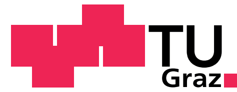


Daniel Mlakar, BSc

# High Performance Mesh Subdivision through Sparse Matrix Algebra 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

Ass.-Prof. Dipl.-Ing. Dr. techn. BSc Markus Steinberger
Dr. Ing. Rhaleb Zayer

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

Graz, February 2018

# 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 |
| --- | --- |

# Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Dissertation identisch.

Datum

Unterschrift

# Acknowledgments

I want to thank Markus Steinberger and Rhaleb Zayer for all their helpful advice and feedback throughout this project. No matter what time of day (or night) I turned to them with the problems I faced, they were always determined to help immediately. I also want to thank them for giving me the freedom to do things the way I wanted to and that they provided guidance where it was needed.

I would like to thank my parents for supporting me in everything I do and in everything I want to achieve. They might be the most patient people I know.

# Abstract

Mesh subdivision has become an important task in geometry. It generates dense and smooth surface representations from rather coarse meshes by iterative refinement. Subdivision surfaces are nowadays a standard modeling tool in character animation for feature films. As the demand for more detailed meshes grows, there is a need for fast and efficient subdivision implementations. Despite the inherent parallelism of subdivision algorithms, their parallelization is a challenging problem, hence, to date no fully GPU enabled implementation exists. Even the current industry standard for mesh subdivision relies on serially precomputed subdivision tables, to be able to utilize some of the parallel computing power of modern graphics hardware. Traditional data structures for serial mesh processing often rely on linked lists. Computations therefore involve pointer chasing and scattered memory accesses, both harming performance, especially on parallel computing devices such as the GPU. In this thesis we propose the use of linear algebra primitives for mesh subdivision based on a sparse matrix representation for meshes. This enables for expressive algorithm descriptions, which can be efficiently parallelized on the GPU. While a straight forward implementation of the high level linear algebra formalizations already results in high performance subdivision implementations, tailoring the linear algebra kernel to account for the underlying structures allows to unleash the sheer power of modern graphics hardware to gain a substantial speedup. In this thesis we present linear algebra descriptions for $\sqrt{3}$, Loop and Catmull-Clark and our implementations surpass state of the art performance by up to three orders of magnitude. While the principles proposed in this thesis are applied to subdivision algorithms, they can be used for general mesh processing tasks to move from slow serial algorithms to high performance parallel implementations.

# Kurzfassung

Mesh Subdivision ist zu einer wichtigen Aufgabe in der Geometrieverarbeitung geworden. Grobmaschige 3D-Modelle können durch iteratives Verfeinern immer dichter und glatter gemacht werden. Subdivision ist heutzutage ein standard Modellierwerkzeug in der Charakteranimation für Animationsfilme. Da Meshes immer detaillierter sein sollen, stieg die Nachfrage nach schnellen und effizienten Subdivision Implementierungen. Trotz der inhärenten Parallelität von Subdivision Algorithmen ist deren Parallelisierung ein anspruchsvolles Problem, weshalb bis jetzt keine vollständig parallelisierte Implementierung auf der GPU existiert. Der aktuelle Industriestandard für Mesh Subdivision verwendet seriell vorberechnete Subdivision Tabellen um einen Teil der massiven parallelen Leistung moderner Grafikhardware nutzen zu können. Traditionelle Datenstrukturen zur Representation von Meshes, welche in der seriellen Meshverarbeitung Anwendung finden, basieren oft auf verketteten Listen. Berechnungen auf diesen Strukturen erfordern oftmals das Traversieren von Index- oder Pointerketten und verstreute Speicherzugriffe. Beides führt zu reduzierter Performance, besonders auf paralleler Hardware wie der GPU. In dieser Arbeit schlagen wir die Verwendung von Primitiven aus der Linearen Algebra zur Durchführung von Mesh Subdivision, unter Verwendung einer Sparse Matrix Representation für Meshes vor. Dies ermöglicht aussagekräftige Algorithmenbeschreibungen, welche effizient auf der GPU parallelisiert werden können. Während wir bereits mit einer einfachen Implementierung dieser Formulierungen sehr gute Performance erzielen, können wir weiters das Wissen über die zugrundeliegenden Strukturen der Matrix verwenden um die Effizienz maßgeblich zu steigern. In dieser Arbeit präsentieren wir Beschreibungen für $\sqrt{3}$, Loop und Catmull-Clark Subdivision in der Sprache von Linearer Algebra und unsere Implementierungen übertreffen State of the Art Performance um bis zu drei Größenordnungen. Während wir die hier präsentierten Prinzipien auf Mesh Subdivision anwenden, können sie auch verwendet werden um von langsamen seriellen Meshverarbeitungsalgorithmen auf schnelle parallele Implementierungen umzusteigen.

# Contents

**Acknowledgments** **iv**

**Abstract** **v**

**Kurzfassung** **vi**

**1 Introduction** **1**

**2 Related Work** **5**

**3 The Mesh Matrix** **9**
3.1 Mesh Matrix Basics . . . . . 10
3.2 Operations on the Mesh Matrix . . . . . 11

**4 Subdivision in the Language of Linear Algebra** **17**
4.1 $\sqrt{3}$-Subdivision . . . . . 18
4.1.1 Traditional Formulation . . . . . 18
4.1.2 Linear Algebra Formulation . . . . . 20
4.2 Loop Subdivision . . . . . 23
4.2.1 Traditional Formulation . . . . . 23
4.2.2 Linear Algebra Formulation . . . . . 26
4.3 Catmull-Clark Subdivision . . . . . 29
4.3.1 Traditional Formulation . . . . . 30
4.3.2 Linear Algebra Formulation . . . . . 33
4.3.3 Feature-Adaptive Subdivision . . . . . 36
4.3.4 Mesh reordering . . . . . 41

**5 Optimization of Mesh Matrix Operations** **44**
5.1 Reduced Mesh Matrix . . . . . 44
5.2 Implicit mapped SpGEMM . . . . . 45

5.3 Specialized SpMVs . . . . . . . . . . 47

**6 Evaluation 51**

6.1 Subdivision performance . . . . . . . . . . 52

6.2 Mesh reordering . . . . . . . . . . 56

6.3 Acknowledgments . . . . . . . . . . 57

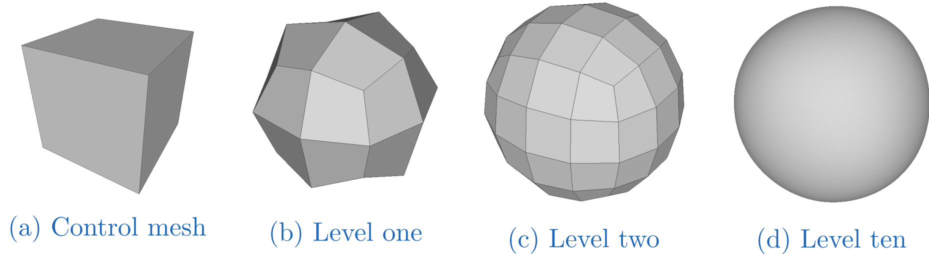
**7 Conclusion 58**

**Bibliography 59**

# 1 Introduction

Mesh subdivision is an old problem, already dating back more than four decades. It is a geometric operation which starts from a coarse mesh and iteratively refines it to get denser and smoother representations. The coarse mesh is also often called the control mesh because of the close relation to splines. Repeated refinement of a control mesh eventually causes the series of resulting meshes to converge to a limit surface, provided that the parameters governing the subdivision are chosen carefully. An example of a cube at different subdivision levels can be seen in Figure 1.1. Subdivision surfaces have various applications. They became a standard tool in feature film production for character animation [DKT98], as it is more pleasant to animate the control mesh and render the subdivision surface than to directly manipulate the vast amount of vertices in the final model. Subdivision models are also used to reduce the pressure on the rendering pipeline, by using coarse representation for meshes in the distance and only render the detailed model if it is close to the camera. A huge advantage of being able to generate high resolution models from rather coarse representations on demand is that only a small amount of data, the control mesh, has to be stored and streamed to the Graphics Processing Unit (GPU) for rendering, because it is well known that data transfer from and to the GPU is expensive. For general information on subdivision in geometric modeling, the reader is referred to the book by Warren et al. [WW01].

The strive for more detailed models with high resolution sparked the need for fast mesh subdivision. Not too long ago, Moore's law ensured that serial algorithms executed on the CPU doubled their performance every two years, as transistors where getting faster and smaller such that more of them could fit on a chip [Moo00]. Unfortunately, the free lunch is over [Sut05] as we hit the power wall and cooling CPUs becomes infeasible. Therefore, parallelization of algorithms has become a common way to speed up computations. While GPUs were exclusively used for image synthesis a few years ago, they are now



(a) Control mesh (b) Level one (c) Level two (d) Level ten

Figure 1.1: Visualization of a cube at different subdivision levels

a powerful platform for general purpose parallel algorithms. A GPU consists of several SIMD (single instruction-multiple data) processors, each capable of executing multiple threads at the same time. Many threads execute the same instructions but on different data. Due to the inherently parallel nature of subdivision algorithms they lend themselves for implementations that are executed on the SIMD hardware of modern graphics cards.

Traditional mesh representations used in serial algorithms on the CPU often rely on linked lists, to support fast adjacency queries. A widely used representation, the half-edge data structure [CKS98], which is a derivate of the winged-edge structure [Bau72], consists of a list of half-edges, each holding pointers to vertices, faces and other half-edges in its neighborhood. Depending on the implementation, details may vary, but a change to the topology in a half-edge like representation requires updating primitives in the local neighborhood and careful pointer handling to preserve consistency. Computations on linked list representations require pointer chasing, causing unbalanced workloads, which hurts performance of parallel implementations. The complexity of topological updates and the required inter thread communication to achieve fine grained parallelism point towards the necessity of different mesh representations for parallel subdivision on the GPU. To reduce the communication between threads working on the subdivision of different faces, the mesh can be split into small fragment meshes which can be subdivided independently of each other [SJP05]. These fragments contain the face to be subdivided and usually one layer of surrounding faces to ensure independence of other fragments. Fragments for neighboring faces in the mesh have a large overlap, and therefore contain duplicate data which increases overall memory footprint. Fragments are usually created in a preprocessing step on the CPU. To avoid the patch generation and data redundancy, data

structures to support parallel breadth first subdivision have been developed [PEO09]. In contrast to the patch based depth first approaches where fragments were subdivided to their final level independently of each other, breadth first subdivision ensures that the whole mesh is at the same refinement level before advancing to the next level. These approaches are more similar to traditional methods using linked list representations and store topological information on a per edge basis. To enable these structures to serve adjacency queries fast, each edge has information about its neighboring faces and edges, which again causes the structures to be large in size. Some subdivision implementations on the CPU [BS02] as well as on the GPU [Nie+12] rely on subdivision tables, which hold for each primitive in the refined mesh, the indices of vertices in the control mesh, which contribute to its calculation. While the evaluation of the refined vertex data can be done efficiently in parallel using these tables, they have to be computed upfront, which is usually done on the CPU. As the table creation account to a full symbolic subdivision, the precomputation time hurts the overall performance of table based approaches.

In this thesis the process of mesh subdivision is recast into linear algebra and implemented using parallel kernels optimized for subdivision. Instead of using complex and hard to maintain data structures we represent meshes as sparse matrices, called mesh matrices $\mathcal{M}$. This enables for expressive formulations of subdivision algorithms in terms of linear algebra and opens up a new toolset to tackle the problem. Operations on the mesh can be performed by sparse matrix-vector (SpMV) and sparse matrix-matrix (SpGEMM) multiplication. Sparse linear algebra implementations are available for the GPU and faster approaches are continuously published. Therefore, linear algebra based subdivision lends itself for parallel implementation on modern graphics hardware and benefits directly from the ongoing improvements of the underlying algebraic routines. The used mesh representation also enables for new optimization techniques, like matrix reordering, which can improve the performance of mesh operations. As a proof of concept, we describe and implement $\sqrt{3}$, Loop and the widely used Catmull-Clark subdivision schemes in our linear algebra framework. We improve upon these results by detecting and utilizing patterns in the operations on $\mathcal{M}$ and achieve a performance up to three orders of magnitude compared to the current industry standard.

This thesis is structured as follows: Chapter 2 discusses existing approaches to parallel subdivision and compares them to our linear algebra formalizations

presented in this thesis. Chapter 3 describes the mesh matrix and the operations used for computation on $\mathcal{M}$. The traditional formulations of $\sqrt{3}$, Loop and Catmull-Clark subdivision, as well as our linear algebra approach are discussed in detail in Chapter 4, and we show that it is capable of handling feature adaptive subdivision. In Chapter 5, the linear algebra operations used to perform the different subdivision schemes are analyzed and optimized versions are presented. To evaluate performance we compared our implementations of the three aforementioned schemes to OpenMesh and the current industry standard, OpenSubdiv, and discuss the results and the effect of matrix reordering in Chapter 6. Finally, the thesis is concluded in Chapter 7.

# 2 Related Work

The fact that subdivision algorithms have been a research topic for more than four decades, and the demand for subdivision in real-time applications [Bra+16], point towards the necessity of fast subdivision implementations. Subdivided meshes are used in different fields, from character animation in feature film production [DKT98] to primitive creation interactive [Zho+09] and real-time rendering [TPO10].

**Mesh representations** on the CPU are often derivatives of the winged-edge mesh representations [Bau72] which already dates back more than 40 years. As the name suggests, these are edge-centric representations where each edge points to its first and second vertex, its left and right face and to the next and previous edge for both traversal directions. Different flavors of the winged-edge representation evolved over time, for example the widely used half-edge data structure [CKS98], which mainly differ in which information is stored and how. Access to entities in the vicinity of a primitive, which is a commonly required operation in subdivision algorithms, requires traversal of the linked list of edges. Parallel traversal of faces, for example, suffers from the different length of the traversal path as face sizes vary and the traversal operation takes as long as it takes to traverse the highest order face. The same is true for traversal of edges incident to a certain vertex. Topological updates are cumbersome, because a considerable amount of pointer or index update operations are required to ensure consistency of the mesh and updating neighboring primitives in parallel requires synchronization between threads which hurts performance. A more suitable mesh representation would be favorable. Numerical operations and optimizations are often performed using matrix representations and linear algebra operations, while the mesh is described by a separate structure. Zayer et al. introduced the mesh matrix, a sparse matrix representation for unstructured grids, which enables numerical and geometric operations to be performed directly on the

mesh representation [ZSS17]. This thesis explores the use of such representations across different subdivision schemes and is concerned with optimizing the used linear algebra operations for that purpose.

**Mesh subdivision** is an active research topic and a large variety of different approaches exists. Shiue et al. divided the mesh into fragments which can be subdivided independently [SJP05]. This reduces inter thread communication but introduces redundant computations and data. To assure certain properties on the mesh, one step of subdivision might have to be done on the CPU as a preprocessing step before the fragment meshes can be extracted. The fragments are then subdivided in shaders using precomputed tables. Subdivision or stencil tables were already used in CPU implementations [BS02]. They encode for each vertex in the refined mesh the indices of primitives in the control mesh which contribute in its calculation. Stam proposed a method to evaluate Catmull-Clark subdivision surface directly without explicit subdivision [Sta98]. The approach requires precomputation of the eigenstructures of the subdivision matrix depending on the vertex valence. The patches are then transformed into eigenspace and can be evaluated at arbitrary parameter values by scaling the contribution of the splines by the eigenvalues. Stam's approach requires the extraordinary, non-valence four vertices to be isolated, such that each patch contains at most one irregular vertex. To enforce this property, explicit subdivision may be required as a precomputation step. Due to the computational cost of these exact methods with explicit subdivision, approximation schemes were introduced. Peters proposed an algorithm that transforms the quadrilaterals of a mesh into bicubic Nurbs patches, one for each face [Pet00]. While the resulting surface is at least tangent continuous everywhere, the algorithm imposes some restricting requirements on the mesh. It has to be a quad-only mesh and the extraordinary vertices have to be isolated, which requires the mesh to be subdivided at least once. The approach of Loop et al. overcomes this restrictions [LS08]. They approximate the Catmull-Clark subdivision surface using bicubic patches, which in regular regions of the mesh leads to the exact same results as the explicit subdivision. The surface regions around irregular faces do not inherit the tangent continuity from the exact computations and are therefore only position continuous. The discontinuity in the tangents may lead to shading artifacts. To get a visually smooth result they use tangent patches which produce smooth normals everywhere on the surface. This of

course requires additional computations. While approximations are usually fast to evaluate, some favorable properties of the subdivision schemes might get lost and additional techniques have to be applied to compensate for that. Therefore, the idea emerged to treat regular and irregular regions differently. Nießner et al. proposed a hybrid approach where regular faces are refined using hardware tessellation and the region around irregular vertices are subdivided recursively in compute shaders using a table based approach [Nie+12]. In feature adaptive subdivision, the tables have to be precomputed on the CPU for each mesh up to a predefined maximum subdivision level. While the subdivision tables are independent of vertex data, which enables this approach to be used for animated meshes, changes to the topology require the tables to be recomputed which makes it infeasible for some applications. Also, the preproccessing step basically requires a full subdivision up to the maximum level, which hurts the overall performance. Several extensions to the standard subdivision algorithms have been made, such as boundary handling [Nas87], dynamic levels for irregular regions [Sch+15] and displacement mapping [Coo84; NL13].

**Matrix reordering** which increases quality of the sparsity pattern is commonly used to increase performance of linear algebra operations. As the mesh matrix is representing a 3D model where indices (of faces or vertices) can be permuted without having an influence on the shape, it is possible to swap columns or rows in the mesh matrix. While reordering the matrix does not change the geometry, it can have an impact on the performance of the operations applied, because memory access patterns change. In case of subdivision it is favorable to have the non-zero entries close to the diagonal. That means that faces that are close to each other in memory use vertices that are topologically close. Mesh subdivision requires access to neighboring primitives. If these primitives have a good memory layout, the access pattern becomes more local which in general increases performance. The problem of matrix reordering has been researched for many decades. One of the most prominent reordering algorithms, which is known to produce low profile results for relatively low cost, is the Cuthill-McKee [CM69] algorithm. It reorders nodes locally based on their valence and globally based on adjacency. After choosing a starting node, which is the first entry in the ordering, the set of adjacent vertices is ordered by their valencies and appended to the global ordering. The vertex to continue with, is the next vertex in the global ordering. The reverse Cuthill-McKee, introduced in [Geo71] is the same

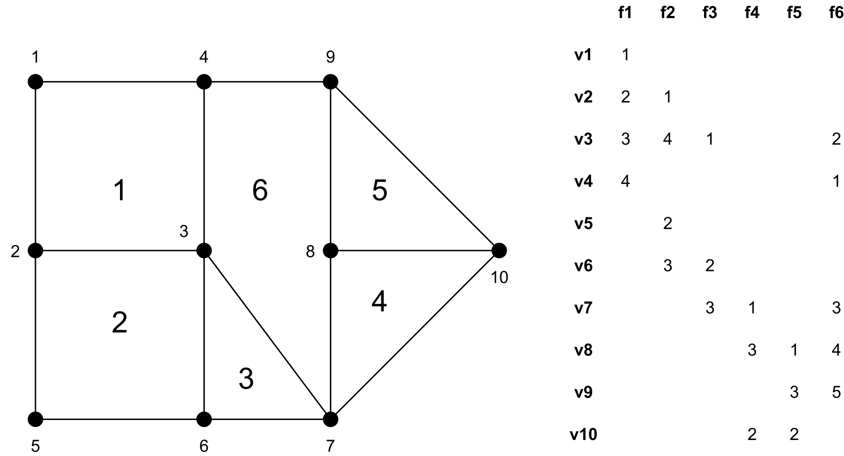
algorithm with reversed index order, which, while having the same bandwidth, usually leads to a better profile [GL81]. A different approach that reduces the envelope of a matrix is the spectral reordering[BPS93]. Instead of using the adjacency matrix, the uniform Laplacian is calculated and a second eigenvector has to be determined. This vector is sorted in monotonically increasing and decreasing order to get two permutations. The permutation that leads to the smaller envelope is chosen.

**The linear algebra approach** proposed in this thesis uses a sparse matrix as mesh representation instead of traditional data structures to eliminate related shortcomings in parallel implementations. It does not suffer from any of the disadvantages inherent to existing approaches. Our approach does not involve any preprocessing as we do not have to enforce special properties on the mesh and we do not need to create auxiliary structures such as subdivision tables. The proposed linear algebra formalizations are easy to understand and modify without any knowledge about underlying data structures or low level optimizations. As the approach is based on linear algebra expressions with only minor changes, implementations are possible on nearly any computing platform. It will be demonstrated that irregular regions in the mesh can be easily identified, extracted and subdivided using propagation in the mesh, which makes the approach applicable in a feature-adaptive setting.

# 3 The Mesh Matrix

The half-edge data structure [CKS98] has become the default choice for representing surface meshes in many computer graphics applications. It is capable of efficiently answering the most common adjacency queries. Depending on the specific implementation the memory requirements might be high. In applications where mesh topology changes frequently special attention has to be paid to keeping track of pointers to keep the data structure consistent. Face and vertex addition or removal requires an update of surrounding entities. Handling this kind of tasks in an efficient way on the GPU is difficult. With GPU implementations becoming available for more algorithms it is necessary to explore different mesh representations which are better suited to be processed in parallel. A mesh data structure that can be processed on the GPU in an efficient way was introduced by Zayer et al. [ZSS17]. The mesh matrix $\mathcal{M}$ represents unstructured grids as sparse matrices. This enables concise algorithm descriptions and efficient implementations in terms of linear algebra. Highly optimized GPU sparse linear algebra libraries are available [NVI17] and their performance improves constantly due to active research in this field. Zayer et al. introduced action maps, which can capture interactions between entities in a mesh during sparse matrix-vector (SpMV) and sparse general matrix-matrix multiplication (SpGEMM), and avoid explicit creation of intermediate results to keep a low memory footprint.

**Notation:** For the mesh matrix, the calligraphic letter $\mathcal{M}$ will be used throughout this thesis. All other matrices are named by normal upper case letters. $M(i,j)$ is the $i$-th entry in the $j$-th row of matrix $M$. $M(i,*)$ and $M(*,i)$ denote the $i$-th row and column respectively. Bold lower case letters $\mathbf{v}$ signify vectors and normal lower case letters $s$ are scalars. The number of faces and vertices in a mesh are denoted as $n_f$ and $n_v$. The number of non-zero entries in a sparse matrix is $z$.



| | f1 | f2 | f3 | f4 | f5 | f6 |
|---|---|---|---|---|---|---|
| v1 | 1 | | | | | |
| v2 | 2 | 1 | | | | |
| v3 | 3 | 4 | 1 | | | 2 |
| v4 | 4 | | | | | 1 |
| v5 | | 2 | | | | |
| v6 | | 3 | 2 | | | |
| v7 | | | 3 | 1 | | 3 |
| v8 | | | | 3 | 1 | 4 |
| v9 | | | | | 3 | 5 |
| v10 | | | | 2 | 2 | |

Figure 3.1: A small mesh consisting of six faces and ten vertices and its mesh matrix. The matrix has one column per face and one row per vertex, with the values capturing the cyclic order of vertices in each face.

## 3.1 Mesh Matrix Basics

The mesh matrix $\mathcal{M}$ is a sparse matrix, representing a surface mesh. It consists of one column per face and one row per vertex. The value stored at a specified entry $M(i, j)$ reflects the position of vertex $v_i$ in the cyclic order of face $f_j$. A small example is illustrated in Figure 3.1.

Different data structures for sparse matrices exist. One for our purpose particularly useful representations is the compressed sparse column (CSC) format, which requires three arrays. The first and second one hold values and row indices of non-zero entries respectively. The column pointer is the third array, which is needed to define a sparse matrix in the CSC format. It has one more entry than the matrix has column. Each entry is an index to the start of a column in the first two arrays. The last entry in the column pointer holds the number of non-zero entries in the matrix and is at the same time a pointer to the end of the last column. While not necessary in general, some linear algebra libraries require the row indices to be sorted in ascending order in each column, as this enables for more efficient implementation of some matrix operations. The CSC

| column pointer | 0 | 4 | 8 | 11 | 14 | 17 | 22 |
|---|---|---|---|---|---|---|---|

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 10 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| row indices | 1 | 2 | 3 | 4 | 2 | 3 | 5 | 6 | 3 | 6 | 7 | 7 | 8 | 10 | 8 | 9 | 10 | 3 | 4 | 7 | 8 | 9 |
| values | 1 | 2 | 3 | 4 | 1 | 4 | 2 | 3 | 1 | 2 | 3 | 1 | 3 | 2 | 1 | 3 | 2 | 2 | 1 | 3 | 4 | 5 |

Figure 3.2: The compressed sparse column representation of the example mesh given in Figure 3.1.

representation of the example mesh from Figure 3.1 is depicted in Figure 3.2.

A very similar and wide spread sparse matrix format is the compressed sparse row (CSR) format, which instead of column pointers and row indices, has row pointers and column indices. It is worth noting that matrix $A$ in CSC format, interpreted as CSR matrix is its transpose $A^T$, because the roles of columns and rows are exchanged. While CSR provides fast access to the rows of a matrix, CSC supports efficient access to its columns. In context of the mesh matrix, this implies fast parallel access to the individual faces and their vertices. The size of $\mathcal{M}$ in CSC format is dependent on the number of faces in the mesh and their order. Assuming $s_d$ bytes per value and $s_i$ bytes per index in the row indices and column pointer array, the size of the mesh matrix representing a mesh with $n_f$ faces, where face $f_i$ has order $c_i$ is equal to

$$\begin{aligned} size\,(\mathcal{M}) &= (s_i + s_d) \sum_{i=1}^{n_f} c_i + s_i(n_f + 1) \\ &= (s_i + s_d) z + s_i(n_f + 1). \end{aligned} \tag{3.1}$$

## 3.2 Operations on the Mesh Matrix

Computations using the mesh matrix can be expressed as basic operations in sparse linear algebra, namely sparse matrix-vector and sparse matrix-matrix multiplication. Fast parallel SpMV implementations are available and the performance is continuously improved by new approaches [SZS17; Der+17]. Performing parallel SpGEMM efficiently is a difficult task that is topic of many recent publications [LV14]. For general information about sparse multiplication

algorithms the reader is referred to the book by Davis [Dav06], which explains the fundamentals of SpMV and SpGEMM in detail.

For operations and computations on the mesh matrix, SpMV and SpGEMM are augmented by action maps [ZSS17], which enable to capture and modify interactions between colliding elements during the multiplication.

**Mapped SpMV** In SpMV, a sparse matrix $M$ is multiplied with a dense vector $\mathbf{v}$. This operation can be done in an efficient way in parallel, regardless of whether to calculate $M\mathbf{v}$ or $M^T\mathbf{v}$, because the matrix transpose does not have to be done explicitly. Instead, the transposed result can be regarded as $\mathbf{v}^T M$. In mapped SpMV, an action map $Q$ is part of the multiplication. Before calculating $M(i,j) \cdot v(j)$, the value $M(i,j)$ is used as an index into the map to get $m = Q(M(i,j))$. Subsequently, $m$ is used as it was the value of $M(i,j)$ to calculate $m \cdot v(j)$. The pseudo code for mapped SpMV is given in Algorithm 1.

**Algorithm 1:** This code illustrates mapped sparse matrix - dense vector multiplication. It calculates $\mathbf{y} = \underset{Q}{A}\mathbf{x} + \mathbf{y}$, where $\mathbf{x}$ and $\mathbf{y}$ are dense vectors, $A$ is a sparse matrix in CSC format and $Q$ is an array used as the map to substitute the matrix values.

```
Input: A, x, Q
Output: y
for c ← 0; c < A.ncols; c ← c + 1 do
    for r ← A.cptr[c]; r < A.cptr[c + 1]; r ← r + 1 do
        row ← A.rids[r];
        val ← A.vals[r];
        mval = Q[val];
        y[row] ← y[row] + mval * x[row];
    end
end
```

While the concept is simple, it enables to use SpMV for computations, which might otherwise require iterating over a subset of vertices or faces. Without loss of generality the task of interpolating vertex normals to triangle normals is given as an example:

$$\mathbf{f_n} = \underset{(1,2,3)\to\frac{1}{3}}{\mathcal{M}^T \mathbf{v_n}} \tag{3.2}$$

Each row in $\mathcal{M}^T$ corresponds to a single triangle and has non-zero entries in the columns corresponding to the triangle's vertices. Therefore, $\mathbf{f_n}(i)$ holds the normal for triangle $i$ calculated by averaging the normals of the triangle's vertices.

Instead of $M(i, j)$, the indices $i$ or $j$ can be used as an index into a map. To demonstrate the usage, the task of averaging face normals to get vertex normals is carried out by

$$\mathbf{v_n} = \underset{i\to\frac{1}{n_i}}{\mathcal{M}\mathbf{f_n}} \quad , \tag{3.3}$$

where $\mathbf{v_n}$ and $\mathbf{f_n}$ are the vectors of vertex and face normals respectively and $n_i$ is the order of the $i$-th vertex. Each row in $\mathcal{M}$ describes the affiliation of a single vertex to the faces of the mesh. It has a non-zero entry in the columns of adjacent faces. The index $i$ of a vertex is mapped to the reciprocal of its order, to average the result. Of course this could also be carried out with a constant map to one and a subsequent element-wise division by the vertex orders $\mathbf{n}$. The purpose is to demonstrate different map kinds.

**Mapped SpGEMM** SpGEMM multiplies two sparse matrices. The general approach is to do the multiplication in two passes. In a first symbolic pass, the required memory is calculated and allocated, by determining the non-zero elements in the result. In the second pass the actual multiplication is performed and the result is generated. For a detailed description the reader is again referred to Davis [Dav06]. Considering $C = AB$, the entry $C(i, j)$ in the result is calculated by multiplying row $A(i, *)$ with column $B(*, j)$. Therefore, $C(i, j)$ can only be non-zero if there is a collision between entries in the two vectors. Assuming $A(i, k)$ and $B(k, j)$ are non-zero, they will cause a collision. In the SpGEMM the two values would be multiplied and the next pair of values would be considered. In mapped SpGEMM the multiplication of the colliding values is not performed. Instead, $A(i, k)$ and $B(k, j)$ are used as row and column index

into the map $Q$, which can be thought of as a small third matrix. The result of the look-up is then used as result of the collision. The pseudo code for mapped SpGEMM is given in Algorithm 2.

**Algorithm 2:** The given pseudo code illustrates mapped sparse matrix - matrix multiplication. It calculates $C = A \underset{Q}{B}$, where $A$, $B$ and $C$ are sparse matrix in CSC format and $Q$ is a dense matrix representing the map.

```
Input: A, B, Q
Output: C
for colB ← 0; colB<B.ncols; colB ← colB+1 do
    for rB ← B.cptr[colB]; rB<B.cptr[colB+1]; rB ← rB+1 do
        rowBcolA ← B.rids[rB];
        valB ← B.vals[rB];
        for rA←A.cptr[rowBcolA]; rA<A.cptr[rowBcolA+1]; rA←rA+1 do
            rowA ← A.rids[rA];
            valA ← A.vals[rA];
            mval ← Q[valA][valB];
            C.add(rowA, colB, mval);
        end
    end
end
```

A special kind of map are the circulant matrices

$$Q_n = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \ddots & 0 & 0 \\ \vdots & \vdots & \vdots & 0 & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \in \mathbb{Z}^{i \times i} \tag{3.4}$$

which can be used to capture the cyclic order of vertices in each face [ZSS17] and can therefore be used to apply operations only to vertices that are in a special relation to each other, as for example to vertices that are connected by an edge. These matrices and their powers do not have to be constructed explicitly as they are described by

$$Q_n^p(i,j) = \begin{cases} 1 & \iff \quad j = ((i+p-1) \mod n) + 1 \\ 0 & else \end{cases} \tag{3.5}$$

and therefore the map values can be calculated on demand. To give a short example, the task of calculating the vertex-vertex adjacency matrix $S_v$ in a triangular mesh is solved using mapped SpGEMM [ZSS17]. The map is chosen to be

$$Q_3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \end{array} \tag{3.6}$$

which is called the circulant matrix of size three. The matrix $S_v$ can then be calculated using the mapped SpGEMM

$$S_v = \mathcal{M} \underset{Q_3}{\mathcal{M}}^T \tag{3.7}$$

The entry $S_v(i,j)$ is calculated by multiplying the two vectors $\mathcal{M}(i,*)$ and $\mathcal{M}(*,j)$, each representing a vertex. Each non-zero entry in these vectors indicates that the vertex is part of the face corresponding to the entry. Therefore, a collision between $\mathcal{M}(i,k)$ and $\mathcal{M}(k,j)$ implies, that vertex $i$ and vertex $j$ are both part of face $k$. The values $\mathcal{M}(i,k)$ and $\mathcal{M}(k,j)$ are the positions of the vertices in the cyclic order of face $k$. The collision between them is mapped to one, if vertex $i$ is connected to vertex $j$ by a directed edge in face $k$ and to zero otherwise. The circulant matrices $Q$ can therefore be seen as adjacency matrices, capturing the connectivity of vertices inside a face. In general the maps are not restricted to existing connectivity. As an example, the map $Q_d$ can be constructed which captures the relation between any two diagonal vertices in a quadrilateral:

$$Q_d = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array} \tag{3.8}$$

Action maps in SpMV can be utilized to avoid constructing intermediate matrices, such as the binary mesh matrix, and help keep the memory footprint small. In SpGEMM action maps capture relations between vertices in the mesh's faces, and can therefore be used to apply operations only to vertices in a certain defined adjacency relation.

# 4 Subdivision in the Language of Linear Algebra

In this chapter we will discuss, how different subdivision algorithms can be expressed in terms of linear algebra. The $\sqrt{3}$-subdivision [Kob00] will be treated first, where we show that a slightly changed view on a refinement scheme can ease parallelization and how linear algebra can be used to perform mesh subdivision algorithms. Subsequently, we will discuss the Loop subdivision scheme [Loo87], to show that the same principles are applicable in different schemes, and lead to concise formulations that can be efficiently parallelized. The latter two schemes are used to subdivide triangular meshes. To show that our approach generalizes to meshes with mixed face orders, we apply the idea of linear algebra subdivision to the Catmull-Clark algorithm [CC78]. As this scheme is widely used, we demonstrate on the example of Catmull-Clark how our approach can be used in a feature-adaptive setting and how reordering of input meshes can improve subdivision performance.

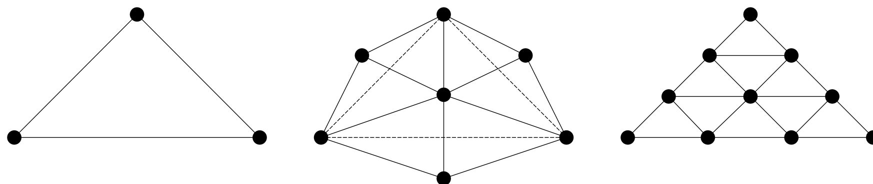


Figure 4.1: Schematic figure of two iterations of topology refinement of a triangle using the $\sqrt{3}$ scheme. After every second step of $\sqrt{3}$-subdivision every original triangle is split into nine descendants.

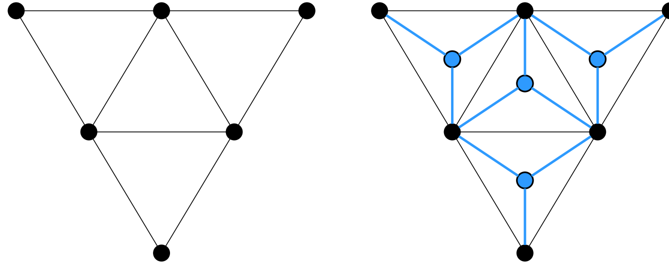


Figure 4.2: A new vertex has to be added to the barycenter of each triangle.

## 4.1 $\sqrt{3}$-Subdivision

The $\sqrt{3}$-subdivision was introduced by Leif Kobbelt [Kob00]. The algorithm is designed to subdivide triangular meshes by adding one point and three edges to each triangle and performing an edge-flip on control mesh edges afterwards. The number of faces in the mesh grows by a factor of three in each iteration. Two iterations of $\sqrt{3}$ subdivision applied to a triangle are illustrated in Figure 4.1.

### 4.1.1 Traditional Formulation

The $\sqrt{3}$-subdivision requires three basic steps - adding new vertices, updating old vertices and performing an edge-flip on the original edges.

**Adding new vertices:** In the first step a new vertex is inserted on each face $i$. The new vertex position is

$$b_i = \frac{1}{3} \cdot (p_k + p_l + p_m) \tag{4.1}$$

which coincides with the triangle's barycenter. Each new vertex is then connected to the vertices of the triangle it lives on. This step is shown in Figure 4.2.

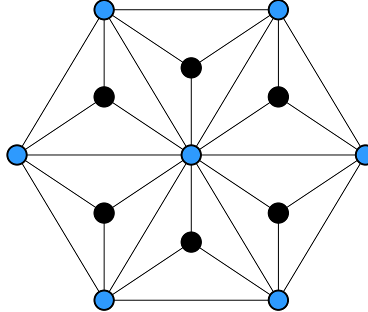


Figure 4.3: The original vertex in the center (blue) is updated using a linear combination of its neighbors in the control mesh (blue).

**Updating old vertices:** The second step smooths the mesh by updating the positions of the original vertices. Each updated vertex position is a linear combination of its old position and the average of the vertices in its 1-ring neighborhood in the control mesh. This can be seen in Figure 4.3.

The update equation is

$$\tilde{p}_i = (1 - \alpha_i)p_i + \alpha_i \frac{1}{n_i} \sum_{j=1}^{n_i} p_j \tag{4.2}$$

where $n_i$ is the vertex's order, $p_j$ is the position of the $j$-th vertex adjacent the $i$-th vertex and

$$\alpha_i = \frac{4 - 2\cos(\frac{2\pi}{n_i})}{9} \tag{4.3}$$

is a valence-dependent term which can be derived from the eigenstructure of the local subdivision matrix [Kob00].

**Edge-flip:** To complete one iteration of $\sqrt{3}$-subdivision, an edge-flip is performed on the original edges of the control mesh such that each new vertex is connect to the barycenters of the three neighboring triangles. The concluding edge-flip operation is shown in Figure 4.4.

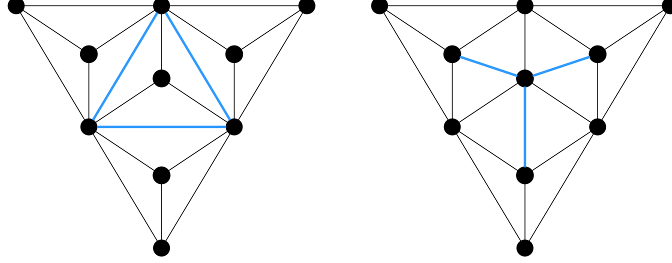


Figure 4.4: To conclude one $\sqrt{3}$ iteration, the old edges of the triangle are flipped, such that the newly added barycenters of neighboring triangles are connected.

**Properties:** The local subdivision matrix $S_i$ of size $n_i + 1 \times n_i + 1$ can be used to analyze convergence of the algorithm and smoothness of the limit surface. This matrix describes the subdivision of a subset $\mathbf{v}$ of the mesh $M$. In case of the $\sqrt{3}$ subdivision this subset is the 1-ring neighborhood of a vertex. $S$ maps the patch $\mathbf{v}$ of the control mesh $M^k$ to $\tilde{\mathbf{v}}$ in the subdivided mesh $M^{k+1}$ [Kob00]. Equations 4.1, 4.2 and 4.3 can be combined in the expression

$$\tilde{\mathbf{v}}_i = S_i \mathbf{v}_i. \tag{4.4}$$

with

$$S_i = \frac{1}{3} \begin{bmatrix} 3(1-\alpha_i) & 3\frac{\alpha_i}{n_i} & \cdots & \cdots & \cdots & 3\frac{\alpha_i}{n_i} \\ 1 & 1 & 1 & 0 & \cdots & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ \vdots & \vdots & & \ddots & \ddots & 0 \\ 1 & 0 & 0 & \cdots & 1 & 1 \\ 1 & 1 & 0 & \cdots & 0 & 1 \end{bmatrix}. \tag{4.5}$$

The limit surface of $\sqrt{3}$ is curvature continuous everywhere except for points that do not have a valence of six. At these irregular points it is tangent continuous [Kob00].

### 4.1.2 Linear Algebra Formulation

The traditional $\sqrt{3}$-subdivision formulation, as described in Section 4.1.1, does not lend itself naturally to be expressed in terms of linear algebra. The algorithm

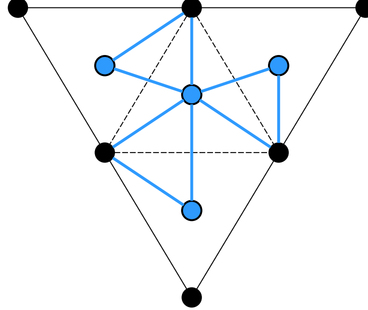


Figure 4.5: Each triangle in the control mesh contributes three new triangles to the next subdivision level.

can be formulated in an easier, more face-centric way [Zay08]. In the new formulation, each triangle contributes three new triangles to the refined mesh as shown in Figure 4.5.

Each of the three new faces consists of the barycenter, one of the triangle's vertices and the barycenter of a neighboring triangle. With this novel view on the $\sqrt{3}$ scheme, the required steps can be reformulated as linear algebra expressions easily.

**Adjacency information:** Clearly, for each half-edge in the mesh, the containing face needs to be known, to be able to determine neighboring faces for the topology refinement. We can extract this information from the mesh matrix by calculating $F$ using the mapped SpGEMM

$$F = \underset{\{Q_3\}[\gamma]}{\mathcal{M}\mathcal{M}^T} \tag{4.6}$$

with the function

$$\gamma(i,j,k) = \begin{cases} k & if \quad Q(i,j) = 1 \\ 0 & else \end{cases} . \tag{4.7}$$

Every time a collision between element $\mathcal{M}(i,k)$ and $\mathcal{M}^T(k,j)$ occurs, the attached function $\gamma$ is called, which performs the map lookup and provides an output. If the vertices described by $\mathcal{M}(i,*)$ and $\mathcal{M}^T(*,j)$ are connected by a

half-edge within face $k$, $Q_3$ will flag a non-zero, and $\gamma$ returns $k$, the index of the face, as the collision's result. Therefore, $F(i,j) = k$ and $k \neq 0$, if vertices $i$ and $j$ are connected by a half-edge in face $k$.

**Adding new vertices:** One new vertex is placed on each triangle's barycenter, which can be expressed as an SpMV with the mesh matrix

$$\mathbf{b} = \underset{(1,2,3)\to\frac{1}{3}}{\mathcal{M}^T\mathbf{p}} \tag{4.8}$$

where $\mathbf{b}$ holds the barycenter for each triangle and $\mathbf{p}$ is the vector of vertex positions.

**Updating old vertices:** The old vertex positions of the control mesh have to be updated. The update is a linear combination of vertex positions in its one-ring neighborhood. The first term in Equation 4.2, which depends on the old position of the vertex can be computed in parallel in the conventional way. The second term calculates a weighted sum of the 1-ring neighborhood and can be computed using

$$\underset{val_i\to\frac{\alpha_i}{n_i}}{F\mathbf{p}} \quad , \tag{4.9}$$

because $F$ has the same sparsity pattern as the vertex-vertex adjacency matrix, only missing the diagonal.

**Creating the new triangulation:** For each face in the control mesh three new triangles are created. Each of them is spanned by one original vertex of the parent, the new vertex in the barycenter and the new point on a neighboring face. To build the mesh matrix for the refined mesh, the indices for these vertices need to be determined. The indices of vertices of the parent triangle $i$ are known as $(k,l,m)$ and the index of the point at the barycenter is $n_v + i$. Indices of neighboring triangles can simply be fetched from the matrix $F$, then indices of new vertices on neighbors can be computed the same way. Each triangle $i$

in the control mesh contributes three columns to the refined mesh matrix, as illustrated in Table 4.1.

| column $i$ | | column $n_f + i$ | | column $2n_f + i$ | |
|---|---|---|---|---|---|
| row | value | row | value | row | value |
| $k$ | 1 | $l$ | 1 | $m$ | 1 |
| $F(l,k)$ | 2 | $F(m,l)$ | 2 | $F(k,m)$ | 2 |
| $n_v + i$ | 3 | $n_v + i$ | 3 | $n_v + i$ | 3 |

Table 4.1: The rows and non-zero values in each column corresponding to the three child triangles created from parent triangle $i$.

## 4.2 Loop Subdivision

The Loop-Subdivision scheme was introduced by Charles Loop in his master thesis [Loo87]. The algorithm subdivides triangular meshes by adding an edge-point to each edge and connecting them to split the triangles of the control mesh. The number of faces after each iteration increases by a factor of four. Two steps of Loop subdivision on a triangle are illustrated in Figure 4.6

### 4.2.1 Traditional Formulation

The algorithm consists of three basic steps. A new vertex is inserted at each edge, the original vertex positions are updated and the topology is refined.

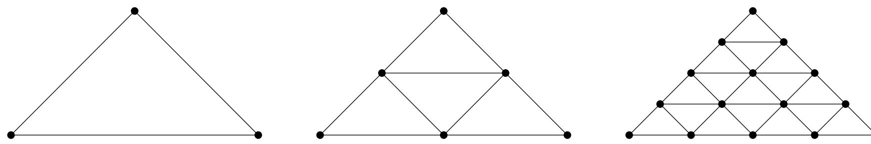


Figure 4.6: Two steps of topology refinement of a triangle (left) using the Loop scheme. After the first (middle) and second (right) step the triangle was split into four and sixteen child triangles respectively.

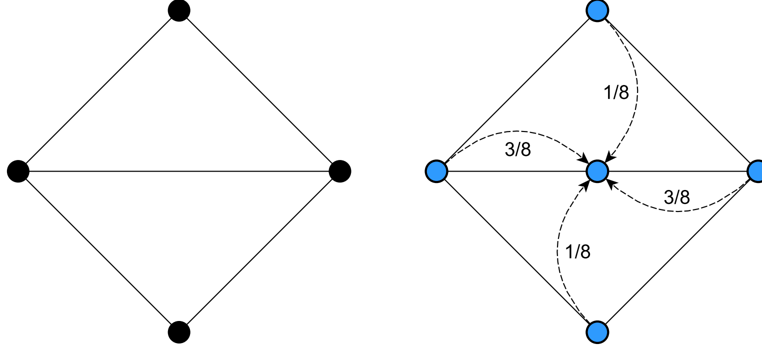



Figure 4.7: One new vertex is added per edge of the control mesh. The vertices of the two triangles adjacent to the edge contribute to the update.

**Adding new vertices:** A vertex is added on each edge at the position calculated via

$$p_{k,l} = \frac{3}{8}(p_k + p_l) + \frac{1}{8}(p_m + p_n) \tag{4.10}$$

where $p_k$ and $p_l$ are the endpoints of the edge and $p_m$ and $p_n$ are the two remaining vertices of the two triangles bordering the edge. This step is visualized in Figure 4.7.

**Updating old vertices:** To update the original vertices of the control mesh, a linear combination of positions in the neighborhood of the vertex is calculated

$$\tilde{p_i} = (1 - n\beta_i)p_i + \beta_i \sum_{j=1}^{n_i} p_j \tag{4.11}$$

where $p_i$ is the position of the vertex in the control mesh, $n_i$ is the order of the vertex and $p_j$ are the vertices in the 1-ring neighborhood. The update for a single valence six vertex is depicted in Figure 4.8.

The update of old vertices described in Equation 4.11 is governed by $\beta_i$ which is a valence dependent term defined by

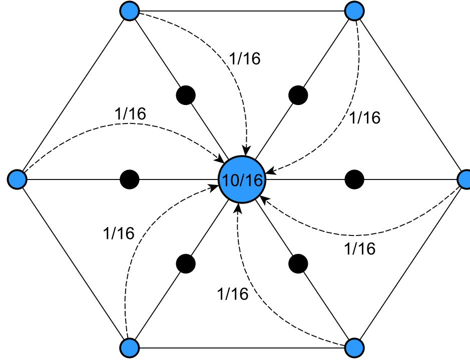



Figure 4.8: Update of an original vertex with valence six. The one-ring neighborhood and the old vertex position contribute to the update (blue). The newly added edge-points are ignored (black).

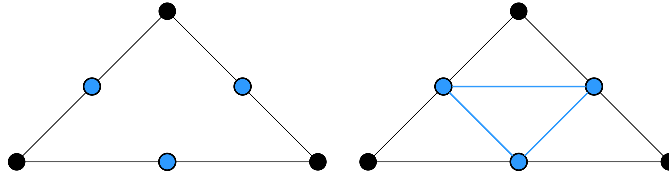


Figure 4.9: Topology update of a triangle using the loop scheme. Three edges connecting the newly inserted edge-points are inserted at the face.

$$\beta_i = \frac{1}{n_i} \cdot \left( \frac{5}{8} - \left( \frac{3}{8} + \frac{1}{4} \cdot \cos\left( \frac{2\pi}{n_i} \right) \right)^2 \right). \tag{4.12}$$

This term was derived from the local subdivision matrix, to support meshes with extraordinary, non-valence-six vertices and ensure convergence of the algorithm to a limit surface [Loo87].

**Creating the new triangulation:** Three edges are inserted at each face, connecting the edge-points and splitting the parent triangle into four new triangles. The topology update of a single triangle can be seen in Figure 4.9.

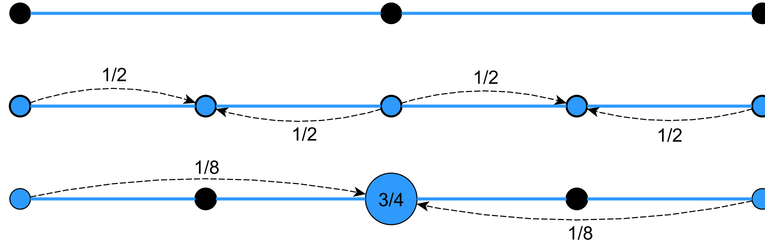



Figure 4.10: The boundary polygon is updated in two steps. First, a vertex is inserted at the midpoint of the edge. In the second step, the original boundary vertices are updated.

**Boundaries:** To be able to handle meshes with boundary, the Loop subdivision defines special rules to subdivide mesh borders.

In each step a new vertex is added to each boundary edge. Edge-point positions on border edges are calculated via:

$$p_{i,i+1} = \frac{1}{2}(p_i + p_{i+1}). \tag{4.13}$$

The rule to update a vertex position $p_i$ in the boundary polygon of a mesh is

$$\tilde{p_i} = \frac{3}{4}p_i + \frac{1}{8}(p_{i-1} + p_{i+1}). \tag{4.14}$$

**Properties** Loop showed in his thesis [Loo87] that the resulting surface is tangent continuous everywhere for correctly chosen parameters and that curvature continuity is only violated at extraordinary vertices.

### 4.2.2 Linear Algebra Formulation

**Adding new vertices:** Each edge-point's position is calculated from a linear combination of two adjacent triangles' vertices. This can be divided into two steps. The mapped SpMV in Equation 4.15 is used to calculate the three vectors $\mathbf{p}_{1,2,3}$, $\mathbf{p}_{2,3,1}$ and $\mathbf{p}_{3,1,2}$, each holding a weighted average for each triangle.

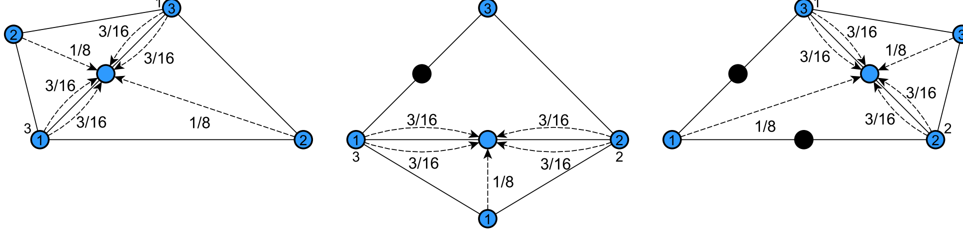



Figure 4.11: Calculation of the three edge-points of one triangle. Each edge-point is calculated from the contribution of the two adjacent triangles.

$$\mathbf{p}_{k,l,m} = \underset{(k,l,m)\to(\frac{3}{16},\frac{3}{16},\frac{1}{8})}{\mathcal{M}^T\mathbf{p}} \tag{4.15}$$

Each of the three SpMVs maps one of the three vertices of each triangle to $\frac{1}{8}$ and the remaining two to $\frac{3}{16}$. Figure 4.11 shows how the contributions of neighboring triangles are combined to form the edge-points.

In order to be able to combine the correct contributions to edge-points, the edges need to be assigned unique and consecutive indices. This can be done using a mapped SpGEMM similar to the calculation of $F$ in Equation 4.6. A matrix $E$ is created via

$$E = \underset{Q_3+Q_3^2}{\mathcal{M}\mathcal{M}^T} \quad , \tag{4.16}$$

where $Q_3$ and $Q_3^2$ capture clockwise and counter-clockwise connections of vertices within each triangle. During the multiplication, whenever a collision between two vertices $i$ and $j$ is detected in face $k$, a lookup is performed and the map returns non-zero if the two vertices are connected by an edge in the face. In this special case of triangular meshes, if two vertices share a face, which certainly causes a collision, they also share an edge. Therefore, the matrix entry $E(i, j)$ flags 2 if vertices $i$ and $j$ are connected by an internal edge and therefore share two faces, a 1 if they are connected by an edge from a boundary polygon and 0 otherwise. Unique edge indices can simply be obtained by enumerating the non-zero entries in the upper triangular part of $E$. Then, $E(\min\{i, j\}, \max\{i, j\})$

stores the index of the new vertex on the edge connecting vertices $i$ and $j$. With this information the three contributions of each triangle can be added to the corresponding edge-points.

**Updating old vertices:** Smoothing the mesh is done by updating each original vertex to a weighted average of its old position and the sum of vertex positions in its 1-ring neighborhood as described by Equation 4.11. Similar the $\sqrt{3}$, the first term is only dependent on the original position and can be calculated in parallel in a straight forward way. The second term involves the 1-ring neighborhood of each vertex, which is encoded by the non-zero entries in the corresponding row of $E$. Therefore, the second term in the update can be written as a single index mapped SpMV

$$\underset{val_i \to \beta_i}{E\mathbf{p}} \tag{4.17}$$

where each non-zero value in row $i$ of $E$ is mapped to the corresponding vertex valence dependent term.

**Creating the new triangulation:** Each triangle in the control mesh contributes four new triangles to the refined topology, hence, four columns to the refined mesh matrix. To create the topology for the four children of a parent triangle $i$ as shown in Figure 4.9, the corresponding columns are created as shown in Table 4.2. Assuming $i$ consists of vertices $k$, $l$ and $m$, the three indices of edge-points $e_{k,l}$, $e_{l,m}$ and $e_{m,k}$ can be determined using the upper triangular part of $E$. The center triangle consists of these three edge-points. Each remaining child is spanned by one of the original vertices and the two edge-points on the edges incoming to and outgoing from the original vertex.

**Boundaries:** Boundary vertices and edges can be identified using the matrix $E$. Whenever a column $i$ contains a one in any row, vertex $i$ is part of the boundary polygon and considered an external vertex. The indices of external edge-points are obtained from $E$. Boundary meshes can be handled in a build and repair fashion, meaning the individual steps to subdivide a mesh are carried out as usual and the external vertices and edge-points are identified and repaired. In the Loop scheme, the only repair that has to be performed happens after the

| column $i$ | | column $i+n_f$ | | column $i+2n_f$ | | column $i+3n_f$ | |
|---|---|---|---|---|---|---|---|
| row | value | row | value | row | value | row | value |
| $e_{k,l}$ | 1 | $k$ | 1 | $l$ | 1 | $m$ | 1 |
| $e_{l,m}$ | 2 | $e_{k,l}$ | 2 | $e_{l,m}$ | 2 | $e_{m,k}$ | 2 |
| $e_{m,k}$ | 3 | $e_{m,k}$ | 3 | $e_{k,l}$ | 3 | $e_{l,m}$ | 3 |

Table 4.2: The rows and non-zero values in each column corresponding to the four child triangles created from parent triangle $i$.

refined vertex data was calculated. Let $\mathbf{v_1}$ and $\mathbf{v_2}$ be start and end vertices of the external edges with corresponding edge-point indices $\mathbf{e}$. With that information the edge-points in the refined vertex data $\tilde{\mathbf{p}}$ can be repaired, using the original positions $\mathbf{p}$

$$\tilde{\mathbf{p}}(\mathbf{e}) = \frac{1}{2}\left(\mathbf{p}(\mathbf{v_1}) + \mathbf{p}(\mathbf{v_2})\right). \tag{4.18}$$

Similarly, position of external vertices is corrected:

$$\tilde{\mathbf{p}}(\mathbf{v_1}) = \frac{3}{4}\mathbf{p}(\mathbf{v_1}) \tag{4.19}$$

$$\tilde{\mathbf{p}}(\mathbf{v_1}) = \tilde{\mathbf{p}}(\mathbf{v_1}) + \frac{1}{8}\mathbf{p}(\mathbf{v_2}) \tag{4.20}$$

$$\tilde{\mathbf{p}}(\mathbf{v_2}) = \tilde{\mathbf{p}}(\mathbf{v_2}) + \frac{1}{8}\mathbf{p}(\mathbf{v_1}) \tag{4.21}$$

## 4.3 Catmull-Clark Subdivision

The Catmull-Clark Subdivision [CC78] is probably the most widely used subdivision scheme. A very interesting property of the algorithm is that it can operate on polygonal meshes and regardless of face orders in the input, always produces pure quadrilateral meshes. The number of children of a face after one iteration is equal to the face's order. This is demonstrated for a single triangle in Figure 4.12.

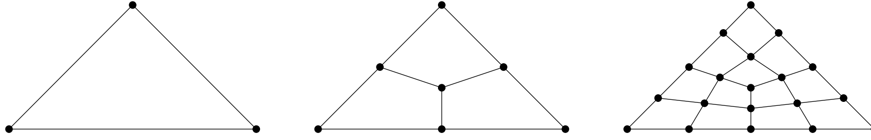


Figure 4.12: Two iterations of topology refinement of a triangle (left) using the Catmull-Clark scheme. After one iteration (middle), all faces in the mesh are quadrilaterals. Another step of Catmull-Clark has split the original triangle into twelve quads (right).

### 4.3.1 Traditional Formulation

One iteration of Catmull-Clark subdivision is done in four steps. The first two add vertices to faces and edges respectively. The third step updates the original vertices to smooth the resulting mesh. To create the refined topology, edges are added to the mesh, each connecting a face-point to a neighboring edge-point.

**Calculating face-points:** The position of the face-point $f_i$ added to a face $i$ is set to the barycenter of the polygon

$$f_i = \frac{1}{c_i} \sum_{k=1}^{c_i} p_k \tag{4.22}$$

where $c_i$ is the order of the face and $p_k$ are the face's vertices.

**Calculating edge-points:** The position of the new edge-point $e_{k,l}$ on each edge is the average of the edge's endpoints $p_k$ and $p_l$ and the face-points $f_m$ and $f_n$ on the two faces adjacent to the edge.

$$e_{k,l} = \frac{1}{4} \left( p_k + p_l + f_m + f_n \right) \tag{4.23}$$

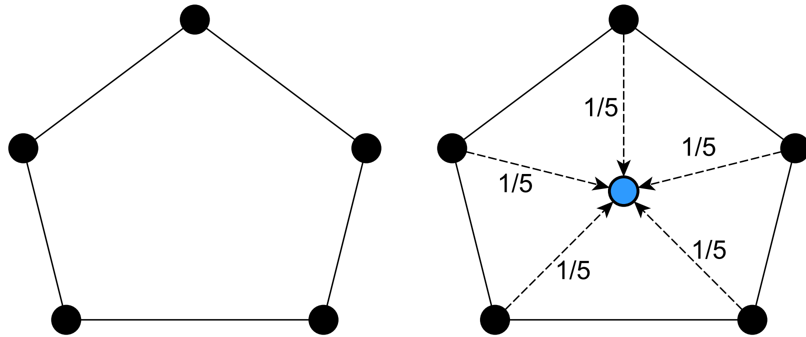



Figure 4.13: A face-point has to be added to each face. The position is equal to the face's barycenter.

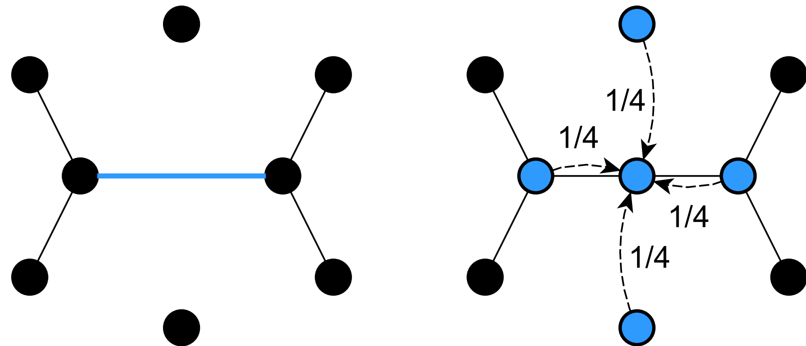



Figure 4.14: An edge-point is added to each edge. The edge-point's position is the average of its two end points and the face-points on the neighboring faces.

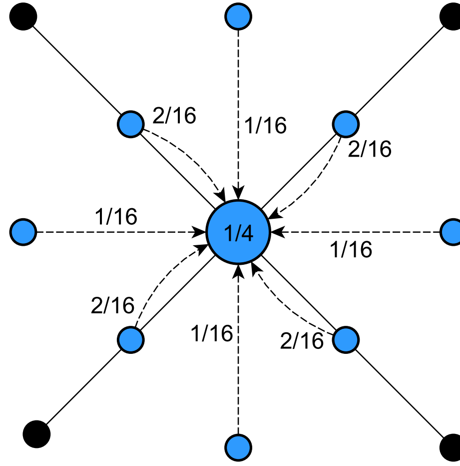



Figure 4.15: Each updated vertex position is a linear combination of the old position, the average of edge-mid points and the average of all face-points on adjacent faces.

**Updating old vertices:** To get a smooth result the original vertices of the control mesh have to be updated. Each original vertex is set to a linear combination of its old position, the edge-mid-points of all adjacent edges and the face-points added to each adjacent face in the current iteration. This is done via

$$\tilde{p_i} = \frac{1}{n_i}\left(\frac{1}{n_i}\sum_{j=1}^{n_i} f_j + \frac{2}{n_i}\sum_{j=1}^{n_i}\frac{1}{2}\left(p_i + p_j\right) + \left(n_i - 3\right)p_i\right) \tag{4.24}$$

where $n_i$ is the vertex's valence, $f_j$ are the face-points on adjacent faces and $p_j$ are the vertices sharing an edge with $p_i$.

**Connecting new vertices:** To conclude the iteration, new edges have to be added to each face. One new edge is added per vertex of the face, which connects the face-point and an edge-point on one of the face's edges. Therefore, each parent face is split into $c_i$ child quadrilaterals, each spanned by one of the original vertices $p_i$, the edge-point on the edge outgoing from $p_i$, the face-point $f_i$, and the edge-point on the edge incoming to $p_i$.

**Boundaries:** Catmull-Clark subdivision also supports meshes that have a boundary. The rule to update a vertex position $p_i$ in the boundary polygon of a mesh is

$$\tilde{p_i} = \frac{3}{4}p_i + \frac{1}{8}(p_{i-1} + p_{i+1}). \tag{4.25}$$

The positions of edge-points on the boundary are set to the edge mid-points

$$e_{i,i+1} = \frac{1}{2}(p_i + p_{i+1}). \tag{4.26}$$

**Properties:** The limit surface of the Catmull-Clark subdivision in regular regions of the mesh can be modeled as bicubic B-splines, and is therefore curvature continuous. The proof of a continuous tangent plane at extraordinary vertices, while not provided in the original paper, was given by Doo et al. [DS98].

### 4.3.2 Linear Algebra Formulation

This section introduces the linear algebra formulation for Catmull-Clark subdivision, which is the computationally most expensive, of the three schemes treated in this thesis, as it combines primitives used in both, $\sqrt{3}$ and Loop. In Catmull-Clark, after the first iteration of refinement, all meshes will only comprise quads. The number of faces in each subdivision level grows exponentially, which makes an efficient implementation for quadrilateral crucial for good performance.

**Calculating face-points:** The positions of face-points added to each polygon coincide with the faces' barycenters which can be calculated using a mapped SpMV. Each row in $\mathcal{M}^T$ represents one face in the mesh. Therefore, each non-zero value $val_i$ in row $i$ is mapped to the reciprocal of the face's order:

$$\mathbf{f} = \underset{val_i \to \frac{1}{c_i}}{\mathcal{M}^T \mathbf{p}}\,. \tag{4.27}$$

To calculate the face-points in an quad mesh, the SpMV is simply

$$\mathbf{f} = \underset{(1,2,3,4)\to\frac{1}{4}}{\mathcal{M}^T\mathbf{p}} . \tag{4.28}$$

**Calculating edge-points:** Edge-points are a linear combination of edge-endpoints and adjacent face-points. The indices of face-points on faces adjacent to each edge are known from the matrix $F$, which is calculated using Equation 4.6. Furthermore, the start and end vertex of each edge, as well as the index of its edge-point can be determined using $E$, calculated in Equation 4.16. Using these information the edge-points are calculated by combining the two face-points on the adjacent faces and the two incident vertices.

**Updating old vertices:** The update Equation 4.24 can be conveniently rewritten as

$$\tilde{p_i} = p_i\left(1 - \frac{2}{n_i}\right) + \frac{1}{n_i^2}\sum_{j=1}^{n_i} p_j + \frac{1}{n_i^2}\sum_{j=1}^{n_i} f_j \tag{4.29}$$

such that the update can be split into three summands. The first is the term which depends on the original position and can be calculated using the mapped SpMV where non-zero entries in row $i$ are mapped to a term dependent on the valence of vertex $i$

$$\mathbf{p}_1 = \underset{val_i\to(1-\frac{2}{n_i})}{\mathcal{I}\mathbf{p}} \tag{4.30}$$

with $\mathcal{I}$ being the identity matrix. The second term sums the 1-ring neighborhood of the vertex. This can again be done using $E$, as it has the same sparsity pattern as the vertex-vertex adjacency matrix with a zero diagonal,

$$\mathbf{p}_2 = \underset{val_i\to\frac{1}{n_i^2}}{E\mathbf{p}} . \tag{4.31}$$

The third and last term required in the smoothing step calculates a weighted sum of the face-points on faces adjacent to the vertex. This is accomplished using a mapped SpMV of the mesh matrix $\mathcal{M}$ with the previously calculated face-points

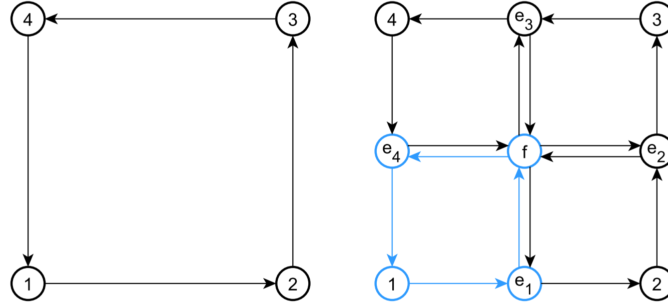



Figure 4.16: Refining the topology of a quadrilateral, resulting in four new faces. Each consists of one of the original vertices, the edge-point on the outgoing edge, the parent's face-point and the edge-point on the incoming edge.

$$\mathbf{p}_3 = \underset{val_i \rightarrow \frac{1}{n_i^2}}{\mathcal{M}\mathbf{f}} . \tag{4.32}$$

To complete the update, the components are added to get the updated vertex positions.

**Creating the new topology:** Each polygon $i$ of the control mesh contributes quads to the refined mesh, equal to its order $c_i$. Each of the new faces consists of one vertex of the parent face, the parent's face-point and two edge-points. The split of a quad is visualized in Figure 4.16.

The index of the original vertex is known and the face-point on face $i$ takes the index $n_v + i$. What remains to be determined are the indices of the two edge-points. The matrix $E$ can be used again. The entry $E(\min\{k, l\}, \max\{k, l\})$ holds the index of the edge $e_{k,l}$ connecting vertex $k$ and $l$. The Index of the corresponding edge-point is simply $n_v + n_f + e_{k,l}$.

**Boundaries:** Meshes that are not closed and therefore have a boundary are handled in the Catmull-Clark subdivision the exact same way as they are in the Loop subdivision. See Section 4.2.2 for a detailed explanation.

### 4.3.3 Feature-Adaptive Subdivision

It is well known that Catmull-Clark subdivision of regular faces can be done using bicubic patches, as they describe the behavior of the limit surface around valence four vertices. To describe the surface around irregular vertices an infinite number of patches would be required [DS98]. Bicubic patches can be evaluated using programmable shaders and hardware tessellation on modern graphics cards, which enables for faster implementations, while sacrificing some flexibility. Hardware tessellation has some limitations, for example, the tessellation factor can be at most 64, which equals 6 subdivision iterations. To improve subdivision performance, the idea emerged to subdivide regions around irregular vertices manually and handle bicubic patches for regular regions in tessellation shaders [Nie+12]. This section discusses how the irregular faces can be identified and how to determine the neighborhood that is needed to subdivide them. Using the mesh matrix, no traversal of data structures is required to find the sub-mesh that is needed for manual subdivision, but rather can be determined by propagation. We refrain from calling the subset of faces patches because the sub-mesh is subdivided just like the whole mesh would be and the algorithm is not changed in any way to work on a per patch basis. The sub-mesh generation is described below.

The irregular vertices are identified by checking the vertex orders, calculated using

$$\mathbf{n} = \underset{\forall vals \in \mathcal{M} \to 1}{\mathcal{M}\mathbf{1}}. \tag{4.33}$$

The algorithm produces two vectors which describe the desired sub-mesh. The first one is the vertex vector, which has non-zero values only at entries corresponding to vertices that are in the sub-mesh. Similarly, the face-vector has non-zero entries for faces which belong to the sub-mesh. The vertex vector $\mathbf{p_0}$ can be initialized by setting irregular vertices to one:

$$\mathbf{p_0}(i) = \begin{cases} 1 & if \quad \mathbf{n}(i) \neq 4 \\ 0 & else \end{cases} \tag{4.34}$$

The propagation consists of two steps. First, the mesh matrix is used to determine the neighboring faces $\mathbf{f_0}$

$$\mathbf{f_0} = \mathcal{M}^T \mathbf{p_0} \tag{4.35}$$

which has a non-zero entry at position $i$, if the $i$-th face contains an irregular vertex. In the second step, all the vertices that are contained in the faces tagged by $f_0$, are identified using the SpMV

$$\mathbf{p_1} = \mathcal{M} \mathbf{f_0} \tag{4.36}$$

where $\mathbf{p_1}$ tags all irregular vertices and their 1-ring neighborhood. To be more general, the two steps

$$\mathbf{f_i} = \mathcal{M}^T \mathbf{p_i} \tag{4.37}$$

$$\mathbf{p_{i+1}} = \mathcal{M} \mathbf{f_i} \tag{4.38}$$

can be repeated multiple times to get the set of faces $\mathbf{f_i}$ containing the vertices specified in $\mathbf{p_i}$ and the set of vertices $\mathbf{p_{i+1}}$ contained in the faces specified in $\mathbf{f_i}$. After $n$ steps, the vector $\mathbf{p_n}$ has non-zero entries for all the vertices contained in the $n$-ring neighborhood of vertices specified in $\mathbf{p_0}$. $\mathbf{f_{n-1}}$ contains all the faces adjacent to at least one vertex of $\mathbf{p_{n-1}}$. The sub-mesh growth is visualized in Figure 4.17.

To perform Catmull-Clark subdivision of faces in the 1-ring neighborhood of an irregular vertex, information contained in the two ring neighborhood is needed. Therefore, the irregular vertices of the control mesh have to be identified and the propagation step described above is performed two times to identify the sub-mesh needed for the adaptive subdivision. Indices of primitives in this sub-mesh are in general not continuous anymore. New unique and continuous indices for faces and vertices are generated by setting non-zero entries to one and performing a cumulative sum over the vectors $\mathbf{f_1}$ and $\mathbf{p_2}$ respectively. With this, a new mesh matrix is created, only containing relevant faces. A condensed vertex vector containing only relevant vertices is built similarly. Due to the fact

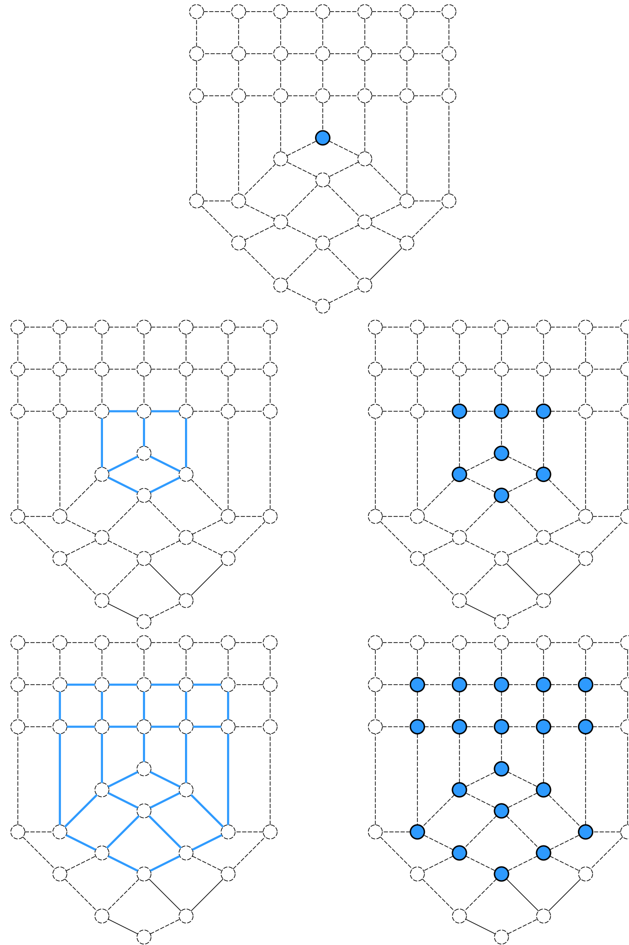


Figure 4.17: To extract the sub-mesh for an irregular vertex the vertex vector $\mathbf{p_0}$ is initialized to flag a one at the position of the irregular vertex and zero otherwise (top). $\mathbf{f_0}$ is calculated by multiplying the mesh matrix with $\mathbf{p_0}$ as stated in Equation 4.37. Only entries in the result corresponding to neighbors of the vertex in $\mathbf{p_0}$ are non-zero (middle, left). In the second part of the propagation step, $\mathbf{p_1}$ is calculated using Equation 4.38 and only entries in $\mathbf{p_1}$ corresponding to vertices that are part of faces tagged in $\mathbf{f_0}$ are non-zero (middle right). $\mathbf{f_0}$ and $\mathbf{p_1}$ represent the one ring-neighborhood of the irregular vertex. A second propagation step is shown in the bottom row.

that all faces, vertices and their corresponding new indices in the sub-mesh are known, building the sub-mesh matrix is a simple copy instruction. A sub-mesh that only consists of the $j$-neighborhood around each irregular vertex is called the $M_{j,l}$-sub-mesh if its primitives are a subset of a mesh at the $l$-th subdivision level. The feature adaptive approach works as follows

$$M \underset{s}{\rightarrow} M_1 \underset{p}{\rightarrow} M_{3,1} \underset{s}{\rightarrow} M_{6,2} \underset{p}{\rightarrow} M_{3,2} \underset{s}{\rightarrow} \cdots$$

where $s$-steps and $p$-steps stand for subdivide and propagate respectively. The control mesh $M$ is subdivided once to get the level one mesh $M_1$. Then propagation is performed to get the $M_{3,1}$ sub-mesh, which is then subdivided to $M_{6,2}$ and propagation reduces it to the $M_{3,2}$. This is repeated until the desired subdivision level is reached. There are also other options in which order subdivision and propagation are applied and which neighborhood size is chosen. The 2-ring neighborhood around irregular vertices would be sufficient to do the subdivision. In the current implementation, we decided to do one full subdivision step and perform the first propagation on level one because in the first iteration some mesh properties like boundary and irregular vertex indices can be evaluated, which are needed upfront. The 3-ring propagation was chosen to conform with the output OpenSubdiv provides. Of course it would also be possible to work with the 2-ring neighborhood during consecutive subdivision steps and only output the 3-ring neighborhood after the last step. By outputting the 3-ring neighborhood it is ensured that it contains all the information needed to build the patches for regular faces, in case the mesh is further subdivided later on. Regardless of which subdivision-propagation combination is chosen, after subdividing the region around irregular vertices, new regular faces emerged as shown in Figure 4.18. For these faces, patches can be built and evaluated using hardware tessellation.

The patches created for some of these faces are transition patches as they lie on a boundary between different subdivision levels and special care has to be taken to avoid cracks and T-vertices in the subdivided mesh. For further detail on this matter the reader is referred to Nießner et al. [Nie+12]. However, regular faces are ignored in our current implementation as the purpose is to show that the linear algebra approach can be used to find and subdivide the irregular

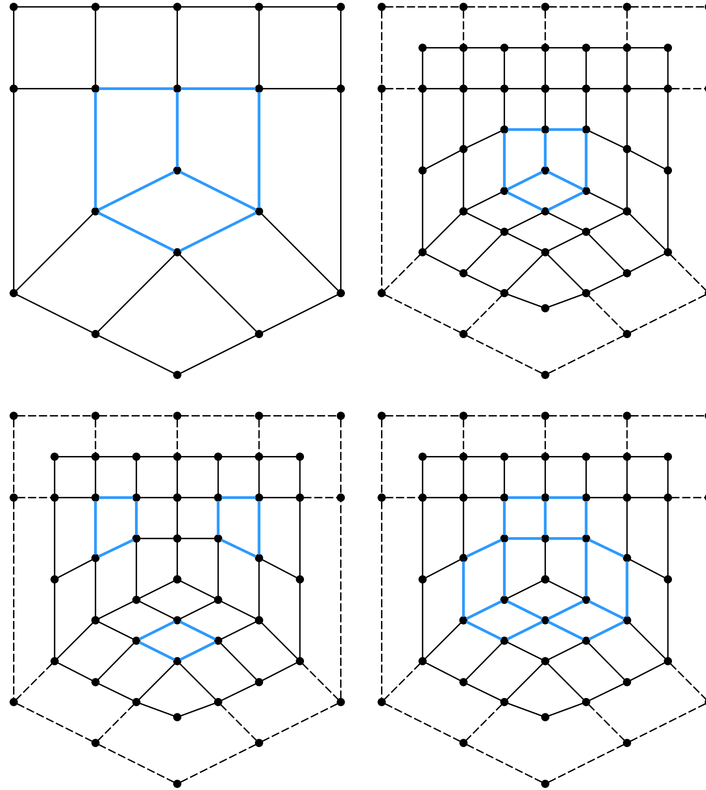


Figure 4.18: Feature adaptive Catmull-Clark subdivision illustrated on a small mesh with one irregular vertex in the center. The top left figure depicts the mesh to subdivide and the edges of irregular faces are shown in blue. After one subdivision iteration and the propagation, the result is the sub-mesh $M_{3,1}$, shown with non-dashed lines in the top right. The new irregular faces for the next iteration are shown in blue. In the bottom left, the blue edges indicate the faces which can be subdivided with bicubic patches. The blue faces in the bottom right indicate those which are subdivided using transition patches [Nie+12], as they are situated between faces of different subdivision levels.

regions and that the information to create patches for emerging regular faces is present.

### 4.3.4 Mesh reordering

Representing a mesh as a sparse matrix $\mathcal{M}$ enables to use techniques that are used to improve performance of linear algebra operations, to make operations on the mesh matrix more efficient. One interesting technique, that was used successfully to improve subdivision performance in our approach, is matrix reordering. To be able to select the best permutation among many different ones, it is required to define some quality metrics for matrices. Recall, that the mesh matrix comprises one column per face in the mesh and each column has one non-zero entry per vertex in the face. Subdivision algorithms usually require plenty of averaging over local neighborhoods. Memory access patterns matter on the CPU and are crucial on the GPU to reach good performance. Therefore, adjacent faces should be close to each other in memory. Also, the vertices of a face, as well as vertices of adjacent faces should be close to each other in memory. In short, primitives that are topologically close in the mesh should also be close in memory. Converted to the sparsity pattern of the mesh matrix, this means that all the non-zero elements should be close to the diagonal. A measure that gives information about closeness of non-zero entries to the diagonal is the matrix bandwidth. The row bandwidth $b_r$ is the maximum distance between two non-zero entries in any row. The column bandwidth $b_c$ can be defined in a similar manner. The bandwidth of the matrix is then $b = \max\{b_r, b_c\}$. For square matrices the row and column bandwidths are equal. In case of a mesh matrix with a small column bandwidth, the faces consist of vertices with indices that are close to each other, which implies that the vertices reside close in memory. Analogously, a small row bandwidth means that faces using the same vertices are close to each other in memory.

To reduce the bandwidth, a modified reverse Cuthill McKee (RCM) reordering was used [CM69; Geo71]. The original RCM works on square symmetric matrices and calculates a permutation to reduce the bandwidth. For further detail the reader is referred to original papers by Cuthill et al. and George. To reorder non-square mesh matrices, the standard RCM algorithm is applied to the Laplacian matrix of the mesh, to obtain a permutation which is applied to the rows of

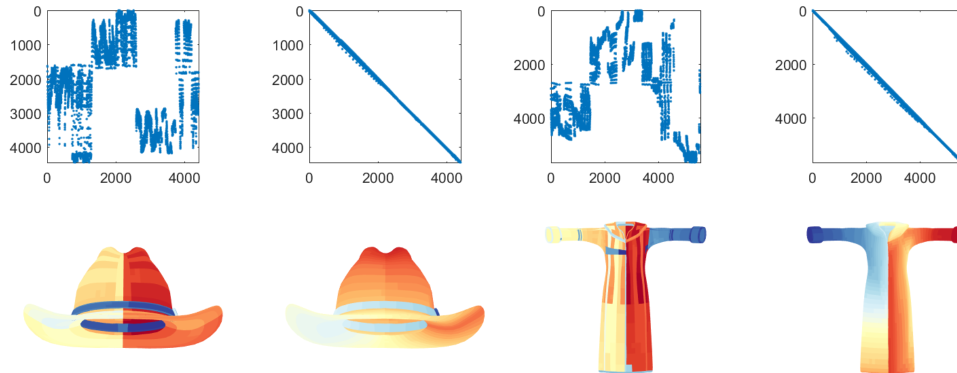



Figure 4.19: Reordering of two models using RCM on the Laplacian for row reordering and subsequently sorting the columns by the row index of their first non-zero entry. For both, the sparsity pattern of the original (left) and the reordered matrix are shown (right). Colors of the mesh indicate positions of the faces in memory.

the mesh matrix. The columns are sorted according to the minimum row index of their non-zero entries. Some results of the reordering algorithm can be seen in Figure 4.19.

When subdividing a mesh using our Catmull-Clark approach, the faces that emerge from the same parent face are placed next to each other in memory. Also, the global ordering of faces is preserved. If parent columns $i$ and $j$ fulfill $i < j$, then their children $i_k$ and $j_l$ also fulfill $i_k < j_l$. By creating the new topology in this way instead of just appending columns to the existing matrix and altering original columns, the locality of faces is preserved. The refined vertex data vector contains all updated vertex positions, followed by the face-points and the edge-points are placed last. Therefore, subdividing a reordered mesh results in a sparsity pattern that features three narrow bands, as clearly visible in Figure 4.20.

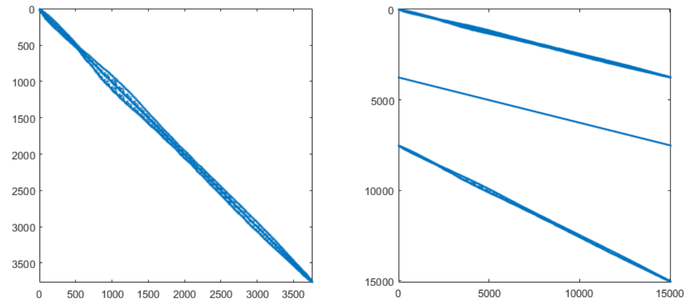



Figure 4.20: A mesh matrix after the initial reordering (left). After one step of subdivision, the clustering of original, face and edge-points is clearly visible in the structure of the matrix (right).

Due to the reordering, access to neighboring primitives in the topology is translated to more local memory accesses which results in an increase in performance. Comparisons of subdivision performance using reordered and original meshes are presented and discussed in Chapter 6.

# 5 Optimization of Mesh Matrix Operations

While the $\sqrt{3}$, Loop and Catmull-Clark subdivision schemes can be implemented using the standard linear algebra operations as described in Sections 4.1.2, 4.2.2 and 4.3.2 respectively, the operations on the mesh matrix can be optimized due to the knowledge of the underlying structures. The fact that the used SpMVs and SpGEMMs do not have to work on arbitrary matrices but rather on mesh matrices enables for specialization of the individual operations to improve the overall performance of the subdivision. This chapter will identify commonalities of the schemes in terms of the applied linear algebra operations and show how they can be optimized, considering the underlying structure and characteristics of mesh matrices.

## 5.1 Reduced Mesh Matrix

Whenever $\mathcal{M}$ represents a homogeneous mesh, such as a pure triangular or quadrilateral mesh, the column pointer can be dropped, because the face orders are consistent and the start index in the row indices for each face is known. Therefore, the correspondence of a row index to a certain face is given by the position of the index in the array. Reordering the row index-value pairs such that the values are sorted in each column also renders the value array unnecessary, because the cyclic order of vertices in a face is implicitly given by the order of their appearance in the row indices array. A mesh matrix for a homogeneous mesh that only consists of the row indices is called the reduced mesh matrix $\mathcal{M}_r$ and its size is equal to the size of a face table representing the same mesh [ZSS17].

To illustrate the concept, let the mesh matrix $\mathcal{M}$ represent a homogeneous mesh with $n_f$ faces, each having face-order $c$. $\mathcal{M}$ is in reduced form and therefore only consists of the row index array $\mathcal{M}.rid$ which comprises $n_f \cdot c$ entries. The entries in the interval $[i \cdot c, (i+1) \cdot c)$ describe the $i$-th column, which represents the face $i$. The row indices are ordered within each interval to capture the cyclic order of the vertices in the corresponding face. The information provided by value array and column pointer are therefore redundant, and can be omitted.

$\sqrt{3}$ and Loop subdivision only operate on triangle meshes and the Catmull-Clark algorithm produces meshes exclusively consisting of quadrilaterals. Therefore this optimization can be applied in all three implementations.

## 5.2 Implicit mapped SpGEMM

A mapped SpGEMM of the form

$$A = \mathcal{M} \underset{\{Q\}[\alpha]}{} \mathcal{M}^T \tag{5.1}$$

is used to calculate the matrices $E$ and $F$ containing adjacency information. Note, that in Equation 4.16 for calculating $E$, the function $\alpha$ was omitted, as it directly returns the map value. While the algorithms to perform SpGEMM are constantly improving, it is still a computationally intense task. Therefore, the goal should be to avoid explicit multiplication whenever possible. In case of the multiplication in Equation 5.1 this is possible and the three matrices can be created directly from $\mathcal{M}$. To see why this is possible it has to be understood what exactly happens during the multiplication and how the different action maps influence the result.

During the matrix-matrix multiplication $\mathcal{M}\mathcal{M}^T$, each row $r_i = \mathcal{M}(i, *)$ is multiplied with each column $c_j = \mathcal{M}^T(*, j)$. Both, $r_i$ and $c_j$ encode one vertex each and have non-zero entries at the positions corresponding to their surrounding faces. During multiplication, a collision between two entries $\mathcal{M}(i, k)$ and $\mathcal{M}^T(k, j)$ happens if both are non-zero, meaning that vertices $i$ and $j$ share a face $k$. Clearly, vertices not part of the same face will never induce a collision. An action map encodes a specific relation between vertices within a face. A collision invokes the function $\alpha$, which provides an output based on the map lookup.

A mapped SpGEMM as in Equation 5.1, would usually require multiplication of each row vector in $\mathcal{M}$ with each column of $\mathcal{M}^T$. As mentioned above, in the special case of the mesh matrix, collisions of non-zero entries can only occur if the vertices, represented by the vectors, share a face. This fact implies that we only need to check the vertices of a face against each other. The number of vertex pairs in each face, for which the function $\alpha$ needs to be executed, can be further reduced, because an invocation will only return a non-zero value, if the map lookup evaluates to non-zero. The vertices in a face for which this is the case are directly determined by the map. Action maps encode relations between positions of vertices within the cyclic order of a face. For the $i$-th vertex of a face, the function $\alpha$ only needs to be called with the $j$-th vertex if $Q(i, j) \neq 0$. Therefore, $Q(i, *)$ is an evaluation pattern for the $i$-th vertex in each face, which determines the invocations of $\alpha$.

Before the actual multiplication can be carried out in the way described above, the number of non-zero values in each column of the result needs to be determined, to be able to allocate sufficient memory for the arrays of its CSC representation. This can be done using a preceding symbolic pass, similar to general SpGEMM algorithms. In parallel for each entry in the row indices of $\mathcal{M}$, we use the map to determine the number of evaluations for the vertex by counting the non-zeros in the map row corresponding to the vertices' position in the cyclic order of the current face. The total number of invocations for each vertex is accumulated in an array and is equal to the non-zero entries in the vertex's column of the result. A simple parallel scan (cumulative sum) over that array gives the column pointer and the number of non-zero values of the resulting matrix. With that information the row index and value arrays can be allocated and subsequently filled during the evaluation pass. It is worth noting that this step can be skipped if each row of the map has the same number of non-zero entries. Then the number of evaluations and therefore the number of non-zero values of the vertex's column in the result, is independent of its position in the cyclic order of adjacent faces. If this is the case, the invocations on each vertex can be directly calculated as a multiple of the vertex order $z_r\mathbf{n}$.

## 5.3 Specialized SpMVs

Whenever certain patterns in SpMVs formulated in the higher level linear algebra description are detected, they can be transformed to specifically tailored GPU kernels. Basis for all eventual SpMV optimizations is a naïve GPU implementation [Ste+16] as shown in Algorithm 3 and Algorithm 4. Note that an explicit transpose of the matrix is not required to perform transposed matrix-vector multiplication. This is particularly important since we make extensive use of the mesh matrix and its transpose.

**Direct mapped SpMV** In the mapped SpMV for CSC matrices, multiple threads collaborate to calculate a single element of the result vector. Parallelization is done over the input elements. A thread reads a single entry of the input vector and multiplies it with the mapped non-zero elements of the corresponding column. These intermediate products are directly accumulated in the result vector using an atomic addition operation to avoid race conditions. Pseudo code for mapped SpMV is given in Algorithm 3.

**Algorithm 3:** Direct CSC matrix vector multiplication with maps. The algorithm calculates $y = Ax + y$ in parallel with a number of threads equal to the number of columns of $A$

```
Input: A, x, m
Output: y
tid ← getUniqueThreadId();
x_val ← x[tid];
for i ← A.cptr[tid]; i < A.cptr[tid + 1]; i ← i + 1 do
    rid ← A.rids[i];
    map_val ← m[A.vals[i]];
    atomicAdd(y[rid], map_val · x_val);
end
```

**Transpose mapped SpMV** In the transpose mapped SpMV for CSC matrices, a single thread is responsible for a single output element, which eliminates the need for atomic operations. Each thread iterates over the non-zero elements of its column, uses the map to substitute them and multiplies each mapped value with the corresponding vector element. This means, that in contrast do the

direct mapped SpMV, a thread has to read multiple elements from the input vector - see Algorithm 4.

**Algorithm 4:** Transpose mapped sparse CSC matrix vector multiplication with maps. The algorithm calculates $y = A^T x + y$ in parallel with a number of threads equal to the number of columns of $A$

```
Input: A, x, m
Output: y
tid ← getUniqueThreadId();
val ← 0;
for i ← A.cptr[tid]; i < A.cptr[tid + 1]; i ← i + 1 do
    rid ← A.rids[i];
    map_val ← m[A.vals[i]];
    val ← val + map_val · x[rid];
end
y[tid] ← y[tid] + val;
```

**Specializations and Memory Optimizations** Depending on the input to the high level linear algebra primitives we can detect several special properties and patterns and produce highly efficient GPU code for the individual applications. If the input matrix to an SpMV is in reduced form, every column has the same number of non-zero entries, which renders the loop over each column obsolete and it can be unrolled. Therefore, the column pointer is not required to perform the multiplication. Value arrays can also be omitted, because row indices in each column are sorted in a reduced mesh matrix to reflect the cyclic order of the face. In both SpMV versions each thread works on one column of the matrix. If the mesh matrix represents a quadrilateral mesh, as it is very common in the Catmull-Clark scheme, each column has exactly four entries. Instead of performing four individual memory accesses while reading the row indices, a single 128-bit request can be issued, reducing the number of reads by a factor of four.

Certain properties of the map might also be exploited to optimize the code. If the map is a constant function, as it is often the case if an SpMV is used to average over some neighborhood, the value of the map can simply reside in shared or constant memory, and frequent map lookups are not necessary. In the non-transposed case, maps that output the same value for each entry in a face can be handled similarly, as each thread works on a single column and only needs to perform a lookup once.

The input vector might give information about the purpose of the SpMV which enables for further optimizations. To count elements (vertices in each face, faces adjacent to each vertex) the mesh matrix is multiplied with a one vector. In this case the reads of vector elements are obsolete and can be omitted, as the linear combination reduces to a simple sum. In many cases a vector of positions is used in a mapped SpMV with the mesh matrix, to average over local neighborhoods in the mesh. As every input position consists of multiple components the number of threads can be increased such that the multiplication is carried out on a per component level. Without loss of generality, consider the case of averaging the vertex positions for each face, as it is done for example, when calculating face-points in the Catmull-Clark scheme. Each position consists of four components and each column in $\mathcal{M}$ has four non-zero entries. In this case an SpMV kernel can be constructed that is launched with 16 threads per face, each responsible for a single component of one vertex position. Each group of sixteen consecutive threads can then calculate the mapped multiplication of a single column. In the general case their intermediate products are then combined in the result using atomic addition operations. As it is known that each output component will only depend on intermediate products of four vertices, efficient SIMD communication primitives (shuffle instructions on NVIDIA hardware) can be used to combine the results. In that way the atomics can be dropped and the number of write accesses can be decreased, as the result for a component can be written by a single thread. Unfortunately the shuffle solution used in the transpose case is not applicable to the direct case, because threads which contribute their intermediate result to the same output element, are not necessarily in the same thread group.

**Fusion** Kernel fusion is an important paradigm in parallel computing, as it enables to reduce kernel launch overheads and costly memory loads and stores by merging kernels that have overlapping inputs or data dependencies. Whenever two operations in the high level linear algebra formulation require the same input vectors, and the number of threads required for both computations agree, the two generated kernels can be merged, such that data is not required to be loaded multiple times. The input to the fused kernel is then the union of the sets of inputs to the two individual kernels. The same is true for the output variables.

The second case where it is advantageous to fuse two kernels, is when the output of the first is the input to the second. In that case the data does not have to go through global memory from the first to the second kernel but can directly be used in the same kernel after it was computed. If the data that causes the dependency is not needed in any further computation it is not necessary to write it to global memory in the fused kernel. Especially for subdivision, where different output data is generated from shared input data, for example, edge points and face points, fusion can greatly reduce memory access. Thus, it can significantly improve performance, especially for memory bound kernels.

# 6 Evaluation

In this chapter the performance of the linear algebra approach and the implemented extensions such as feature adaptive subdivision and matrix reordering are evaluated by comparing to widely used and state of the art subdivision implementations.

**OpenSubdiv** [Tec17] was developed by Pixar and offers industrial strength subdivision implementations. OpenSubdiv uses a table based approach that involves three steps, of which the first two are considered preprocessing. In the first step, the topology is refined. This does not involve any vertex data such as positions but only connection information. Using the refined topology, the stencil tables are computed in a second step. These are the indices of control vertices which contribute in the calculation of a refined vertex. While the third step is done in parallel on the GPU, the two preprocessing steps are done entirely on the CPU. This is a problem in applications with frequently changing topology, because the topology refinement has to be done after each topological update and stencil tables have to be recomputed. OpenSubdiv does not support $\sqrt{3}$ subdivision, which is why we could only compare to their Loop and Catmull-Clark implementations. OpenSubdiv implements feature adaptive subdivision for the Catmull-Clark scheme, where only regions around irregular vertices are subdivided explicitly and for regular regions patches are created, which are tessellated using shaders.

**OpenMesh** [Uni17] is a framework for serial mesh processing. The used mesh representations allow for fast access to neighboring primitives, which is an advantage in mesh subdivision. OpenMesh supports all subdivision schemes that were implemented in our linear algebra framework.

All tests are performed on a Intel core i5-4690k with 16GB of RAM and an Nvidia Titan X (Pascal). All given timings are averages over several runs. The given measurements for our approaches are the sum of individual timings of all required kernel calls. The results for our approach, using standard linear algebra kernels and maps, are labeled *SpLA*. The results for the optimized approach that does not require an SpGEMM and that uses the specialized SpMVs is label *opt*.

**$\sqrt{3}$-Subdivision** The $\sqrt{3}$ is the computationally least costly of the ones implemented. After each step the number of triangles in the mesh tripled. Only OpenMesh supports this scheme.

**Loop Subdivision** The Loop subdivision splits each triangle into four new triangles in each iteration. Therefore, the number of faces grows faster than for $\sqrt{3}$-subdivision, which is one reason of the higher performance compared to Loop. This subdivision method is supported by all three approaches we compare to.

**Catmull-Clark: Additional Experiments** The Catmull-Clark subdivision is the most widely used method for mesh refinement. It is implemented in nearly any popular modeling software. To demonstrate the flexibility of our approach, we implemented some extensions, like feature adaptive subdivision. For that, we compare the table based approaches to subdivide irregular regions used by OpenSubdiv and the original code by Nießner et al. to our propagation based method. We also evaluate the impact of initial mesh reordering on the performance of OpenSubdiv, OpenMesh and our Catmull-Clark implementations.

## 6.1 Subdivision performance

For OpenSubdiv we list the individual timings of the two preprocessing steps *topology refinement* (ref.) and *stencil table creation* (sten.) and the total subdivision time (tot.) is given for all implementations. Tables 6.1 and 6.2 compare

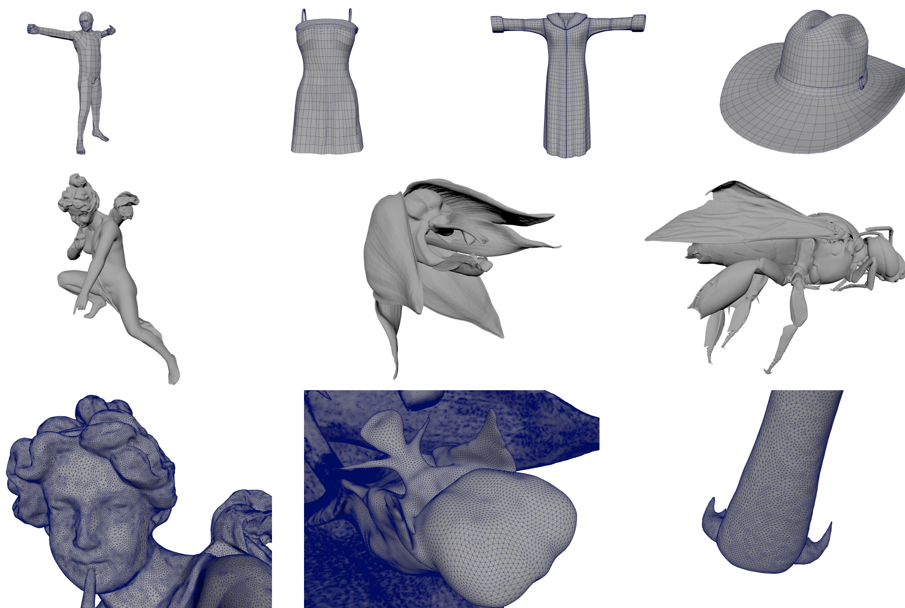


Figure 6.1: A variety of different models were used for evaluation. Some of them are shown here. Small models in the top row, large models in the bottom row and zooms of the large models in the bottom row. The Embreea Orchid and Eulaema Meriana Bee models are courtesy of The Smithsonian Institution.

the timings of our optimized approach to our unoptimized linear algebra implementation and to OpenSubdiv as well as OpenMesh. Models with a variety of face counts were used - for examples see Figure 6.1. The corresponding peak GPU memory consumption is given in Tables 6.3 and 6.4.

Data recorded in the undertaken experiments suggest that we outperform OpenSubdiv and OpenMesh for both, Loop and Catmull-Clark, while additionally having a lower memory footprint. Taking into account the whole subdivision process, our SpLA version reaches 14×-24× speedup to OpenMesh and 13×-42× compared to OpenSubdiv. The optimized implementation pushes the speedup further to 390×-620× for OpenMesh and 380×-1500× for OpenSubdiv. The large performance gain is mostly due to the full parallelization of the subdivision operation in our approach, eliminating any precomputations. Our optimized implementation only takes a little longer for the complete subdivision process than OpenSubdiv requires for the evaluation of the precomputed stencil tables.

| model;lvl | faces | | OM | OSD | | | | SpLA | OM/SpLA | OSD/SpLA | opt | OM/opt | OSD/opt |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | in | out | tot. | ref. | sten. | eval. | tot. | tot. | | | tot. | | |
| *archer;6* | 2k | 7M | 3468.41 | 578.37 | 9477.41 | 4.37 | 10060.15 | 248.26 | 13.97 | 40.52 | 7.54 | 460.00 | 1334.24 |
| *dress;6* | 2k | 9M | 4942.78 | 799.23 | 12634.02 | 5.41 | 13438.67 | 332.46 | 14.87 | 40.42 | 9.95 | 496.76 | 1350.62 |
| *Embreea Orchid;1* | 4M | 12M | 3683.69 | 476.10 | 2684.41 | 2.66 | 3163.18 | 167.76 | 21.96 | 18.86 | 8.30 | 443.82 | 381.11 |
| *phil;6* | 3k | 12M | 6606.27 | 1095.07 | 17550.84 | 7.95 | 18653.87 | 432.65 | 15.27 | 43.12 | 12.62 | 523.48 | 1478.12 |
| *hat;6* | 4k | 18M | 9697.63 | 1521.98 | 25143.67 | 11.52 | 26677.18 | 638.11 | 15.20 | 41.80 | 18.44 | 525.90 | 1446.70 |
| *coat;6* | 6k | 23M | 12223.98 | 2020.40 | 31563.84 | 14.99 | 33599.23 | 812.88 | 15.04 | 41.33 | 22.65 | 539.69 | 1483.41 |
| *neptune;2* | 4M | 48M | 17903.35 | 3506.40 | 27278.92 | 23.83 | 30809.16 | 1198.90 | 14.93 | 25.70 | 33.90 | 528.12 | 908.82 |
| *Eulaema Bee;1* | 17M | 51M | 16106.23 | 2074.85 | 12170.30 | 14.40 | 14259.56 | 718.10 | 22.43 | 19.86 | 34.78 | 463.09 | 410.00 |

Table 6.1: Performance comparison of running Uniform Catmull-Clark subdivision on different models. All timings in *ms*.

| model;lvl | faces | | OM | OSD | | | | SpLA | OM/SpLA | OSD/SpLA | opt | OM/opt | OSD/opt |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | in | out | tot. | ref. | sten. | eval. | tot. | tot. | | | tot. | | |
| *archer;6* | 3k | 13M | 4068.45 | 882.07 | 7048.93 | 3.09 | 7934.09 | 292.73 | 13.90 | 27.10 | 10.53 | 386.37 | 753.47 |
| *Embreea Orchid;1* | 4M | 16M | 2986.20 | 494.19 | 1754.34 | 1.95 | 2250.50 | 166.09 | 17.98 | 13.55 | 5.21 | 573.17 | 431.96 |
| *dress;6* | 4k | 18M | 5792.07 | 1235.65 | 9426.01 | 3.84 | 10665.50 | 418.90 | 13.835 | 25.46 | 14.61 | 396.45 | 730.01 |
| *phil;6* | 6k | 25M | 7770.21 | 1692.61 | 13802.41 | 6.19 | 15501.22 | 560.56 | 13.86 | 27.65 | 19.09 | 407.03 | 812.01 |
| *hat;6* | 9k | 36M | 11312.70 | 2457.62 | 18837.46 | 8.13 | 21303.23 | 816.18 | 13.86 | 26.10 | 27.41 | 412.72 | 777.21 |
| *coat;6* | 11k | 46M | 14803.18 | 3103.37 | 23295.68 | 10.59 | 26409.65 | 981.96 | 15.08 | 26.89 | 34.46 | 429.58 | 766.39 |
| *neptune;2* | 4M | 64M | 14984.42 | 3513.48 | 15161.60 | 13.45 | 18688.53 | 982.88 | 15.25 | 19.01 | 27.67 | 541.54 | 675.41 |
| *Eulaema Bee;1* | 17M | 68M | 13564.64 | 2103.95 | 7903.19 | 9.92 | 10017.07 | 708.60 | 19.14 | 14.14 | 21.75 | 623.63 | 460.53 |

Table 6.2: Performance comparisons for the Loop scheme; all timings in *ms* and all models are triangulated;

| Model;lvl | OSD | SpLA | opt |
|---|---|---|---|
| *hat;6* | 2.77 | 1.47 | 0.84 |
| *dress;6* | 1.37 | 0.74 | 0.43 |
| *phil;6* | 1.93 | 1.01 | 0.58 |
| *coat;6* | 3.47 | 1.85 | 1.06 |
| *archer;6* | 1.03 | 0.53 | 0.31 |
| *neptune;2* | 4.23 | 3.90 | 2.24 |
| *Embreea Orchid;1* | 0.62 | 0.80 | 0.61 |
| *Eulaema Bee;1* | 2.65 | 3.41 | 2.59 |

Table 6.3: Memory usage of the Uniform Catmull-Clark scheme; All measurements in *GB*

| Model;lvl | OSD | SpLA | opt |
|---|---|---|---|
| *hat;6* | 2.19 | 1.82 | 1.08 |
| *dress;6* | 1.08 | 0.92 | 0.54 |
| *phil;6* | 1.60 | 1.25 | 0.74 |
| *coat;6* | 2.73 | 2.29 | 1.36 |
| *archer;6* | 0.81 | 0.66 | 0.39 |
| *neptune;2* | 2.69 | 3.23 | 1.91 |
| *Embreea Orchid;1* | 0.46 | 0.80 | 0.47 |
| *Eulaema Bee;1* | 1.96 | 3.41 | 2.02 |

Table 6.4: Memory usage for the Loop scheme; All measurements in *GB*

| model;iter. | Nießner et al. | | | OSD | | | | SpLA | opt |
|---|---|---|---|---|---|---|---|---|---|
| | ref. + sten. | eval. | tot. | ref. | sten. | eval. | tot. | tot. | tot. |
| *hat;6* | 4.81 | 0.05 | 4.86 | 1.37 | 2.30 | 0.01 | 3.69 | 23.91 | 1.38 |
| *dress;6* | 29.13 | 0.06 | 29.19 | 5.33 | 18.10 | 0.01 | 23.45 | 27.48 | 1.59 |
| *phil;6* | 119.43 | 0.08 | 119.51 | 15.79 | 47.77 | 0.02 | 63.58 | 28.62 | 1.65 |
| *coat;6* | 17.38 | 0.05 | 17.44 | 3.53 | 7.88 | 0.01 | 11.42 | 24.24 | 1.85 |
| *archer;6* | 125.00 | 0.08 | 125.08 | 16.73 | 59.55 | 0.02 | 76.30 | 36.22 | 1.81 |

Table 6.5: Performance comparison for the Adaptive Catmull-Clark scheme; All timings in *ms*; all models subdivided to level 6;

| Model;iter. | Nießner et al. | OSD | SpLA | opt |
|---|---|---|---|---|
| *hat;6* | 0.18 | 0.23 | 1.48 | 0.98 |
| *dress;6* | 0.75 | 1.66 | 1.35 | 0.99 |
| *phil;6* | 2.96 | 4.42 | 3.62 | 2.67 |
| *coat;6* | 0.55 | 0.77 | 1.89 | 1.26 |
| *archer;6* | 2.99 | 5.01 | 4.27 | 3.15 |

Table 6.6: Memory usage comparison for the Adaptive Catmull-Clark approach; all measurements in *MB*

The results when exclusively subdividing regions around irregular vertices, as in a feature adaptive approach, are summarized in Tables 6.5 and 6.6. The SpLA approach performance deteriorates, as the overhead of feature extraction dominates the computations. Still, we are able to outperform the original implementation by Nießner et al. using our optimized approach. As the number of faces that require explicit subdivision is so small, the full power of the GPU can not be utilized.

OpenMesh is the only implementation in our test suite that supports $\sqrt{3}$ subdivision. Clearly, GPU implementations can outperform serial CPU implementations. Still, it is worth noting, that the changed view on $\sqrt{3}$ allowed for linear algebra formalizations and efficient GPU implementation. The results of the comparison are summarized in Table 6.7.

| Model;iter. | faces | | OpenMesh | SpLA | OM/SpLA | opt | OM/opt |
|---|---|---|---|---|---|---|---|
| | in | out | tot. | tot. | | tot. | |
| *fox;6* | 1k | 453k | 157.55 | 34.34 | 4.59 | 1.07 | 147.24 |
| *goblet_tri;6* | 1k | 729k | 259.15 | 43.67 | 5.93 | 1.28 | 202.46 |
| *Hhomer;6* | 10k | 7M | 3055.23 | 193.29 | 15.81 | 10.36 | 294.91 |
| *trim-star;6* | 10k | 8M | 3077.58 | 195.17 | 15.77 | 10.29 | 299.08 |
| *Embreea Orchid;1* | 4M | 12M | 1932.23 | 145.17 | 13.31 | 4.21 | 458.96 |
| *neptune;2* | 4M | 36M | 8582.17 | 676.20 | 12.69 | 20.14 | 426.13 |
| *girl_bust;6* | 61k | 45M | 18959.06 | 1173.01 | 16.16 | 79.48 | 238.54 |
| *Eulaema Bee;1* | 17M | 51M | 8525.38 | 621.24 | 13.72 | 15.67 | 544.06 |

Table 6.7: Performance comparisons for the $\sqrt{3}$ scheme, all timings in ms

## 6.2 Mesh reordering

To evaluate the influence of initial mesh matrix reordering on the subdivision performance, we subdivided ordered and original meshes using different Catmull-Clark approaches. Interestingly, the speedup for our optimized approach using an RCM reordered mesh compared the original mesh was as large as $8\times$, while for the other implementations in our comparison the maximum speedup was $2\times$.

| model;iter. | faces | | OM | | | OSD | | | SpLA | | | opt | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | in | out | orig. | re. | impr. | orig. | re. | impr. | orig. | re. | impr. | orig. | re. | impr. |
| *angel;1* | 474k | 1M | 728.76 | 434.92 | 1.68 | 562.80 | 381.00 | 1.48 | 29.81 | 25.93 | 1.15 | 3.69 | 1.50 | 2.46 |
| *angel;2* | 474k | 6M | 3566.70 | 2092.22 | 1.70 | 4669.25 | 3412.05 | 1.37 | 168.92 | 151.87 | 1.11 | 10.50 | 5.07 | 2.07 |
| *angel;3* | 474k | 23M | 15196.35 | 10662.78 | 1.43 | 25101.77 | 21176.15 | 1.19 | 745.03 | 675.06 | 1.10 | 29.67 | 18.35 | 1.62 |
| *Eulaema Bee;1* | 17M | 51M | 18444.64 | 17234.08 | 1.07 | 15936.20 | 14936.64 | 1.07 | 752.06 | 720.71 | 1.04 | 54.98 | 35.09 | 1.57 |
| *Beetle;1* | 2M | 6M | 3848.24 | 1886.73 | 2.04 | 3152.81 | 1639.66 | 1.92 | 134.90 | 89.70 | 1.50 | 42.90 | 5.63 | 7.62 |
| *Beetle;2* | 2M | 24M | 17934.28 | 9073.76 | 1.98 | 24603.65 | 14785.59 | 1.66 | 772.88 | 608.19 | 1.27 | 107.04 | 22.40 | 4.78 |
| *Blue Crab;2* | 11M | 34M | 12474.40 | 10774.76 | 1.16 | 10567.69 | 9265.71 | 1.14 | 546.01 | 480.84 | 1.13 | 42.04 | 23.83 | 1.76 |
| *Armadillo;1* | 346k | 1M | 487.01 | 316.02 | 1.54 | 401.63 | 277.40 | 1.45 | 22.26 | 19.45 | 1.14 | 2.61 | 1.07 | 2.45 |
| *Armadillo;2* | 346k | 4M | 2512.39 | 1492.77 | 1.68 | 3435.99 | 2473.22 | 1.39 | 125.48 | 111.71 | 1.12 | 7.68 | 3.44 | 2.24 |
| *Hunter;1* | 994k | 3M | 1888.75 | 918.37 | 2.06 | 1490.97 | 807.27 | 1.85 | 68.75 | 48.51 | 1.42 | 13.95 | 2.48 | 5.63 |
| *Hunter;2* | 994k | 12M | 8622.89 | 4381.06 | 1.97 | 11690.37 | 7335.97 | 1.59 | 382.69 | 308.79 | 1.59 | 36.16 | 8.95 | 4.04 |

Table 6.8: Subdivision time of applying uniform Catmull-Clark to original (orig.) and reordered (re.) meshes, as well as the achieved improvement (impr.); All timings in *ms*;

Our results suggest that reordering the meshes using RCM can yield a substantial speedup. Reordering the mesh after each subdivision step using RCM is too expensive. Finding a different reordering or mapping scheme, that could be applied after every iteration turned out to be a challenging task, as the additional computation cost compensated any improvement in performance.

## 6.3 Acknowledgments

The *Eulaema Bee*, *Embreea Orchid* and *Blue Crab* models are courtesy of The Smithsonian Institution [Ins18]. The *Armadillo* model was taken from The Stanford 3D Scanning Repository [Lab18]. The model *The Hunter and His Dog*, named Hunter in the result tables, is available from Lincoln 3D Scans [CLL18]. The *Beetle* model is from trimetric 3D Service GmbH [3D 18]. The Neptune model is available from Aim@Shape [vis18]. We want to thank all mentioned parties for making these 3D models available.

# 7 Conclusion

This thesis proposed the use of optimized linear algebra kernels to perform mesh subdivision on the GPU, using a sparse matrix to represent meshes and linear algebra primitives augmented with action maps to refine them. While the higher level linear algebra formulations are easy to understand and modify without the need of any knowledge about the underlying implementation, our approach surpasses state of the art performance. Continuous improvements of parallel linear algebra primitives translate directly to linear algebra subdivision. While the proposed principle are used for subdivision in this thesis, they can be used in general mesh processing applications to speed up computations.

The proposed approach is fully parallelized on the GPU. To the best of our knowledge, at the time of writing this thesis, there is no other subdivision implementation that runs entirely in parallel on the GPU without any CPU preprocessing. Three different refinement schemes were implemented to show that the approach is universal and the solution is not carefully crafted and optimized to reach good performance for a single approach only. Instead, we showed with the undertaken experiments that for every implemented approach we reach superior performance compared to widely used subdivision implementations and the current industry standard, OpenSubdiv. Even better performance can be reached by using specialized SpMVs for reoccurring tasks and by eliminating the SpGEMMs used to generate adjacency information. In particular, we reached up to three orders of magnitude speedup compared to OpenSubdiv, while not requiring any preprocessing. We showed that the linear algebra subdivision can simply be extended with additional features such as boundary handling or feature adaptive subdivision.

# Bibliography

[3D 18] trimetric 3D Service GmbH. *trimetric.* 2018. URL: http://www.trimetric.com/ (visited on 02/15/2018) (cit. on p. 57).

[Bau72] Bruce G. Baumgart. *Winged Edge Polyhedron Representation.* Tech. rep. Stanford, CA, USA, 1972 (cit. on pp. 2, 5).

[BPS93] Stephen T. Barnard, Alex Pothen, and Horst D. Simon. "A Spectral Algorithm for Envelope Reduction of Sparse Matrices." In: *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing.* Supercomputing '93. Portland, Oregon, USA: ACM, 1993, pp. 493–502. ISBN: 0-8186-4340-4. DOI: 10.1145/169627.169790. URL: http://doi.acm.org/10.1145/169627.169790 (cit. on p. 8).

[Bra+16] Wade Brainerd, Tim Foley, Manuel Kraemer, Henry Moreton, and Matthias Nießner. "Efficient GPU Rendering of Subdivision Surfaces Using Adaptive Quadtrees." In: *ACM Trans. Graph.* 35.4 (July 2016), 113:1–113:12. ISSN: 0730-0301. DOI: 10.1145/2897824.2925874. URL: http://doi.acm.org/10.1145/2897824.2925874 (cit. on p. 5).

[BS02] Jeffrey Bolz and Peter Schröder. "Rapid Evaluation of Catmull-Clark Subdivision Surfaces." In: *Proceedings of the Seventh International Conference on 3D Web Technology.* Web3D '02. Tempe, Arizona, USA: ACM, 2002, pp. 11–17. ISBN: 1-58113-468-1. DOI: 10.1145/504502.504505. URL: http://doi.acm.org/10.1145/504502.504505 (cit. on pp. 3, 6).

[CC78] Edwin Catmull and James Clark. "Recursively generated B-spline surfaces on arbitrary topological meshes." In: *Computer-Aided Design* 10.6 (1978), pp. 350–355. ISSN: 0010-4485. DOI: http://dx.doi.org/10.1016/0010-4485(78)90110-0. URL: http://www.sciencedirect.com/science/article/pii/0010448578901100 (cit. on pp. 17, 29).

[CKS98] Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. “Directed edges-A scalable representation for triangle meshes.” In: *Journal of Graphics tools* 3.4 (1998), pp. 1–11 (cit. on pp. 2, 5, 9).

[CLL18] The Collection, Lincoln, and Laric. *Lincoln 3D Scans*. 2018. URL: http://lincoln3dscans.co.uk (visited on 02/15/2018) (cit. on p. 57).

[CM69] Elizabeth Cuthill and James McKee. “Reducing the Bandwidth of Sparse Symmetric Matrices.” In: *Proceedings of the 1969 24th National Conference*. ACM ’69. New York, NY, USA: ACM, 1969, pp. 157–172. DOI: 10.1145/800195.805928. URL: http://doi.acm.org/10.1145/800195.805928 (cit. on pp. 7, 41).

[Coo84] Robert L. Cook. “Shade Trees.” In: *SIGGRAPH Comput. Graph.* 18.3 (Jan. 1984), pp. 223–231. ISSN: 0097-8930. DOI: 10.1145/964965.808602. URL: http://doi.acm.org/10.1145/964965.808602 (cit. on p. 7).

[Dav06] Timothy Davis. Fundamentals of Algorithms. Society for Industrial and Applied Mathematics, 2006. ISBN: 978-0-89871-613-9. DOI: 10.1137/1.9780898718881. URL: https://doi.org/10.1137/1.9780898718881 (cit. on pp. 12, 13).

[Der+17] Andreas Derler, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. “Dynamic Scheduling for Efficient Hierarchical Sparse Matrix Operations on the GPU.” In: *Proceedings of the International Conference on Supercomputing*. ICS ’17. Chicago, Illinois: ACM, 2017, 7:1–7:10. ISBN: 978-1-4503-5020-4. DOI: 10.1145/3079079.3079085. URL: http://doi.acm.org/10.1145/3079079.3079085 (cit. on p. 11).

[DKT98] 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*. SIGGRAPH ’98. New York, NY, USA: ACM, 1998, pp. 85–94. ISBN: 0-89791-999-8. DOI: 10.1145/280814.280826. URL: http://doi.acm.org/10.1145/280814.280826 (cit. on pp. 1, 5).

[DS98] Daniel Doo and Malcolm Sabin. "Seminal Graphics." In: New York, NY, USA: ACM, 1998. Chap. Behaviour of Recursive Division Surfaces Near Extraordinary Points, pp. 177–181. ISBN: 1-58113-052-X. DOI: 10.1145/280811.280991. URL: http://doi.acm.org/10.1145/280811.280991 (cit. on pp. 33, 36).

[Geo71] John Alan George. "Computer Implementation of the Finite Element Method." AAI7205916. PhD thesis. Stanford, CA, USA, 1971 (cit. on pp. 7, 41).

[GL81] Alan George and Joseph W. Liu. *Computer Solution of Large Sparse Positive Definite*. Prentice Hall Professional Technical Reference, 1981. ISBN: 0131652745 (cit. on p. 8).

[Ins18] The Smithsonian Institution. *Smithsonian Digitization*. 2018. URL: https://3d.si.edu/ (visited on 02/15/2018) (cit. on p. 57).

[Kob00] Leif Kobbelt. "$\sqrt{3}$-Subdivisionn." In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 103–112. ISBN: 1-58113-208-5. DOI: 10.1145/344779.344835. URL: http://dx.doi.org/10.1145/344779.344835 (cit. on pp. 17–20).

[Lab18] Stanford Computer Graphics Laboratory. *The Stanford 3D Scanning Repository*. 2018. URL: http://graphics.stanford.edu/data/3Dscanrep/ (visited on 02/15/2018) (cit. on p. 57).

[Loo87] Charles Loop. "Smooth subdivision surfaces based on triangles." In: (1987) (cit. on pp. 17, 23, 25, 26).

[LS08] Charles Loop and Scott Schaefer. "Approximating Catmull-Clark Subdivision Surfaces with Bicubic Patches." In: *ACM Trans. Graph.* 27.1 (Mar. 2008), 8:1–8:11. ISSN: 0730-0301. DOI: 10.1145/1330511.1330519. URL: http://doi.acm.org/10.1145/1330511.1330519 (cit. on p. 6).

[LV14] Weifeng Liu and Brian Vinter. "An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data." In: *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IPDPS '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 370–381. ISBN: 978-1-4799-3800-1. DOI: 10.1109/

IPDPS.2014.47. URL: http://dx.doi.org/10.1109/IPDPS.2014.47 (cit. on p. 11).

[Moo00] Gordon E. Moore. “Readings in Computer Architecture.” In: ed. by Mark D. Hill, Norman P. Jouppi, and Gurindar S. Sohi. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000. Chap. Cramming More Components Onto Integrated Circuits, pp. 56–59. ISBN: 1-55860-539-8. URL: http://dl.acm.org/citation.cfm?id=333067.333074 (cit. on p. 1).

[Nas87] Ahmad H. Nasri. “Polyhedral Subdivision Methods for Free-form Surfaces.” In: *ACM Trans. Graph.* 6.1 (Jan. 1987), pp. 29–73. ISSN: 0730-0301. DOI: 10.1145/27625.27628. URL: http://doi.acm.org/10.1145/27625.27628 (cit. on p. 7).

[Nie+12] Matthias Nießner, Charles Loop, Mark Meyer, and Tony Derose. “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. URL: http://doi.acm.org/10.1145/2077341.2077347 (cit. on pp. 3, 7, 36, 39, 40, 52, 55).

[NL13] Matthias Nießner and Charles Loop. “Analytic Displacement Mapping Using Hardware Tessellation.” In: *ACM Trans. Graph.* 32.3 (July 2013), 26:1–26:9. ISSN: 0730-0301. DOI: 10.1145/2487228.2487234. URL: http://doi.acm.org/10.1145/2487228.2487234 (cit. on p. 7).

[NVI17] NVIDIA. *The API reference guide for cuSPARSE, the CUDA sparse matrix library.* v9.0. NVIDIA. 2017 (cit. on p. 9).

[PEO09] Anjul Patney, Mohamed S. Ebeida, and John D. Owens. “Parallel View-dependent Tessellation of Catmull-Clark Subdivision Surfaces.” In: *Proceedings of the Conference on High Performance Graphics 2009.* HPG ’09. New Orleans, Louisiana: ACM, 2009, pp. 99–108. ISBN: 978-1-60558-603-8. DOI: 10.1145/1572769.1572785. URL: http://doi.acm.org/10.1145/1572769.1572785 (cit. on p. 3).

[Pet00] Jörg Peters. "Patching Catmull-Clark Meshes." In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 255–258. ISBN: 1-58113-208-5. DOI: 10.1145/344779.344908. URL: http://dx.doi.org/10.1145/344779.344908 (cit. on p. 6).

[Sch+15] Henry Schäfer, Jens Raab, Benjamin Keinert, Mark Meyer, Marc Stamminger, and Matthias Nießner. "Dynamic Feature-adaptive Subdivision." In: *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games*. i3D '15. San Francisco, California: ACM, 2015, pp. 31–38. ISBN: 978-1-4503-3392-4. DOI: 10.1145/2699276.2699282. URL: http://doi.acm.org/10.1145/2699276.2699282 (cit. on p. 7).

[SJP05] 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. URL: http://doi.acm.org/10.1145/1186822.1073304 (cit. on pp. 2, 6).

[Sta98] Jos Stam. "Exact Evaluation of Catmull-Clark Subdivision Surfaces at Arbitrary Parameter Values." In: *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '98. New York, NY, USA: ACM, 1998, pp. 395–404. ISBN: 0-89791-999-8. DOI: 10.1145/280814.280945. URL: http://doi.acm.org/10.1145/280814.280945 (cit. on p. 6).

[Ste+16] Markus Steinberger, Andreas Derler, Rhaleb Zayer, and Hans-Peter Seidel. "How naive is naive SpMV on the GPU?" In: *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. Sept. 2016, pp. 1–8. DOI: 10.1109/HPEC.2016.7761634 (cit. on p. 47).

[Sut05] Herb Sutter. "The free lunch is over: A fundamental turn toward concurrency in software." In: *Dr. Dobb's Journal* 30.3 (2005) (cit. on p. 1).

[SZS17] Markus Steinberger, Rhaleb Zayer, and Hans-Peter Seidel. "Globally Homogeneous, Locally Adaptive Sparse Matrix-vector Multiplication on the GPU." In: *Proceedings of the International Conference on Supercomputing*. ICS '17. Chicago, Illinois: ACM, 2017, 13:1–13:11.

ISBN: 978-1-4503-5020-4. DOI: 10.1145/3079079.3079086. URL: http://doi.acm.org/10.1145/3079079.3079086 (cit. on p. 11).

[Tec17] Pixar Graphics Technologies. *OpenSubdiv*. 2017. URL: http://graphics.pixar.com/opensubdiv/docs/intro.html (visited on 12/20/2017) (cit. on p. 51).

[TPO10] Stanley Tzeng, Anjul Patney, and John D. Owens. "Task Management for Irregular-parallel Workloads on the GPU." In: *Proceedings of the Conference on High Performance Graphics*. HPG '10. Saarbrucken, Germany: Eurographics Association, 2010, pp. 29–37. URL: http://dl.acm.org/citation.cfm?id=1921479.1921485 (cit. on p. 5).

[Uni17] RWTH Aachen University. *OpenMesh*. 2017. URL: https://www.openmesh.org/ (visited on 12/20/2017) (cit. on p. 51).

[vis18] visionair. *AIM@SHAPE*. 2018. URL: http://visionair.ge.imati.cnr.it (visited on 02/15/2018) (cit. on p. 57).

[WW01] Joe Warren and Henrik Weimer. *Subdivision Methods for Geometric Design: A Constructive Approach*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. ISBN: 1558604464 (cit. on p. 1).

[Zay08] Rhaleb Zayer. *Mesh Matrix Methods for Geometry Processing*. Tech. rep. Inria, Sept. 2008 (cit. on p. 21).

[Zho+09] Kun Zhou, Qiming Hou, Zhong Ren, Minmin Gong, Xin Sun, and Baining Guo. "RenderAnts: Interactive Reyes Rendering on GPUs." In: *ACM SIGGRAPH Asia 2009 Papers*. SIGGRAPH Asia '09. Yokohama, Japan: ACM, 2009, 155:1–155:11. ISBN: 978-1-60558-858-2. DOI: 10.1145/1661412.1618501. URL: http://doi.acm.org/10.1145/1661412.1618501 (cit. on p. 5).

[ZSS17] Rhaleb Zayer, Markus Steinberger, and Hans-Peter Seidel. "A GPU-Adapted Structure for Unstructured Grids." In: *Computer Graphics Forum* (2017). ISSN: 1467-8659. DOI: 10.1111/cgf.13144 (cit. on pp. 5, 6, 9, 12, 14, 15, 44).