterfly subdivision scheme proposed by Dyn, Levin and Gregory [7].

The $\sqrt{3}$ subdivision scheme, introduced by Kobbelt [11] also uses triangle meshes as control meshes, but it's splitting method is different. While Loop subdivision splits one triangle into four, $\sqrt{3}$ subdivision splits one triangle into three, by introducing just one new face vertex.

Doo-Sabin [6] is a totally different subdivision scheme. It does not split faces and introduce new vertices, but splits vertices and introduces new faces. In other words, one subdivision step replaces every vertex by a face and its new vertices are computed using linear combinations of the surrounding vertices. This introduces different kinds of polygons, depending only on the valence of the vertex they replace.

The Catmull-Clark algorithm was later picked up by Pixar Animation Studios in 1997 with the short film Geri's Game. The animation showed the potential of the method and from this point onward all Pixar movies used subdivision surfaces for character modeling. The Pixar movie "A Bug's Life", which was released in 1998, was the first full length movie to feature this new technique.

The resulting surface of these different algorithms is a smooth approximation of the control mesh. This is advantageous when a smooth surface should be the result, but is not optimal when the resulting mesh should include some sharp creases and corners. A solution to this was proposed by DeRose et al. [4]. They introduced special subdivision rules that take into account a sharpness value, defined for each edge of the mesh. This is a widespread approach to defining sharp or semi-sharp creases in a subdivision control mesh. Another approach will be introduced next.

2.2 Gaussian-Product Subdivision

The idea behind Gaussian Product Subdivision (GPS) surfaces was introduced by Preiner et al. [17], while also developing a compact and efficient way to calculate the limit surface of the scheme. The researches designed a non-linear subdivision scheme, which uses information about the probability distribution of the location of new vertices around existing vertices,

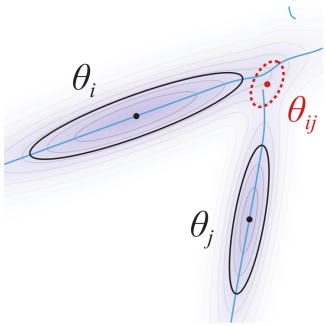


Figure 2.2: The product of the two black Gaussians Θ_i and Θ_j is the red Gaussian Θ_{ij}. The height lines denote lines of equal probability density. The blue lines, running along the Gaussians is the probability ridge. Image taken from [17].

stored in a covariance matrix of a Gaussian distribution. This method yields a non-linear subdivision algorithm, which is rather expensive to compute in 3D. Therefore, they proposed a method of mapping the problem to a higher-dimensional space, where the subdivision scheme becomes linear, and then mapping the limit surface back to 3D. This makes the computation easier and more intuitive. Further, the researchers proofed that using this transformation, the limit surface computed inherits the smoothness properties of the underlying linear subdivision scheme used in the higher-dimensional space.

The input to a Gaussian-Product subdivision is a *covariance mesh*, which is defined as a manifold Mesh $\Pi = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ that encodes the parameters of a Gaussian distribution $\Theta_i = (\mu_i, \Sigma_i)$ in each vertex $V = \{\Theta_i\}$ and their topology in edges $\mathcal{E}$ and faces $\mathcal{F}$. The subdivision step is similar to other existing subdivision schemes, but defines newly inserted vertex Gaussians as the product of given or old vertex Gaussians. The continuous limit surface closely follows the probability ridges of the vertex Gaussians. This can be seen in Figure 2.2 in the 2-dimensional case.

The Probability Density Function (PDF) of a Gaussian is defined as follows:

$$f(x|\Theta) = \frac{1}{\sqrt{|2\pi\Sigma|}} e^{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)} \tag{2.1}$$

In order to define a smooth contour between two vertices along their connecting probability ridge, the combined PDF of Gaussians is used:

$$f(x|\Theta_{ij}) = \omega^{-1} f(x|\Theta_i)^{\alpha_i} f(x|\Theta_j)^{\alpha_j} \tag{2.2}$$

where ω is the normalization factor $\omega = \int_{\mathbb{R}^d} f(x|\Theta_i)^{\alpha_i} f(x|\Theta_j)^{\alpha_j}$. The equation shows that the weights are encoded in the power of the PDF functions of the Gaussians. Reformulating this to calculate the position μ_{ij} and covariance Σ_{ij} of the new vertex gives:

$$\mu_{ij} = \left(\alpha_i \Sigma_i^{-1} + \alpha_j \Sigma_j^{-1}\right)^{-1} \left(\alpha_i \Sigma_i^{-1} \mu_i + \alpha_j \Sigma_j^{-1} \mu_j\right) \tag{2.3}$$

$$\Sigma_{ij} = \left(\alpha_i \Sigma_i^{-1} + \alpha_j \Sigma_j^{-1}\right)^{-1} \tag{2.4}$$

which describes a curve resembling the ridge of the combined PDFs of the two Gaussians using the parameter $t = \frac{\alpha_i}{\alpha_i + \alpha_j}$. This parameter reduction is valid since the weights are independent of any scaling of the Gaussians. Extending the product of the two PDFs to n PDFs gives:

$$f(x|\Theta_{ij}) = \omega^{-1} \prod_{i \in \mathcal{J}} f(x|\Theta_i)^{\alpha_i} \tag{2.5}$$

This extension gives us the formula to calculate the combination of multiple vertices, which is used by all subdivision schemes. This equation leads to different subdivision operations. What remains is the choice of the weights of the vertices. These need to produce the desired subdivision properties, like smoothness and continuity.

2.2.1 Weights of the Gaussian-Product Subdivision

For a linear subdivision scheme the determination of the weights is typically done by analyzing the subdivision matrices. In a non-linear subdivision scheme, like the one introduced above, it is non-trivial to derive weights that also fulfill requirements such as smoothness and continuity. To derive such weights, a mapping of the Gaussians to a higher-dimensional space is discussed. As a Gaussian distribution has nine parameters in 3D space, three for the position and six for the symmetric covariance matrix, a transformation to 9D space is aimed for. In this 9D space the non-linear subdivision will be reduced to a linear subdivision.

First the PDF is reformulated, to express its exponent as a quadratic basis $b(x)$ and a coefficient vector q_i:

$$f(x|\Theta_i) = c \cdot e^{-\frac{1}{2}b(x)^T q_i} \tag{2.6}$$

where c collects all non exponential factors. This way, every Gaussian is encoded as q_i using a bijective map $F : \Theta_i \mapsto q_i$ into a space of quadratic functions Q. Substituting this with the Equation 2.5:

$$f(x|\Theta_J) = c \cdot \exp^{-\frac{1}{2}b(x)^T \sum_i \alpha_i q_i} = c \cdot \exp^{-\frac{1}{2}b(x)^T q_J} \tag{2.7}$$

where $q_J = \sum_{i \in J} \alpha_i q_i$. This shows that there is a bijective map F that transforms the vertices of a covariance mesh into a dual space where the non-linear product of Gaussians is a linear combination. Using the weights of a linear subdivision scheme on that hypermesh results in a non-linear subdivision of the control mesh in 3D space. Furthermore, we can apply the weights of any suited linear subdivision scheme to the subdivision of the hypermesh.

2.2.2 Transformation of the Input

In order to obtain the desired properties of the subdivided mesh, the mapping function that is used to transform the Gaussians into higher-

dimensional space has to fulfill certain conditions. First, the mapping needs to be bijective and smooth over its input domain. Only then the 3D limit surface can inherit the smoothness properties of the underling linear subdivision scheme operating in 9D space. This will be proven in the Evaluation Chapter 5, along with the continuity of the whole pipeline. Further, the mapping has to satisfy the equation:

$$b(x)^T q_i + c_{q_i} = (x - \mu_i)^T \Sigma^{-1} (x - \mu_i) \tag{2.8}$$

For convenience the polynomial basis is defined as:

$$b(x)^T = (vech(2xx^T - diag(x)^2)^T, -2x^T) \tag{2.9}$$

where the parts are the bases for the quadratic and linear coefficients respectively. The $vech(\cdot)$ function describes the half vectorization operator of the matrix, linearizing the lower triangular part of the matrix and the $diag(\cdot)$ operator transforms the vector into a matrix, with the elements of the vector in the diagonal of the matrix.

Using this quadratic basis function, the map $F(\mu, \Sigma)$ is given by the vector $q = (\hat{q}, \overline{q})$, the quadratic and linear coefficients respectively. The coefficients are calculated as follows:

$$\hat{q} = vech(\Sigma^{-1}) \qquad \overline{q} = \Sigma^{-1}\mu \tag{2.10}$$

For the inverse transformation, the covariance matrix is restored first and using the restored matrix the mean is restored lastly:

$$\Sigma = F_\Sigma^{-1}(q) = [\hat{q}]^{-1} \qquad \mu = F_\mu^{-1}(q) = \Sigma\overline{q} \tag{2.11}$$

where $[\cdot]$ restores the matrix linearized by the $vech(\cdot)$ operator. The authors [17] show that using this transformation any linear approximating subdivision scheme can be applied to such covariance meshes, guaranteeing that the resulting limit mesh is well-defined and adopts the smoothness characteristics of the underling linear subdivision scheme.

2.3 Real-Time Rendering of Subdivision Surfaces

Many attempts were made to directly evaluate subdivision surfaces. The main problem with the direct evaluation is that in the limit, subdivision surfaces do not behave exactly like a Bézier patch at extraordinary vertices. This can be tackled in may different ways. REYES style subdivision proposed by Patney and Owens [16] introduces a different style of rendering, compared to the standard way a GPU rasterizes a model. It subdivides all meshes to micro-polygons of the size of one pixel and then calculates the shading and occlusions for them. Another method uses optimized memory layout and GPU kernels achieving almost real time performance [18]. Furthermore, a solution was proposed by Nießner et al. [15] and later refined by Brainerd et al. [2]. They proposed a solution that stores the topology of the control mesh in a quad tree on the CPU. Then the control mesh gets subdivided on the GPU and transformed to Bézier patches. These are then evaluated using the tessellation shaders and tessellator in the OpenGL pipeline.

In the following section a closer description of the method of Brainerd et al. is given, as it is the basis for the rendering pipeline in this thesis.

2.3.1 Algorithm overview

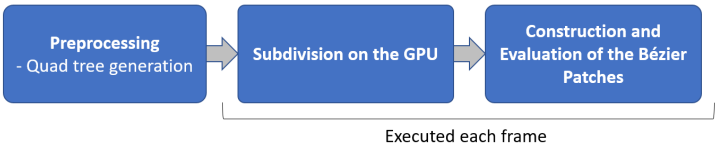


Figure 2.3: Visualization of the steps in the rendering pipeline.

The naive way of performing subdivision on the GPU would be to parallelize the individual face, edge and vertex subdivisions and then executing them as many times as subdivision steps are needed. Using this approach we would need to render the resulting mesh in a second pass. This method is very costly, since it has to be executed many consecutive times and the required memory bandwidth on the GPU and to the GPU is high, since

mesh data needs to be streamed from global memory on the GPU to the GPU multiprocessors.

Hardware tessellation methods like the one described in the following can bypass these limitations by doing less iterative subdivision steps and consuming the data directly from the GPU without a second render pass. The challenge of this method is splitting the input faces into convenient surface portions that can be directly evaluated. Jos Stam [20] showed that subdivision surfaces can be directly evaluated. He described an implementation for the CPU, whose extension to the GPU is straightforward, but requires many expensive floating point calculations. The method below combines GPU tessellation with the evaluation techniques of Jos Stam, circumventing most expensive floating point operations.

Figure 2.3 shows the steps of the rendering pipeline. These steps are discussed in the following subsections.

2.3.2 Mesh Preprocessing

The preprocessing is performed once on the CPU. It is an important step in the pipeline, since it not only prepares the topology of the mesh for later use on the GPU but it also is the main step in the pipeline responsible for the performance gains later on. This step calculates the control point stencils and structures them in a quad tree data structure to be later sent to the GPU for the evaluation. The quad tree data structure is composed of three types of nodes: internal nodes, regular nodes and terminal nodes. The internal and terminal node can be seen in Figure 2.4. The internal node does not contain any control point stencil, but references to four child nodes. The regular node represents a regular Bézier patch composed of four regular vertices. It contains the control point stencils for this patch. The terminal node is used when the Bézier patch is composed of three regular vertices and one irregular one. It simplifies the calculation of the control points, because it combines 3 regular patches and an irregular patch. This is done because it is an often recurring constellation of patches. Note that, since at least one subdivision step is always executed and thus always produces regular vertices, no more than one irregular vertex can be contained in

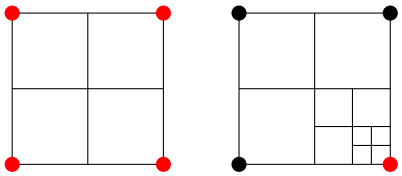


Figure 2.4: Node types example. Left an internal node, with 4 extraordinary vertices. An internal node divides a face into the four smaller faces of the next subdivision step. Right a terminal node, with one extraordinary vertex (bottom right). Terminal nodes store the information to evaluate the surface patch increasing the resolution of Bézier patches bordered by extraordinary vertices.

a single node. A terminal node references the four corner points of the patch and their one-ring neighborhoods in a 5 by 5 array. Based on this array, the control points of the Bézier patch defining the surface patch are calculated.

In order to calculate these quad trees, the input mesh gets subdivided step by step and for each face a quad tree is built up, starting from the original face as root node, and adding its subdivided faces as child nodes. This is done up to a certain user-chosen depth. Increasing the depth gives a closer approximation, but also a more demanding calculation in real time.

Since apart from the quad tree generation the preprocessing step includes subdivision steps, those steps have to be done on the GPU in real-time, for the algorithm to be truly real-time. As shown later in Chapter 4, in our testing we found that a good approximation could already be achieved with two subdivision steps computed on the GPU. The implementation of the subdivision was done straightforward with the subdivision formulas introduced by Catmull and Clark [3].

2.3.3 Computation of the Control Polygon

In order to evaluate a Bézier patch at an arbitrary point, a control polygon of 4 x 4 vertices is needed. This is calculated using the quad tree, mentioned in the section before. Once the two parameters, describing the position of the new vertex inside the face of the mesh are calculated by the tessellation unit, the quad-tree is traversed to find the correct Bézier patch corresponding to

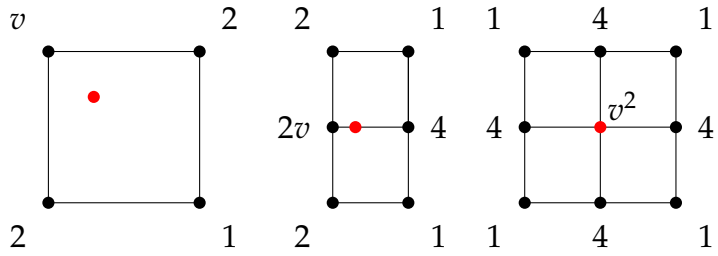


Figure 2.5: Rules for calculating the control polygon for a new point inside the patch 2.5a, on an edge 2.5b and on a vertex 2.5c. The numbers at the vertices represent the weights and the variable v represents the valency of the vertex.

the location on the surface. The three node types are handled differently when encountered during the traversal. When an internal node is encountered, the traversal continues with the respective child node referenced by the internal node corresponding to the two position parameters inside the control mesh face. If a regular node is encountered, the control polygon is computed using the referenced stencils, as shown in Figure 2.5. Finally if a terminal node is encountered, the control polygon is calculated using the rules in Figure 2.5 and Equation 2.12.

In order to calculate a control point inside the patch (2.5a), a weighted sum of the vertices defining the patch is used. The calculations for a control point on an edge (2.5b) uses the vertices of the patch and the two vertices of the opposing edge outside the patch. Finally, the computation of a control point on a vertex (2.5c) uses the whole one-ring neighborhood. The numbers in Figure 2.5 are the weights for the vertices, the v parameter stands for the valency of the vertex. The calculation of the control point c is then defined by

$$c = \frac{1}{\sum_i w_i} \sum_i w_i v_i \tag{2.12}$$

where w_i are the weights of the vertices v_i. In order to calculate the whole control polygon these basic rules are rotated and mirrored.

2.3.4 Evaluation of the Control Polygon

After the computation of the control polygon follows the evaluation of it's Bézier patch. The input of the evaluation are the normalized coordinates of the point inside the patch as well as the control polygon, consisting of 16 points in a 4 by 4 arrangement. The evaluation algorithm is straightforward: it computes the weighted sum of the vertices c_{ij} in u-direction and then the weighted sum of four results in the v-direction. This is done using the basis functions of a Bézier patch as the weights of the vertices:

$$B_i(u) = \frac{6}{i!(3-i)!}u^i(1-u)^{3-i} \tag{2.13}$$

$$B_j(v) = \frac{6}{j!(3-j)!}v^j(1-v)^{3-j} \tag{2.14}$$

These are the i-th and j-th Bernstein polynomials in u and v direction. The coefficients 3 and 6 in the formula result from the fact that cubic Bézier patches are used. The calculation of a point on the Bézier patch is then given by:

$$p(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} B_i(u)B_j(v)c_{ij} \tag{2.15}$$

where c_{ij} is the control point at position (i,j) in the 4 by 4 grid.

3 Gaussian-Product Subdivision on the GPU

In this chapter the implementation of the GPS algorithm on the GPU will be discussed. Information on the OpenGL pipeline will be given first, then the utilization of this pipeline for the realization of Gaussian-product subdivision will be focused on. Lastly the computation of the normal vectors in this setting is explained.

3.1 OpenGL Tessellation

In Figure 3.1 the full OpenGL pipeline is displayed with its optional steps. Reducing it to the required steps, the vertex shader, rasterization and fragment shader are left. The vertex shader is used to project the vertices to the image plane, the rasterization then transforms the continuous primitives into discrete fragments the size of pixels. Finally, the fragment shader performs the shading on those fragments. Each pixel gets the color of the fragment closest to the screen. The geometry shader is an optional step which gets a primitive as input and can output a different number of primitives, even of different types. The tessellation stage of the OpenGL Pipeline is an optional step, which subdivides input patches into smaller primitives. It consists of two programmable shaders, the tessellation control shader (TCS) and the tessellation evaluation shader (TES) and a fixed function part, which performs the actual subdivision. The two shaders are explained in more detail in the following sections.

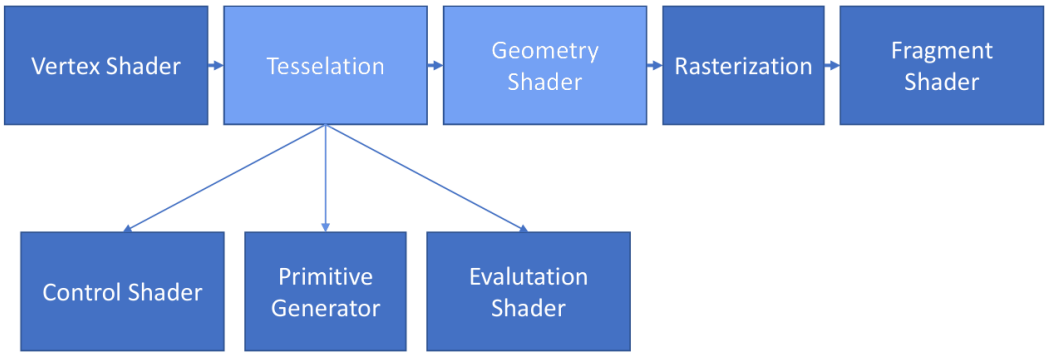


Figure 3.1: The full OpenGL pipeline with all optional stages in light blue

3.1.1 Tessellation Control Shader

The tessellation control shader (TCS) determines in how many primitives the patch gets subdivided. It is also responsible for patch continuity and applying transformations to input vertices if needed. The definition of the tessellation granularity is defined in two arrays in the GLSL language:

```
float gl_TessLevelOuter[4]
float gl_TessLevelInner[2]
```

Here the *gl_TessLevelOuter* array defines the number of divisions of the outer edges of a primitive and the *gl_TessLevelInner* defines the number of divisions on the inside. Figure 3.2 illustrates this concept for a quad and a triangle mesh. In this example the outer tessellation levels are set as follows: 2, 3, 4, 5, beginning at index 0 (in this case the left side) and then going around counter-clockwise. The numbers indicate the number of line segments after the tessellation on the outer quad. The inner tessellation levels are: 5, 4. The inner levels define the number of line segments on the inside of the quad. The first value indicates the number of line segments defined across the quad horizontally, and the second value the number of segments defined across the quad vertically. For triangle meshes, the outer subdivision levels are defined similarly, only with three subdivision numbers provided. The first level is used for the side starting at vertex index 0 (here the left side) of

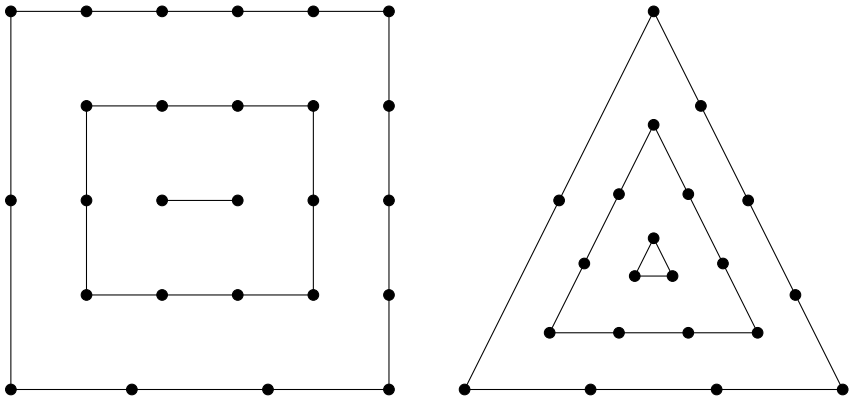


Figure 3.2: Tessellation example of a quad and triangle patch

the outer triangle, the second level is used for the bottom side and the last level is used for the right side. There is just one inner level for the triangle and it defines the number of patch segments going along the sides of the inner triangle. In the figure there are five such segments.

When using the tessellation stage, specifying a custom TCS is optional. There is a standard TCS, that can be used, which just sets the tessellation granularity, as defined by the program and then terminates. When using the standard TCS, the inner and outer tessellation levels can be defined by the GL API function:

```
void glPatchParameterfv(GLenum pname,
        const GLfloat *values);
```

where *pname* is GL_PATCH_DEFAULT_OUTER_LEVEL and *values* points to an array of length 4 for the outer levels, or *pname* is GL_PATCH_DEFAULT _INNER_LEVEL and *values* specifies an array of length 2 to define the inner levels.

3.1.2 GPU Tessellator

The tessellator is the fixed function part of the tessellation stage. It creates new vertices with the given input information from the tessellation control shader and the input vertices. The main parameters defining the tessellation are the tessellation levels defined by the CPU code or the tessellation control shader, and the tessellation spacing defined by the TES and the output primitive. The TES also defines the winding order of the output primitives as clockwise or counter-clockwise.

3.1.3 Tessellation Evaluation Shader

The tessellation evaluation shader determines the actual position of the tessellated vertices in 3D space. As inputs it gets the boundary vertices of the face and the 2-dimensional parameter-space coordinates of the vertex inside the patch. In the default case, the actual position of the vertex is then a linear combination of the input vertices, using the 2-dimensional coordinates as factors. Moreover, it defines the type of input primitive it gets and the winding order of the input vertices. Lastly, it defines the spacing of the tessellated vertices. All this is defined by the *layout* specification in the GLSL shader:

```
layout(quads, equal_spacing, ccw) in;
```

This sets the input primitive type as quadrilaterals, the spacing as equal spacing and the winding order as counter-clockwise. There are three different kinds of vertex spacing: equal spacing, odd fractional spacing and even fractional spacing. The effect of even and odd fractional spacing is shown in Figure 3.3. When even fractional spacing is set, if the tessellation level is divisible by 2 the spacing corresponds to the equal spacing (e.g., see subdivision level 2), if not the vertices are a linear combination of the previous equal spacing and the next one. This is the same for odd fractional spacing, with the difference that the spacing is the same as equal spacing when the tessellation level is not divisible by 2 (e.g., see subdivision level 3).

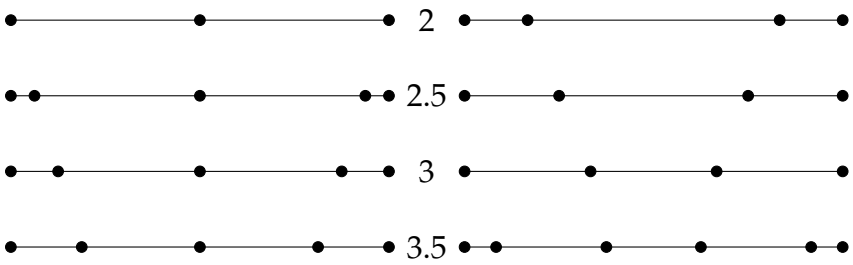


Figure 3.3: The differences between even (left) and odd (right) fractional spacing with different subdivision levels (middle)

3.2 Implementation of Real-Time Subdivision in the OpenGL Pipeline

In Figure 3.4 the interaction of each tessellation shader stage with the GPU Memory can be seen. The steps proposed by Brainerd et al. [2] are mainly implemented in the tessellation control shader and the tessellation evaluation shader. The GPU memory is divided in two parts (per patch memory and global memory) in order to illustrate the different kinds of memory on the GPU, which have not only different sizes but also different data throughputs. The global memory is much slower than the per-patch memory, which lies in the shared memory.

3.2.1 Tessellation Control Shader

The tessellation control shader is used, apart from its usual function of defining how the tessellator divides up the patch, to perform the necessary subdivision steps on the GPU. This needs to be done a fixed number of times and thus is implemented using the usual subdivision formulas. This shader stage uses the μ and Σ buffers, holding the mean and covariance data of the vertices and performs the transformation into the 9-dimensional space when reading this input data. The high-dimensional data is then stored efficiently in a 4x3 matrix. Note that, although the 9 floats should fit into a 3x3 matrix, the OpenGL pipeline has trouble processing datatypes whose

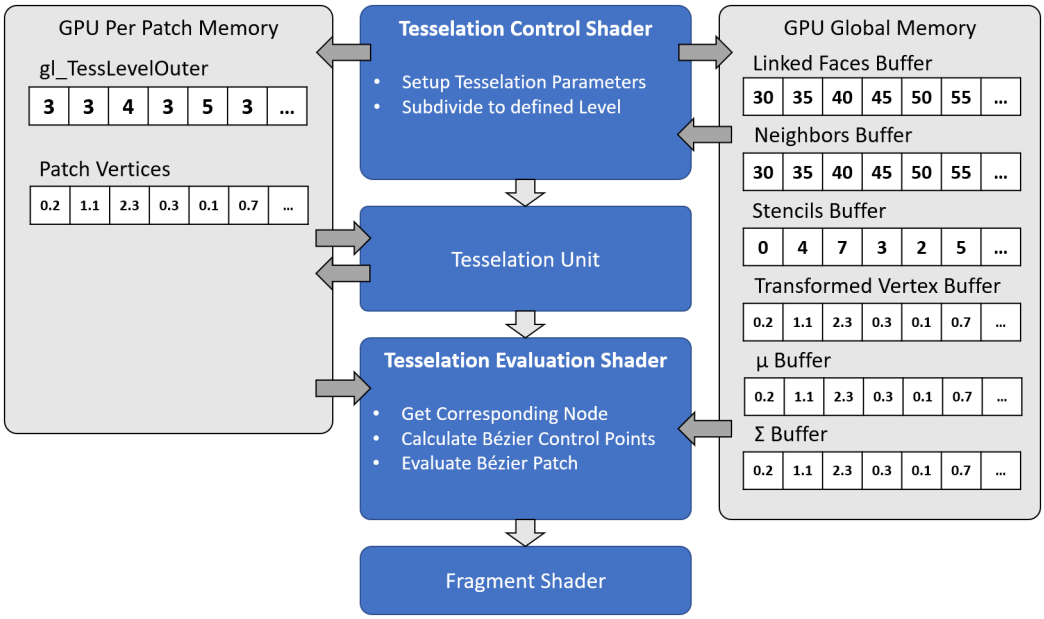


Figure 3.4: Interactions between the tessellation shader stages and the GPU memory.

size is not a multiple of four, which is usually addressed by an appropriate buffer alignment of the data.

The shader further uses the *Neighbors* buffer and the *Linked Faces* buffer to find the necessary vertices for the subdivision calculation of each face. Since the tessellation control shader is executed once per vertex but is needed only once per face, three threads get discarded and one is used for the subdivision calculation. The three discarded threads just pass on the input positions and the valence vertex attribute. The subdivided data gets written to the *Transformed Vertex* buffer. In order to write them to the correct index (the one used in the CPU subdivision), the *Neighbors* buffer gets consulted. The order of the neighbors is equal to the order of the edge vertices in the global buffer holding all vertex positions after the subdivision step. The required vertex data used by each face was buffered into local arrays in each per-face thread. This slightly improved performance and eliminated a potential bottleneck of this step.

Since the tessellator is a fixed function, it is used just to compute the correct

per patch coordinates for the resulting tessellated point. This is set up in the tessellation control shader. The coordinates are then being used in the tessellation evaluation shader stage.

3.2.2 Tessellation Evaluation Shader

In the tessellation evaluation shader the exact evaluation of the vertex positions is done. The input data is set to be an equally spaced quad mesh using counter-clockwise orientation of the faces. This is done because as a result uniformly tessellated quad is needed. The shader then takes the generated coordinates and performs the lookup into the quad-tree datastructure in order to find the corresponding child node, where the 3D position of the output vertex is evaluated. Once this node is found, the information about what kind of node it is and its stencil reference is used to calculate the control polygon of the Bézier patch. For regular nodes, this is only done using the stencil reference, since all the required data resides in the *Stencil* buffer containing indices and weights of the vertices. These get read from the *Transformed Vertex* buffer and the linear combination shown in Equation 2.12 is computed. For a terminal node, the vertex information resides in the *Stencil* buffer as well, but there are no weights stored. The vertices lie in a linearized 5x5 array to get an easier access and to simplify the computation of the control points in a later step. The indices are read from the *Stencil* buffer and the vertex values are read from the *Transformed Vertex* buffer. After the calculation of the control point polygon of the Bézier patch, the evaluation follows next. The calculation is again a linear combination of the control points using the basis functions of the Bézier patch and the coordinates of the vertex inside the patch. The computed 9D vertex position is then transformed back to 3D space and passed on down the OpenGL pipeline.

3.3 Buffer Setup

The setup of the buffers used in the calculations above is important, since it determines the complexity of the calculation and the effectiveness and

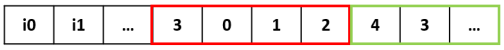


Figure 3.5: Visualization of the structure of the *Neighbors* and *Linked Faces* buffer. $i0$, $i1$ being the indices in this buffer of the 0th and 1st vertex. Framed in red is the vertex information, starting with the size of the slice (e.g. 3), followed by 3 vertex/face indices. The green frame contains the information of the next vertex.

speed of the access into these buffers.

Neighbors and Linked Faces buffer The *Neighbors* and *Linked Faces* buffer have a very similar structure and are therefore stored using the same layout. In Figure 3.5 the structure of these buffers is visualized. The first part of the buffer contains the indices of the individual arrays of neighbors contained in the second part of the buffer. The indices are placed corresponding to the vertex indices. Therefore, when accessing the neighbors or linked faces of a vertex, first the offset needs to be read, which is located at the index of the vertex. Then the number of neighbors/linked faces needs to be read at the offset position. Then the indices of the vertices/faces ($i0$, $i1$, $i2$,...) can be read starting at the position after the *size* (red array starting with size 3, followed by three indices). This is illustrated more precisely in listing 3.1.

Listing 3.1: Example of access to the *Neighbors* and *Linked Faces* Buffes

```
int offset = buffer[vertex_index];
int size = buffer[offset];
for(int j = 1; j <= size; j++){
  //access j-th neighbor of vertex_index
  int neigh_index = buffer[offset+j];
...
}
```

Vertex data input buffers The vertex data consists of a position μ and a 3x3 covariance matrix Σ for each vertex. Since the covariance matrix is symmetric it can be stored more efficiently as two vec4 containing the upper

triangular part of the matrix. This tuple of vec4s are stored in the covariance (Σ) buffer and sorted by the vertex index. The position μ is also stored in a position buffer, sorted by the vertex index. In order to get the full information of the vertex, the position buffer needs to be read at the vertex index and the covariance buffer needs to be read at the vertex position times 2 and at the subsequent position. Listing 3.2 contains the code used for getting the vertex position, transforming it into 9D space and storing it in a mat4x3 for the use in the shaders. Since this code is executed in the tessellation control shader, the matrices are defined and indexed column first, so a mat3x4 in an OpenGL shader corresponds to a matrix with three columns and four rows.

Listing 3.2: Code for getting the vertex information

```
vec4 covo = covmat_array[idx*2];
vec4 cov1 = covmat_array[idx*2+1];
vec4 muw = position_array[idx];
mat3 cov = mat3(covo.x, covo.y, covo.z, covo.y, covo.w,
cov1.x, covo.z, cov1.x, cov1.y);
mat3 inv_cov = inverse(cov);
vec3 trans_mu = inv_cov * muw.xyz;
mat3x4 g = mat3x4(inv_cov[0].x, 0, 0, trans_mu.x,
inv_cov[1].x, inv_cov[1].y, 0, trans_mu.y,
inv_cov[2].x, inv_cov[2].y, inv_cov[2].z, trans_mu.z);
```

Transformed Vertex buffer The *Transformed Vertex* buffer is filled at runtime by the GPU, so it just needs to be initialized by the CPU to the correct size and filled with zeros. The buffer consists of one mat4x3 element per vertex. It is initialized by a vector on the CPU, whose values are then passed to the GPU.

3.4 Normal Vector Computation

For shading the rendered subdivision surfaces, there are different ways of calculating the required normals on the GPU. The first and most common one is storing vertex normals with the position and then letting the GPU rasterizer interpolate between them, obtaining a normal information for each pixel. The second one is performed using the geometry shader of the OpenGL pipeline. The geometry shader is executed after the tessellation stage and can read information of the whole face and all its vertices. With this information it is easy to compute the face normal for each face, given the formula:

$$n = (v_2 - v_0) \times (v_1 - v_0) \tag{3.1}$$

where v_0, v_1, v_2 are the vertices of the triangle and the $\times$ operator denotes the cross product of two vectors. Since it computes only one normal across the face the individual faces need to be rather small in order for the render to look smooth and not pixelated.

The method used in our real-time rendering pipeline for covariance meshes computes the tangent vectors, based on Bézier spline information. In order to do this the derivative of the basis functions of the patch needs to be calculated. The basis functions for the parametric u direction, according to Equation 2.13, are as follows:

$$B_0 = (1-u)^3 \tag{3.2}$$
$$B_1 = 3\,(1-u)^2\,u \tag{3.3}$$
$$B_2 = (1-u)\,3\,u^2 \tag{3.4}$$
$$B_3 = u^3 \tag{3.5}$$

The basis functions for the v parameter are analogous to these. The derivatives of these in u-direction are:

$$\frac{\delta B_0}{\delta u} = -3\,(1-u)^2 \tag{3.6}$$
$$\frac{\delta B_1}{\delta u} = 9\,u^2 - 12\,u + 3 \tag{3.7}$$
$$\frac{\delta B_2}{\delta u} = 3\,(2-3\,u)\,u \tag{3.8}$$
$$\frac{\delta B_3}{\delta u} = 3\,u^2 \tag{3.9}$$

The derivatives in the v direction are analogous to these, since u and v are independent. The tangent T_u in u-direction is then defined as:

$$T_u = B_0(v)T_{u,0}(u) + B_1(v)T_{u,1}(u) + B_2(v)T_{u,2}(u) + B_3(v)T_{u,3}(u) \tag{3.10}$$

where

$$T_{u,i}(u) = \frac{\delta B_0}{\delta u}(u)p_{i,0} + \frac{\delta B_1}{\delta u}(u)p_{i,1} + \frac{\delta B_2}{\delta u}(u)p_{i,2} + \frac{\delta B_3}{\delta u}(u)p_{i,3} \tag{3.11}$$

and $p_{i,j}$ are the control points of the Bézier patch. Using the derivatives in u and v direction, the two tangent vectors are calculated in the tessellation evaluation shader. Their cross product finally defines the normal vector at this parametric position. This method has the advantage that the normal vectors also get interpolated by the GPU rasterizer and appear generally smoother since every pixel is shaded using a interpolated normal vector instead of a constant normal per face.

The extension of this algorithm to the GPS is not trivial, since the basis function is defined in 9D space and needs to be transformed back. This transformation back to 3D space is not the same as transforming back a 9D vertex. Since the definition of the tangent vector is the derivative and the result should be the derivative of the transformed vector, the transformation needs to be adjusted. This is discussed in the appendix of the paper [17] by Preiner et al. Taking the 9D vector $q = (\hat{q}, \overline{q})$, the 3D tangent vector μ_u in u-direction is given by:

$$\mu_u = \Sigma(\overline{q}_u - [\hat{q}_u]\mu) \tag{3.12}$$

where $\overline{q}_u$ and $\hat{q}_u$ denote the partial derivatives of the respective components of q.

3.5 Implementation Details

In this section some interesting details are discussed, which came up during the implementation of the rendering pipeline. They range from hardware limitations to quirks of the OpenGL pipeline.

First, using Shader Storage Buffer Objects (SSBOs) one should avoid any datatype whose size is not divisible by four. The GPU is designed to most efficiently process vec4 data, the usual datatype used for position data. Providing buffers containing other datatypes like vec3 or mat3, without proper data alignment to 32 bit chunks will therefore lead to problems when reading data.

The next thing could be a hardware limitation of Nvidia-branded GPUs. No AMD GPUs were tested. As with the version used at the time of this work (OpenGL 4.6), there is a limit on how large inline arrays of floats can be in an GLSL shader. This limit just applies to float arrays and limits any array to contain not more than 124 floats. There is a compiler error if a shader contains a larger array anywhere in the code. This limit just applies to arrays directly declared in the shader and not SSBOs or Vertex Buffer Objects (VBOs).

Another important implementation aspect is the way GLSL shaders pass parameters to functions. Every parameter is passed by value, so every array that is passed to a function is copied in memory and then passed. When many function calls are made with many parameters and further many of them being arrays, this introduces a lot of allocated memory. In order to circumvent this memory allocation, many functions were redefined as a *define* macro. This prevents an actual function call and with that the memory allocation.

4 Results

In this chapter the results of the implemented rendering pipeline are presented, interpreted and analyzed. First performance tests are executed for different models and tessellation levels, then comparisons with a CPU reference implementation of GPS are done.

4.1 Material Capture Shading

All renderings in this section use a shading technique called Material Capture Shading (MatCap) in order to shade the resulting surfaces. Using the normal of the surface, it computes the reflecting vector in view space and projects it onto the xy plane. These 2-dimensional coordinates are used to look up the color of the shaded pixel in a reflective texture map. This texture contains a rendered sphere, centered in the middle and filling out the whole image. An example can be seen in Figure 4.1.

This is used to get a realistic shading with less computational effort than methods that produce similarly realistic results (e.g. Raytracing). To generate these textures, a sphere needs to be rendered having the desired properties. Another method of obtaining such a texture is to actually capture a real life spherical object placed in a desired environment.

4.2 Visual Comparision

First, in order to show that all valences are supported and are calculated correctly, test meshes were made. The renders of these test meshes can be

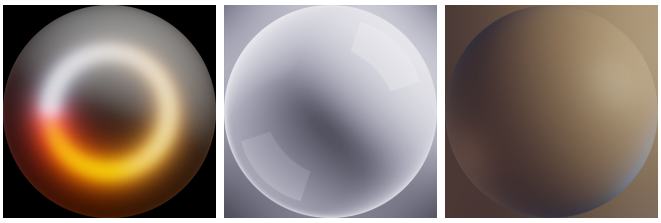


Figure 4.1: The 3 MatCap texture used in the results section.

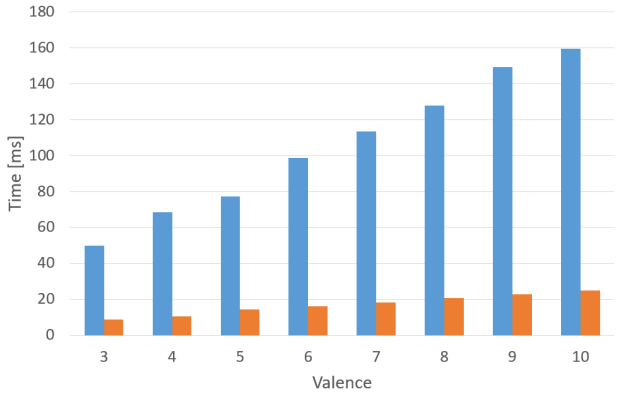


Figure 4.2: Preprocessing times (blue) and render times (orange) for the valence test meshes.

seen in Figure 4.3 along with their CPU subdivision equivalent. Preprocessing and render times are displayed in Figure 4.2. As the tessellation level was set to 6 in the rendering process of these images, these times are to be expected.

A detailed comparison to a CPU reference subdivision can be seen in Figure 4.4. Apart from the GPU and CPU renderings, a difference image can be seen to the right. This image was calculated using the per-pixel world-space position data, read from an additional renderbuffer. Then the difference was calculated based on the Euclidean norm of the difference between the image pixel vectors. This was then divided by the diagonal of the bounding box of the mesh and finally multiplied with a factor (200 in this case) and written to a gray-scale image. Another interesting render can be seen in Figure 4.5.

Figure 4.3: All valency test meshes, from valency 3 to valency 10 rendered with real time covariance rendering (top) and their base mesh and covariances (bottom).

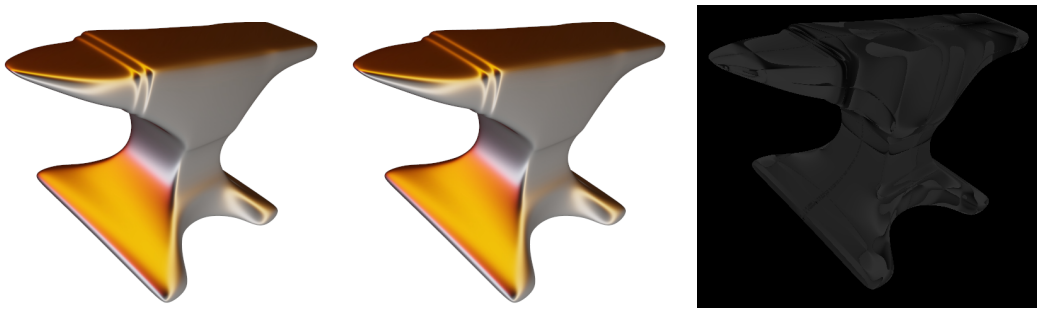


Figure 4.4: The anvil mesh rendered by our GPU subdivision (left), by the reference CPU subdivision (middle) and the difference image between the two (right), equal pixels are black and pixel that differ are colored. The difference image was enhanced by factor 200.

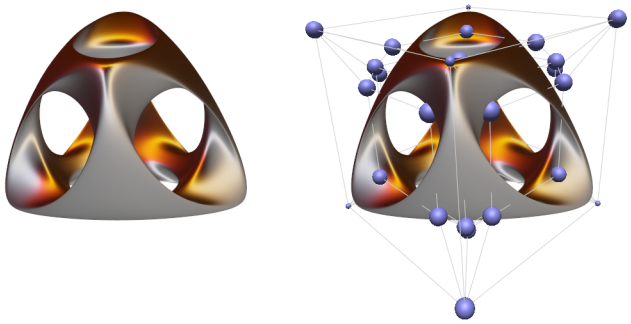


Figure 4.5: The wirecube mesh subdivided and rendered by the GPU.

name	#vertices	#faces	preprocessing time
cube	8	6	68.77 ms
wirecube	40	48	209.14 ms
cylinder	30	28	114.46 ms
anvil	246	244	1346 ms

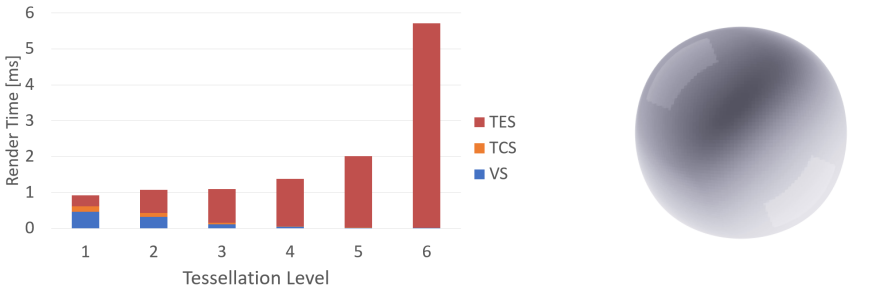
Table 4.1: List of meshes used in the performance analysis, showing their number of vertices, faces and their preprocessing time.

4.3 Performance Tests

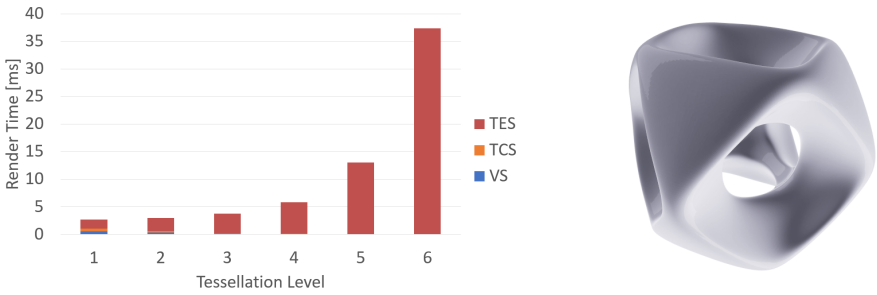
The evaluation of the performance of the system was done in different ways. They were all done on a desktop PC with a Ryzen 1800X CPU and a Nvidia GeForce GTX 1080 GPU with Nvidia Driver version 445.75. Firstly, the performance of each stage (preprocessing, rendering) was captured and then plotted in a time stack diagram. This was done for different meshes and all supported tessellation levels in order to compare not only different mesh topologies, but also different tessellation levels against each other. The meshes used in the performance analysis are listed in Table 4.1.

Figure 4.6 shows the time stacks for the cube, wirecube, cylinder and anvil mesh respectively, along with the renders for each mesh. The colors represent each OpenGL pipeline stage, from bottom to top: vertex shader, tessellation control shader, tessellation evaluation shader. Because we were not able to directly measure the time spent on each shader stage, we have read out the number of shader executions per stage using *Nvidia Nsight*. The height of each color represents the relative number of executions of each shader stage, in order to approximate the execution time of each shader stage.

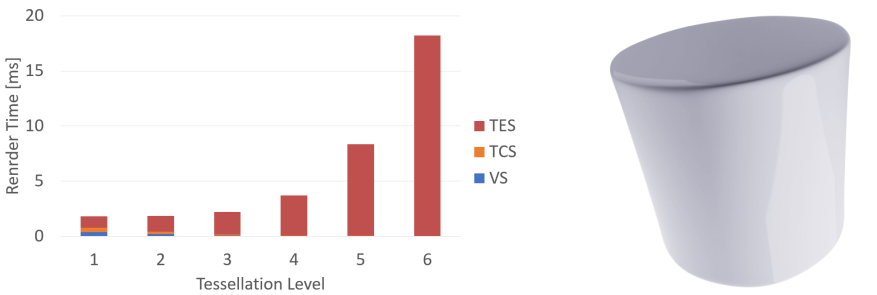
The time stack figures exhibit a roughly exponential growth of render time with increasing tessellation level, which is expected since the number of faces, and with it the number of shader executions, also grows exponentially. Moreover, it can be seen that depending of the number of vertices in the base mesh, the relative amount of tessellation evaluation shader executions (red) is different. When there are many vertices in the base mesh (anvil



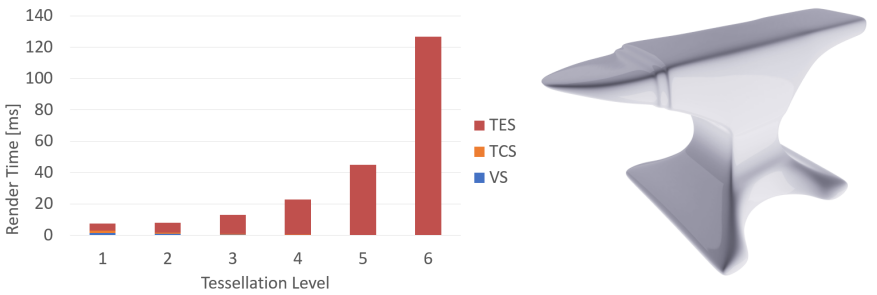
(a) cube mesh



(b) wirecube mesh



(c) cylinder mesh



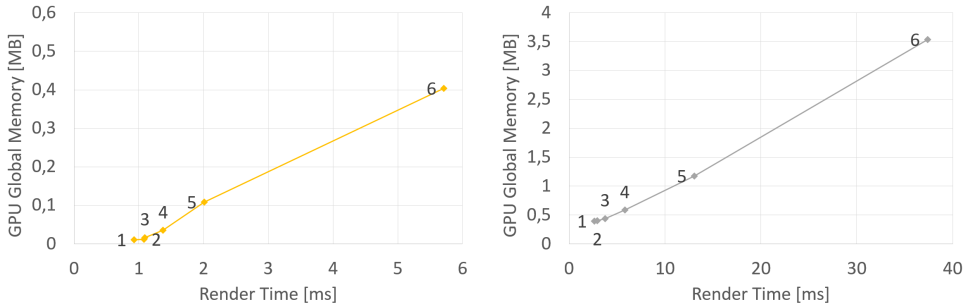
(d) anvil mesh

Figure 4.6: Render time analysis for different meshes

mesh), the relative amount is greater. When there are less vertices in the base mesh (cube mesh) the relative amount is smaller.

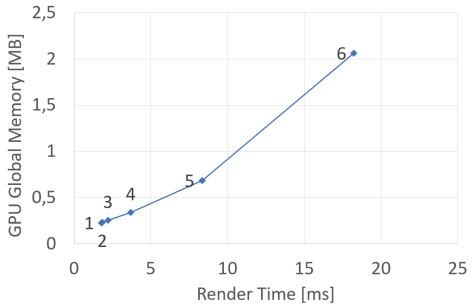
Another interesting aspect to look at is the GPU memory consumption of the rendering pipeline. In Figure 4.7 the global GPU memory was estimated and plotted against the render time of each mesh and each tessellation level. Each mesh is shown in a separate plot and also in one common diagram to compare them to each other.

As seen before, the render time behaves like an exponential function when increasing the tessellation level, and so does the memory consumption, but not as extreme as the render time. So in the plots the render time vs memory consumption can be roughly described as a linear function. Depending on how fast the memory consumption increases, the slope of the function is steeper (anvil) or flatter (cube). This means that when increasing the tessellation level, the memory consumption increases faster when rendering the cube mesh, than it does when rendering the anvil mesh. The slopes of the wirecube and cylinder meshes are in between the two, the wirecube mesh exhibits a flatter slope than the cylinder mesh. This phenomenon is based on the number of vertices in the base mesh. For a base mesh with a larger number of vertices like the anvil mesh, the memory increases faster than the render time and therefor the slope is steeper. For meshes like the cube mesh with very few vertices in the base mesh, the render time increases faster and the slope is flatter. How fast the render time or memory consumption increases mainly depends on how many vertices there are in the base mesh. The difference is that the render time is just proportional to the number of vertices and the memory consumption can be directly computed using the number of vertices and topology information. This is due to the fact that just the number of shader executions increases with the number of vertices. The scheduler on the GPU takes care to execute the shader programs in parallel as efficiently as possible, and therefore creates this difference in growth.

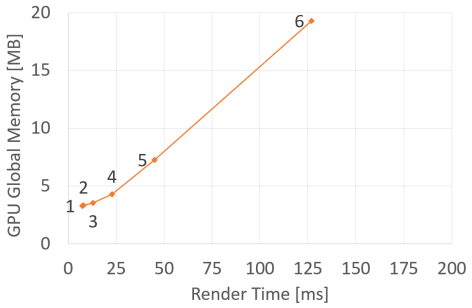


(a) cube, 8 vertices

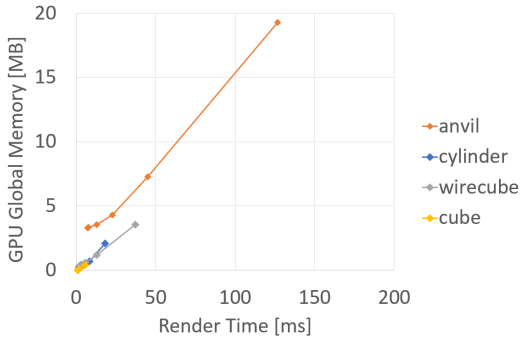
(b) wirecube, 40 vertices



(c) cylinder, 30 vertices



(d) anvil, 246 vertices



(e) all combined

Figure 4.7: Plots of memory consumption over render time for all performance evaluation meshes. The markers highlight the measurement points at the tessellation levels (1-6).

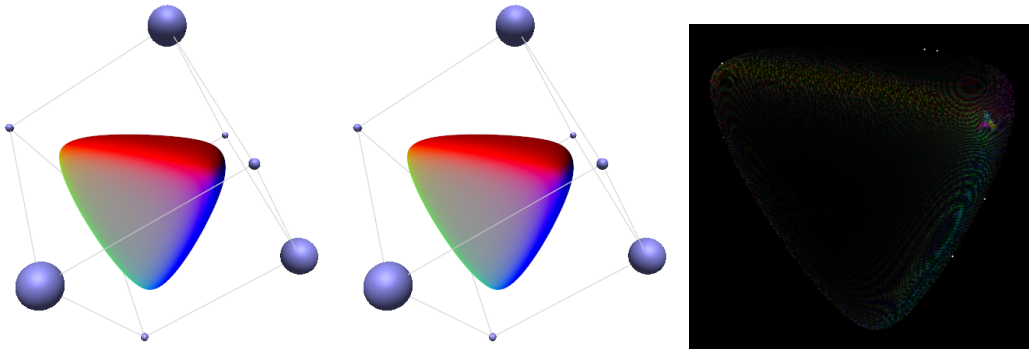


Figure 4.8: The normal vectors of the distorted cube mesh encoded in the colors of the mesh, for the GPU render (left), the CPU subdivision (middle) and their difference image (right), equal pixels are black and pixel that differ are colored. The difference image was enhanced by factor of 5000.

4.4 Normals Comparison

Another important result to check and evaluate is the correctness of the resulting normals of the limit surface. This not only is important for the completeness of the system, but also essential for many different shading techniques. In Figure 4.8 a comparison of the normals can be seen. The shader used in these renderings encodes the normal vector as a color on the mesh, so that these values can be compared. The image was computed taking the normals directly from an additional renderbuffer and writing them to the disk. Then we have calculated the component-wise difference between the rendered GPU and the reference CPU images for each of the three components, multiplied it with a factor (in this case 5000) and wrote them out to an image file where the xyz components of the difference vector were encoded in the rgb channels of each pixel. It can again be seen that the difference image is mostly black, meaning almost identical images. Furthermore, the computation of the normals does not result in a significant increase of render time since it just requires the computation of the tangent vectors given in Equation 3.10, the transformation back given in Equation 3.12 and the cross product of the vectors.

4.5 Conclusion

In conclusion the developed GPU subdivision algorithm works as well as the CPU subdivision, as can be seen in the figures above. The difference images between the rendered results indicate only a low measurable difference, by pure visual inspection the renders are indistinguishable. There are slight differences, resulting from the tessellation of the patches, especially near the extraordinary vertices, due to the continuity there being just C^1.

For the 3D models analyzed in this thesis, the real-time performance requirement is fulfilled up to a tessellation level of about 5, where the subdivision computation time exceeds 16.67 ms (60 Hz). For most meshes, real-time performance is achieved when the tessellation level is set to medium values (3-4). The actual maximum tessellation level that just allows real-time performance depends on the number of vertices in the base mesh and the computing power of the GPU.

5 Geometric Discussion

In this chapter the resulting geometry of the application is analyzed and the continuity of the limit surface is derived and discussed. This is important, since a robust method should produce a continuous surface without faults or discontinuities. Furthermore, another important aspect of a subdivision scheme is analyzed, the calculations of the surface normals.

5.1 Continuity of Gaussian-Product Subdivision

The continuity and smoothness of the covariance mesh subdivision are important properties for real world applications. The analysis of Preiner et al. [17] is recapitulated in the following paragraphs. The covariance mesh's vertices are defined as follows:

$$\Theta_i = (\mu_i, \Sigma_i) \tag{5.1}$$

where μ_i is the mean of a Gaussian distribution and Σ_i is its covariance matrix. The map $F(\mu, \Sigma) \mapsto q$, where $q = (\hat{q}, \overline{q})$, transforming the covariance vertices into the higher-dimensional space, is defined in Equation 2.10.

Moreover, the covariance matrices Σ_i are positive-definite matrices and so are their inverses Σ_i^{-1}. Therefore, their images $\hat{q}_i$ all lie within a conical region $Q_{pd} \subset Q$ of the 6-dimensional subspace spanned by the coefficients of the first part of the basis $b(x)$ in Equation 2.9 [10]. The boundary δQ_{pd} of this cone includes just singular, non-invertible symmetric matrices, where the map $F(\mu_i, \Sigma_i)$ and its inverse $F^{-1}(\mu_i, \Sigma_i)$ are not defined. So discontinuities occur when the corresponding hypersurface S^* of the covariance mesh intersect that boundary. These intersections can be avoided if the

hypersurface S^* at all times stays withing the convex hull of the input mesh, which is true, when using set of convex weights α_i. This implies the use of the non-negative weights $\alpha_i \geq 0$, as given in a linear subdivision scheme like Catmull-Clark. In conclusion, when performing probabilistic Gaussian-Product subdivision, using the weights α_i of such an approximative scheme L, the resulting limit surface $S = \{F_\mu^{-1}(q) : q \in S^*\}$ is continuous, since the inverse map F_μ^{-1} is well defined and C^∞ differentiable over the cone Q_{pd}. The limit surface further adopts the smoothness properties of the hypersurface, that is the continuity orders at the extraordinary vertices are the same as defined in the linear subdivision scheme L.

5.2 Continuity Properties in our Real-Time Pipeline

After in the last section it was shown that Gaussian-Product subdivision inherits the properties of the underlying linear subdivision model, it is important to show this continuity property for the real-time subdivision pipeline as well.

The surface is constructed using Bézier patches. The continuity within the patches is given, since its basis functions (defined in Equation 2.13) are continuous. The patch borders are of a greater interest. The border edges of the patches are C^0 continuous if the control polygons of the bordering patches have equal edge curves, meaning the four control points at the border have to be equal. Looking at the way the control polygons are calculated, information about the edge points of the control polygons are shared between edges. This means that both calculations are using the same data and therefore produce the same result.

For C^1 continuity, the analysis has to go further. Ball and Storry [1] did a fundamental geometric analysis of the approximation of subdivision surfaces looking at their subdivision matrices in an average case with a general choice of weights. They achieved this by doing a discrete Fourier transformation on the vertices and then reformulated the subdivision matrix in this Fourier-space. Computing the eigenvalues and eigenvectors of this matrix,

the researchers showed that the approximation of the subdivision is C^2 smooth everywhere except it has C^1 continuity at extraordinary vertices, if and only if the dominant eigenvalue is greater than the eigenvalue involving α, the weight of the original vertex when computing the vertex update.

Ball and Storry showed this on iterative subdivision algorithms, but since a Catmull-Clark subdivision scheme converges to a Bézier patch and the first steps in our real-time subdivision pipeline involve actual Catmull-Clark subdivision, the proof is also valid in our pipeline. Therefore, our subdivision pipeline provides C^2 continuity everywhere except extraordinary vertices where it has C^1 continuity.

Combining both proofs above we can conclude that, since the Gaussian-Product subdivision inherits the properties from the underling linear subdivision scheme and the approximation of the subdivision surface by Bézier patches was shown and proven by Ball and Storry [1], the resulting limit surfaces of our pipeline are C^2 continuous everywhere except at extraordinary vertices, where it has C^1 continuity.

6 Outlook

The most important task for future work is to introduce triangle meshes to the application. There are a few things that need to be implemented in order to achieve this: The preprocessing step needs to be altered to be able to handle triangles and some data-structures need to be updated and adjusted. Further down the pipeline, the tessellation control shader needs to be able to handle Loop triangle subdivision and the tessellation evaluation shader has to handle the input as a triangle and not a quadrilateral. Moreover, the representation of a patch as a Bicubic Bézier surface needs to be extended to be able to handle triangles. This could either be done using triangular Bézier patches, or by switching to Gregory patches [9].

Another interesting aspect of the covariance mesh subdivision algorithm is the higher-dimensional cone, containing all valid symmetric covariance matrices. The analysis of what happens with the corresponding 3D vertex if one higher-dimensional vertex lies outside this cone or on the boundary is a very interesting research topic. Although the inverse is not defined for these matrices outside of this cone, some numerical replacement for the inverse operation could be used instead.

In order achieve further performance gains, the GPU subdivision algorithm can be updated. There has been some interesting research in this topic, since, when set up just right, it can be sped up quite much on the GPU, in comparison with a traditional CPU subdivision. Mlakar et al. [14] showed some promising results and a rather interesting setup, using a sparse mesh matrix, an edge matrix and a face matrix. Using sparse matrix-vector multiplication and sparse matrix-matrix multiplication, they subdivided the mesh on the GPU. Since sparse matrix-vector and matrix-matrix multiplication can be implemented very efficiently on the GPU, they achieved real-time performance.

Further changes in some internal data structures could yield more preprocessing performance increase. This could also increase performance further down the pipeline.

Bibliography

[1] A. A. Ball and D. J. T. Storry. "Conditions for Tangent Plane Continuity over Recursively Generated B-Spline Surfaces." In: *ACM Trans. Graph.* 7.2 (Apr. 1988), pp. 83–102. ISSN: 0730-0301. DOI: 10.1145/42458.42459 (cit. on pp. 1, 40, 41).

[2] Wade Brainerd et al. "Efficient gpu rendering of subdivision surfaces using adaptive quadtrees." In: *ACM Transactions on Graphics (TOG)* 35.4 (2016), pp. 1–12 (cit. on pp. 3, 12, 21).

[3] Edwin Catmull and James Clark. "Recursively generated B-spline surfaces on arbitrary topological meshes." In: *Computer-aided design* 10.6 (1978), pp. 350–355 (cit. on pp. 2, 5, 6, 14).

[4] Tony DeRose, Michael Kass, and Tien Truong. "Subdivision surfaces in character animation." In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. 1998, pp. 85–94 (cit. on pp. 2, 7).

[5] Daniel Doo. "A subdivision algorithm for smoothing down irregularly shaped polyhederons." In: *Computer Aided Design* (1978), pp. 157–165 (cit. on pp. 2, 5).

[6] Daniel Doo and Malcolm Sabin. "Behaviour of recursive division surfaces near extraordinary points." In: *Computer-Aided Design* 10.6 (1978), pp. 356–360 (cit. on pp. 5–7).

[7] Nira Dyn, David Levin, and John A Gregory. "A 4-point interpolatory subdivision scheme for curve design." In: *Computer aided geometric design* 4.4 (1987), pp. 257–268 (cit. on pp. 6, 7).

[8] Nira Dyn, Hed. S., and David Levin. *Subdivision schemes for surface interpolation*. Tel Aviv University, 1993 (cit. on p. 2).

[9] John A. Gregory. "Smooth Interpolation without Twist Constraints." In: *Computer Aided Geometric Design*. Ed. by Robert E. Barnhill and Richard F. Riesenfeld. Academic Press, 1974, pp. 71–87. ISBN: 978-0-12-079050-0. DOI: 10.1016/B978-0-12-079050-0.50009-6 (cit. on p. 43).

[10] Richard D Hill and Steven R Waters. "On the cone of positive semidefinite matrices." In: *Linear Algebra and its Applications* 90 (1987), pp. 81–88 (cit. on p. 39).

[11] Leif Kobbelt. "Square Root 3 Subdivision." In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 2000, pp. 103–112 (cit. on pp. 2, 6, 7).

[12] Leif P Kobbelt. "A subdivision scheme for smooth interpolation of quad-mesh data." In: *Procs of EUROGRAPHICS* 98 (1998) (cit. on p. 6).

[13] Charles Loop. "Smooth subdivision surfaces based on triangles." In: *Master's thesis, University of Utah, Department of Mathematics* (1987) (cit. on pp. 2, 6).

[14] Daniel Mlakar et al. "Subdivision-Specialized Linear Algebra Kernels for Static and Dynamic Mesh Connectivity on the GPU." In: *Computer Graphics Forum*. Vol. 39. 2. 2020, pp. 335–349 (cit. on p. 43).

[15] Matthias Nießner et al. "Feature-adaptive GPU Rendering of Catmull-Clark Subdivision Surfaces." In: *ACM Trans. Graph.* 31.1 (Feb. 2012), 6:1–6:11. ISSN: 0730-0301. DOI: 10.1145/2077341.2077347 (cit. on pp. 3, 12).

[16] Anjul Patney and John D. Owens. "Real-time Reyes-style Adaptive Surface Subdivision." In: *ACM SIGGRAPH Asia 2008 Papers*. SIGGRAPH Asia '08. Singapore: ACM, 2008, 143:1–143:8. ISBN: 978-1-4503-1831-0. DOI: 10.1145/1457515.1409096 (cit. on pp. 3, 12).

[17] Reinhold Preiner, Tamy Boubekeur, and Michael Wimmer. "Gaussian-Product Subdivision Surfaces." In: *ACM Transactions on Graphics* 38.4 (July 2019), 35:1–35:11. ISSN: 0730-0301. DOI: 10.1145/3306346.3323026 (cit. on pp. 2, 7, 8, 11, 27, 39).

[18] Le-Jeng Shiue, Ian Jones, and Jörg Peters. "A Realtime GPU Subdivision Kernel." In: *ACM SIGGRAPH 2005 Papers*. SIGGRAPH '05. Los Angeles, California: ACM, 2005, pp. 1010–1015. DOI: 10.1145/1186822.1073304 (cit. on pp. 3, 12).

[19] Jos Stam. "Evaluation of loop subdivision surfaces." In: *SIGGRAPH'98 CDROM Proceedings*. Citeseer. 1998 (cit. on p. 3).

[20] Jos Stam. "Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values." In: *Siggraph*. Vol. 98. Citeseer. 1998, pp. 395–404 (cit. on pp. 3, 13).

[21] Denis Zorin, Peter Schröder, and Wim Sweldens. "Interpolating subdivision for meshes with arbitrary topology." In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. 1996, pp. 189–192 (cit. on p. 6).