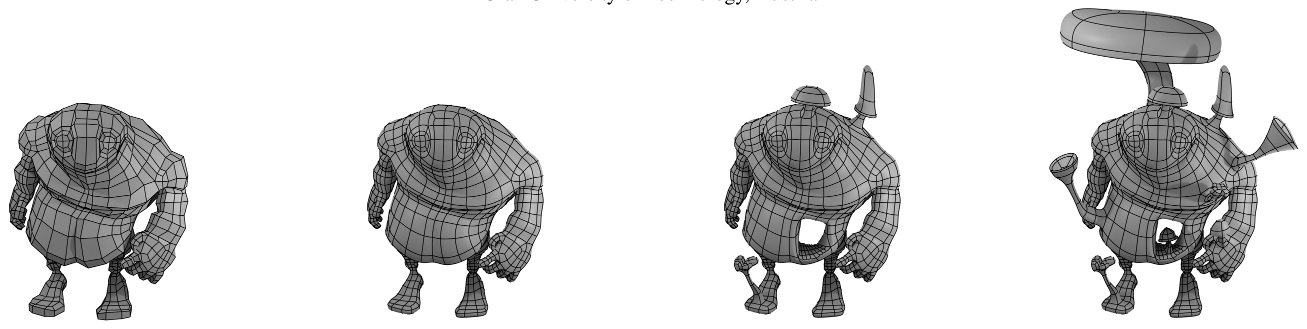
DOI: 10.1111/cgf.13934
EUROGRAPHICS 2020 / U. Assarsson and D. Panozzo
(Guest Editors)

Volume 39 (2020), Number 2

Subdivision-Specialized Linear Algebra Kernels for Static and Dynamic Mesh Connectivity on the GPU

D. Mlakar[1,2], M. Winter[2], P. Stadlbauer[1], H.-P. Seidel[1], M. Steinberger[2], R. Zayer[1]

[1]Max Planck Institute for Informatics, Germany
[2]Graz University of Technology, Austria



(a) *Control Mesh* **(b)** *Subdiv at lvl 6* **(c)** *14 topology changes* **(d)** *75 topology changes*

Figure 1: *A set of modeling operations applied to the control mesh of the BigGuy model (a). In total, a sequence of* 75 *connectivity changing operations are performed from (b) to (d). The current industry standard, OpenSubdiv, needs serial preprocessing after each topology change. These delays sum up to a two-minute idle time till (c) and an eleven-minute delay till (d). Using our subdivision-specialized linear algebra kernels, a modeler performs the whole sequence within two minutes with a consistent* 30 *fps preview of the subdivision surface at level six.*

Abstract

Subdivision surfaces have become an invaluable asset in production environments. While progress over the last years has allowed the use of graphics hardware to meet performance demands during animation and rendering, high-performance is limited to immutable mesh connectivity scenarios. Motivated by recent progress in mesh data structures, we show how the complete Catmull-Clark subdivision scheme can be abstracted in the language of linear algebra. While this high-level formulation allows for a fully parallel implementation with significant performance gains, the underlying algebraic operations require further specialization for modern parallel hardware. Integrating domain knowledge about the mesh matrix data structure, we replace costly general linear algebra operations like matrix-matrix multiplication by specialized kernels. By further considering innate properties of Catmull-Clark subdivision, like the quad-only structure after refinement, we achieve an additional order of magnitude in performance and significantly reduce memory footprints. Our approach can be adapted seamlessly for different use cases, such as regular subdivision of dynamic meshes, fast evaluation for immutable topology and feature-adaptive subdivision for efficient rendering of animated models. In this way, patchwork solutions are avoided in favor of a streamlined solution with consistent performance gains throughout the production pipeline. The versatility of the sparse matrix linear algebra abstraction underlying our work is further demonstrated by extension to other schemes such as $\sqrt{3}$ *and Loop subdivision.*

CCS Concepts

• ***Computing methodologies*** $\rightarrow$ ***Shape modeling;*** *Massively parallel algorithms;*

1. Introduction

Throughout four decades, subdivision surfaces have evolved from a pure research topic to an indispensable tool in 3D modeling packages and production software. This rise in prevalence is largely due to the performance gains achieved by adapting the evaluation part of the subdivision to take advantage of modern graphics hardware, as in OpenSubdiv [Pix19]. While this offers artists the ability to modify the vertex data, *e.g.* positions, interactively during simulation and animation, mesh connectivity must stay static. However, modelers change the mesh connectivity frequently, which causes slow, serial re-initialization of the subdivision process (*cf.* Fig. 1). Accelerating this step is challenging as the control mesh undergoes a series of averaging, splitting and relaxation operations, which complicate the problem of efficient parallelization of subdivision.

Existing efforts towards high performance subdivision usually follow one of two ideas: (1) Splitting the mesh into patches that can be subdivided independently [BS02, BS03, SJP05, PEO09] seems appealing for parallelization, but entails a series of issues. First, patches require overlap, introducing redundant data and computations, which may lead to cracks between patch boundaries due to floating point inaccuracies. Second, global connectivity is lost, as patches are treated independently. And third, re-patching and workload distribution are required as the model is subdivided recursively. (2) Factoring the subdivision into precomputation and evaluation [NLMD12, Pix19]. The bulk of the subdivision process is performed as preprocessing on the CPU, while the evaluation only performs simple vertex mixing on the GPU. While this is an ideal solution for parallel rendering of animated meshes, it is restricted to immutable topology, as the cost of CPU precomputation of subdivision tables is orders of magnitude higher than the GPU evaluation. Thus, these approaches are unusable for interactive modeling.

The lack of efficient, parallel and versatile subdivision approaches prompted a patchwork of solutions across production pipelines. When uniform subdivision is required, *e.g.*, for physics simulation, patch-based parallelization is used. During topology-changing modeling operations, only low-level previews of the full subdivision are shown to provide high performance. After modeling is completed, subdivision tables are used for animation. Finally, during rendering, partial subdivision or patch-based approaches are used to reduce the workload. As different approaches lead to slightly different results, the meshes used for simulation, preview, animation and rendering may differ in detail—the modeling experience is further spoiled.

In this work, we start with the mesh matrix formalism by Zayer et al. [ZSS17] to write geometry processing algorithms using sparse matrix operations. We extend their work to describe the complete Catmull-Clark subdivision scheme and reveal opportunities for parallelization. Combining this high-level view with low-level knowledge about execution on massively parallel processors, we propose a *flexible*, *high-performance* subdivision approach running entirely on the GPU. We make the following contributions:

- We extend Zayer's action map notation with lambda functions, increasing the formalism's expressiveness and versatility. Lambda functions for mapped matrix multiplication allow us to gather and create expressive adjacency data, which is vital for efficient topology changing operations during subdivision.
- We show that algebraic operations reveal potential for *parallelization* and optimization of data access and thus achieve significant performance gains compared to a serial approach.
- We combine the high-level algebra formulation with *low-level knowledge* about the execution platform to replace costly general algebra kernels with subdivision-specialized kernels, which are optimized for the target hardware platform and use domain knowledge about the subdivision process.
- We demonstrate that our approach is *modular* in the sense that topological operations can be separated from evaluation, leading to an efficient *parallel* preprocessing for immutable topology, followed by a single matrix-vector product vertex-data refinement.
- We extend our approach with sharp and semi-sharp creases and subdivision of selected regions, *e.g.*, for feature adaptiveness or path tracing, demonstrating its *extendability*.

Compared to the state of the art OpenSubdiv implementation commonly used in production, our specialized subdivision kernels achieve speed-ups of up to $1.7\times$ in the surface evaluation and over $15\times$ in preprocessing.

We further demonstrate the versatility of the sparse matrix linear algebra abstraction underlying our work by devising appropriate algorithmic formulations for additional schemes such as $\sqrt{3}$ and Loop subdivision and show consistent performance gains.

2. Related Work

Mesh subdivision has been honed for geometric modeling through the concerted effort of Chaikin [Cha74], Doo et al. [Doo78, DS78] and Catmull and Clark [CC78]. Subdivision meshes are commonly used across various fields, from character animation in feature films [DKT98] to primitive creation for REYES-style rendering [ZHR*09] and real-time rendering [TPO10].

2.1. Mesh Representations

Mesh subdivision is a refinement procedure, relying on data structures supporting fast adjacency queries and connectivity updates. Often variants of the *winged-edge* mesh representations [Bau72], like quad-edge [GS85] or half-edge [Lie94, CKS98] are used. While they are well suited for the serial setting, parallel implementations are difficult to achieve, require locks, suffer from scattered memory access and increased storage cost. Compressed formats designed for GPU-rendering, like triangle strips [Dee95, Hop99], do not offer connectivity information and are thus not suitable for subdivision. Therefore, patch-based GPU subdivision approaches have tried to find efficient patch data structures for subdivision [PS96, SJP05, PEO09].

2.2. Efficient Subdivision

Given the pressing need for high-performance subdivision, various parallelization approaches have been proposed. Shiue et al. [SJP05] splits the mesh into patches that can be subdivided independently on the GPU. This introduces redundancies and potentially cracks due to numeric inaccuracies. Patch-based approaches hide adjacency relations between patches, complicating further processing post subdivision. Subdivision tables have been introduced to efficiently reevaluate the refined mesh after moving control mesh vertices [BS02]. However, the creation of such tables requires a symbolic subdivision, whose cost is similar to a full subdivision. Table-based approaches are no solution to parallel subdivision, as the assembly of the tables is performed on the CPU and only the simple evaluation via weight mixing is done in parallel. Similarly, the precomputed eigenstructure of the subdivision matrix can be used for direct evaluation of Catmull-Clark surfaces [Sta98].

To avoid the cost induced by exact subdivision approaches, approximation schemes have been introduced. Peters [Pet00] transforms the quadrilaterals of a mesh into bicubic Nurbs patches, which imposes restrictions on the mesh. Loop et al. [LS08] represents the Catmull-Clark subdivision surface in regular regions using bicubic patches. Irregular faces still require additional computations. Approximations are fast, but along the way, desirable properties are lost and visual quality deteriorates. While regular faces can be rendered efficiently by exploiting the bicubic representation using hardware tessellation, irregular regions require recursive subdivision to reduce visual errors [NLMD12, SRK*15]. Brainerd et al. [BFK*16] improved upon these results by introducing subdivision plans. Beyond classical subdivision, several extensions have been proposed to allow for meshes with boundary [Nas87], sharp creases [DKT98], feature-based adaptivity [NLMD12] or displacement mapping [Coo84, NL13].

We provide a parallel subdivision approach that keeps track of the entire mesh while being able to do both: fully evaluate subdivision in one shot or split the processes into preprocessing and evaluation, both efficiently parallelized.

2.3. Matrix algebra

Using matrix algebra for subdivision was attempted before. The effort by Mueller-Roemer et al. [MRAS17] for volumetric subdivision uses boundary operators to boost performance on the GPU. While these differential forms have been used earlier [CRW05], their storage cost and redundancies continue to limit their practical scope. As an alternative to subdivision tables, Driscoll [Dri14] proposed the use of sparse matrix-vector multiplication to speedup per-frame evaluations. However, data conversion and processing cost makes it unsuitable for practical use. This suggests that using matrix algebra alone does not solve the problem of efficient subdivision. With our approach, we show that an extended matrix algebra in combination with bottom-up knowledge and optimizations is key to achieve modular, high-performance, parallel subdivision.

3. Mapped Matrix Algebra for Catmull-Clark Subdivision

To start the discussion of our approach, we review the sparse matrix mesh representation of Zayer et al. [ZSS17], propose our extensions to their formalism and describe how they can be used to derive Catmull-Clark subdivision using efficient sparse linear algebra.

3.1. Mesh Representation and Operations

We use the *sparse* mesh matrix $\mathcal{M}$ [ZSS17] in our linear algebra formulation. Each column in $\mathcal{M}$ corresponds to a face. Row indices of non-zeros in a column correspond to the face's vertices and the values reflect the cyclic order of the vertices locally within a face. For example, assume face i is comprised of vertices j,k,l,m in that order, then column i of $\mathcal{M}$ has entries in row j,k,l,m with values $1,2,3,4$, respectively.

Mapped SpMV [ZSS17] extends the common sparse matrix-vector multiplication (SpMV) to alter its outcome on the fly:

$$\mathbf{v}' = \underset{\mu}{M}\mathbf{v} = \underset{\sigma\to\delta}{M}\mathbf{v}; \quad \mathbf{v}'(i) = \sum_j \mu\left(M\left(i,j\right)\right)\mathbf{v}(j), \tag{1}$$

where M is a matrix, v is a vector and $\mu = \sigma \to \delta$ is an action map. $\mu = \sigma \to \delta$ is a user-defined, univariate function describing a mapping from the set of non-zero entries σ in M to a set of destination values δ. The mapping is performed on the fly, leaving M unchanged. This leads to a multiplication with a matrix of identical sparsity pattern but different values without explicitly creating it.

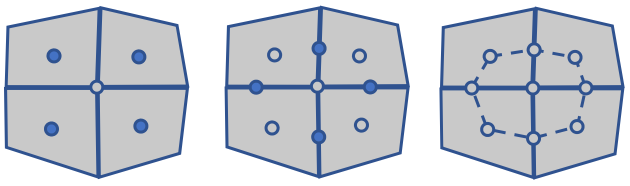


Figure 2: *The Catmull-Clark scheme inserts face-points (left), edge-points (center) and creates new faces by connecting face-points, edge-points and the original central point, which is updated in a smoothing step (right).*

Mapped SpGEMM proposed by Zayer et al. is not sufficiently general to formalize the full Catmull-Clark scheme. Thus, we extend the notation to offer more freedom in altering the result of a sparse general matrix-matrix multiplication (SpGEMM):

$$C = \underset{\{\mu\}[\lambda]}{AB}; \quad C(i,j) = \sum_k \lambda\left(\mu\left(A\left(i,k\right), B\left(k,j\right)\right), i,j,k,a,b\right),$$

where A, B, and C are sparse matrices, μ is an action map [ZSS17] and we call λ the lambda function. The map μ is a user-defined, bivariate function that maps from $\{A \times B\}$ to a set of values passed to the lambda function. The lambda function is user-defined and may perform arbitrary computations relying on information about the colliding non-zeros. During the multiplication, whenever a non-zero entry of A, *e.g.* $a = A(i,k)$, collides with a non-zero entry of B, *e.g.* $b = B(k,j)$, λ is invoked with parameters $\mu(a,b), i, j, k, a, b$ and performs the user-defined operations and returns a value that replaces the result of the multiplication $a \cdot b$ of the common SpGEMM. In contrast to action maps, lambda functions can capture any data available before the mapped SpGEMM is performed, which has two important implications: (1) Lambdas may use and manipulate data that would otherwise not be available in an SpGEMM, *e.g.*, vertex or face attributes. (2) Lambdas might have a state, *e.g.*, a lambda can count the number of collisions during the multiplication or create a histogram of non-zero values and alter their behavior based on the state. Thus, the matrix algebra captures data movement, while action maps and lambdas capture the operations to be carried out.

3.2. Catmull-Clark Subdivision

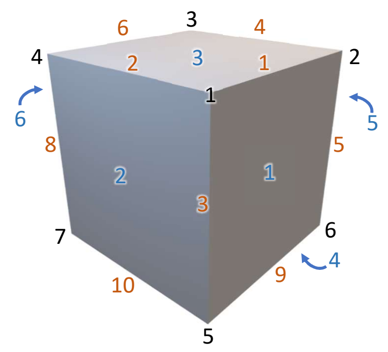
The Catmull-Clark scheme offers a generalization of bicubic patches to the irregular mesh setting [CC78]. It can be applied to polygonal faces of arbitrary order and always produces quadrilaterals. Fig. 2 outlines the four steps of a Catmull-Clark subdivision iteration.

Face-Points f_i are calculated for each face i by averaging the face's vertices. To compute the barycenters, face orders can be obtained using an action-mapped SpMV:

$$\mathbf{c} = \underset{val\to 1}{\mathcal{M}^T}\mathbf{1}, \tag{2}$$

where $\mathbf{1}$ is a vector of ones spanning the range of the faces. The mapping replaces all entries in $\mathcal{M}^T$ by 1. Thus, the SpMV counts the non-zero entries in each row of $\mathcal{M}^T$, *i.e.*, the number of vertices in each face, *cf.* Figure 3. This information is subsequently used in

$$\mathbf{f} = \underset{val_{i,*}\to\frac{1}{c_i}}{\mathcal{M}^T}\mathbf{P} \tag{3}$$



$$\mathcal{M}$$
$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 1 & 2 & 1 & & & \\ 2 & 4 & & 2 & & 4 & \\ 3 & & & 3 & & 3 & 3 \\ 4 & & 3 & 4 & & & 2 \\ 5 & 2 & 1 & & 1 & & \\ 6 & 3 & & & 4 & 1 & \\ 7 & & 4 & & 2 & & 1 \\ 8 & & & & 3 & 2 & 4 \end{array}$$

$$E$$
$$\begin{array}{c|cccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline 1 & & 2 & & 2 & 2 & & & \\ 2 & 1 & & 2 & & & 2 & & \\ 3 & & 4 & & 2 & & & & 2 \\ 4 & 2 & & 6 & & & & 2 & \\ 5 & 3 & & & & & 2 & 2 & \\ 6 & & 5 & & & 9 & & & 2 \\ 7 & & & & 8 & 10 & & & 2 \\ 8 & & & 7 & & & 11 & 12 & \end{array}$$

$$F$$
$$\begin{array}{c|cccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline 1 & & 3 & & 2 & 1 & & & \\ 2 & 1 & & 3 & & & 5 & & \\ 3 & & 5 & & 3 & & & & 6 \\ 4 & 3 & & 6 & & & & 2 & \\ 5 & 2 & & & & & 1 & 4 & \\ 6 & & 1 & & & 4 & & & 5 \\ 7 & & & & 6 & 2 & & & 4 \\ 8 & & & 5 & & & 4 & 6 & \end{array}$$

Figure 3: *The matrices used throughout one subdivision step for a small mesh (top left): mesh matrix $\mathcal{M}$, the edge indices provided in E and the adjacent face indices in F.*

to calculate the face-points, where $\mathbf{P}$ is the vector of all vertex data. Every non-zero value $val_{i,*}$ in row $\mathcal{M}^T(i,*)$ is mapped to the reciprocal of the number of vertices in face i.

Edge-Points are placed on every edge at the average of each edge's end-points and the face-points on its adjacent faces. The computation of edge-points requires the assignment of unique indices to mesh edges. Such an enumeration can be obtained from the lower triangular part of the adjacency matrix of the undirected mesh graph. With the standard linear algebra machinery, this matrix could be created by first computing the adjacency matrix of the oriented mesh graph and then summing it with its transpose to account for meshes with boundaries. This is not viable since it requires additional data creation (transpose) and, more importantly, matrix assembly, which is notoriously costly on parallel platforms.

We conveniently encode this step as a mapped SpGEMM

$$E = \underset{\{Q_c + Q_c^{c-1}\}[\iota]}{\mathcal{M}\mathcal{M}^T}, \tag{4}$$

where c is the face order, ι is a lambda function, and Q_c and its power Q_c^{c-1} are combined to capture the counterclockwise (CCW) and clockwise (CW) orientation inside a given face. For quads, Q_4 captures the CCW and Q_4^3 the CW adjacency. These two maps can be thought of as small circulant matrices, *e.g.*, of size 4×4 for quads, which are not created explicitly, as their entries can be computed on demand. This is particularly useful, when the face orders vary within a mesh:

$$Q_c^r(i,j) = \begin{cases} 1 & if \quad j = ((i+r-1) \bmod c) + 1 \\ 0 & \text{otherwise} \end{cases}. \tag{5}$$

The lambda function ι returns the number of faces shared by the vertices i and j. Thus, we can create unique indices for edges (and edge-points) by enumerating the non-zeros in the upper or lower triangular part of E as indicated by the orange entries in Figure 3.

To complete the computation of edge-points, faces adjacent to a given edge are required. For this purpose, we construct a second matrix, F, which has the same sparsity pattern as the adjacency matrix of the *oriented* graph of the mesh, but each non-zero entry stores the index of the face containing the edge. It can be similarly constructed by mapped SpGEMM:

$$F = \underset{\{Q_c\}[\gamma]}{\mathcal{M}\mathcal{M}^T} \tag{6}$$

$$\gamma(k,a,b) = \begin{cases} k & if \quad Q_c(a,b) = 1 \\ 0 & \text{otherwise} \end{cases}. \tag{7}$$

Whenever the action map returns a non-zero for a collision between elements $\mathcal{M}(i,k)$ and $\mathcal{M}^T(k,j)$, the face index k is stored in $F(i,j)$, see Figure 3. Hence, for each edge i,j in the mesh, its unique edge index is known from E and the two adjacent faces are $F(i,j)$ and $F(j,i)$. The edge-point position can then be computed.

Updating Vertex-Points The vertex data refinement is concluded by updating original vertex positions according to

$$S(p_i) = \frac{1}{n_i}\left((n_i - 3)\, p_i + \frac{1}{n_i}\sum_{j=1}^{n_i} f_j + \frac{2}{n_i}\sum_{j=1}^{n_i} \frac{1}{2}(p_i + p_j) \right), \tag{8}$$

where n_i is the vertex's valence, f_j are the face-points on adjacent faces and p_j the vertices in the 1-ring neighborhood of p_i. Eq. (8) can be conveniently rewritten as

$$S(p_i) = \left(1 - \frac{2}{n_i}\right) p_i + \frac{1}{n_i^2}\sum_{j=1}^{n_i} p_j + \frac{1}{n_i^2}\sum_{j=1}^{n_i} f_j. \tag{9}$$

With vertex valences obtained as the vector

$$\mathbf{n} = \underset{\sigma \to 1}{\mathcal{M}\mathbf{1}}, \tag{10}$$

the updated vertex locations can be written as three mapped SpMVs:

$$\mathbf{P}_{sub} = \underset{val_{i,i} \to 1 - 2n_i^{-1}}{I\mathbf{P}} + \underset{val_{i,*} \to n_i^{-2}}{E\mathbf{P}} + \underset{val_{i,*} \to n_i^{-2}}{\mathcal{M}\mathbf{f}}. \tag{11}$$

With I representing the identity matrix, the first mapped SpMV corresponds to an element-wise multiplication.

Topology Refinement The refined topology is built by inserting new edges that connect the face-point to the face's edge-points, splitting each face of order c into as many quadrilaterals. A new face consists of one (updated) vertex of the parent, its face-point and two edge-points (see Figure 2, right). We enumerate the subdivided vertices sequentially starting with the (updated) vertices, followed by the face-points and edge-points. To create the mesh matrix $\mathcal{M}_{sub}$ for the subdivided mesh, a column referencing the respective vertices for each face has to be created. Assume $\mathcal{M}$ has $|v|$ vertices and $|f|$

faces. $\mathcal{M}_{sub}$ can be created with the help of a mapped SpGEMM

$$T = \underset{\{Q_c\}[\sigma]}{\mathcal{M}\mathcal{M}^T} \tag{12}$$

$$\sigma = \begin{cases} \{i,\ E(i,j)+|v|+|f|, k+|f|,\ E(g(i,k),i)+|v|+|f|\} \\ 0 \quad if \quad Q_c(a,b)=0 \end{cases},$$

where g is a function that returns the predecessor vertex of a given vertex in a certain face. Each non-zero value of T is a quadruple that references the vertices of a face in the subdivided mesh, *i.e.*, the non-zeros form the columns of the subdivided mesh matrix $\mathcal{M}_{sub}$.

Combination Clearly, the described operations can be split into operations related to topology refinement and computing vertex location. Thus, one subdivision iteration can be split into what we call a build and evaluation step. The build step takes the current mesh matrix and generates F, E as well as the mesh matrix of the subdivided mesh. The evaluation step receives the matrices F and E as well as the mesh matrix and vertex positions from the last iteration to generate the new vertex locations. This split is possible over any number of iterations, carrying out all topology changing operations before subdividing the vertex data.

4. Mapping SpLA Catmull-Clark to Efficient Kernels

We implemented the high-level formalization discussed in Section 3 on top of standard sparse linear algebra (SpLA) kernels, which we extended with action maps. While those allow for efficient prototyping, they are not adapted to the particular computation patterns of the subdivision approach. In this section, we show that specialized GPU kernels designed for the structure of the underlying matrices and the exact computations carried out throughout subdivision can take full advantage of the parallel compute power of graphics hardware.

Data structure We use the Compressed Sparse Column (CSC) sparse matrix format, which is comprised of three arrays. The first two hold row indices and values of non-zero entries, both sorted by column. The third is a column pointer array that contains an offset to the start of each column in the first two arrays [Saa94]. CSC allows to efficiently access the vertices of a face, which is important during subdivision. Thus, the involved matrix operations can also be completed more efficiently than with a different format. To reduce memory requirements, we omit the value array in the mesh matrix and sort the row indices to reflect the traversal order of vertices in the mesh [ZSS17].

4.1. Adjacency, Offset and Mapping

Eq. (10) computes the valency for each vertex in the mesh by counting the number of non-zero entries in each row of $\mathcal{M}$. GPU SpMV would at first transpose the CSC format to allow for parallelization over the rows. As transpose is costly, we avoid it and consider the alternatives: gather and scatter. Gather would also parallelize over the rows, while searching through the columns of the CSC format. According to our experiments this does not improve performance compared to computing an explicit transpose. Thus, we use a scatter approach, which increases parallelism and improves read access; see Alg. 1. Each thread reads one non-zero from the mesh matrix (ln. 1). Consecutive threads read consecutive row indices, which yields perfect read access. Each thread uses atomic operations to increment the output vector element corresponding to its row and stores the old value in an offset array (ln. 2-3); note the perfect write access pattern. We use this offset, which enumerates the occurrences of each vertex, during later processing. While atomic operations cause overhead if they access the same memory location, the number of these collisions is limited to the valency of each vertex—which is low compared to the overall number of entries. Thus, scatter performs best among the presented alternatives.

ALGORITHM 1: Valency and offset calculation

```
input  : row indices of $\mathcal{M}$
output : vertex valences and offset array
1 $v \leftarrow$ Mrids $[t]$ ;                          // read vertex id
2 old $\leftarrow$ atomicAdd(valences $[v]$, 1) ; // increase valency
3 offsets $[t] \leftarrow$ old ;                  // store occurrence id
```

ALGORITHM 2: Filling non-zero entries of E

```
input  : sparsity pattern of $\mathcal{M}$, column pointer of $E$, offsets
output : row indices of and temporary values of $E$
1 first $\leftarrow$ Mcptr $[t]$ ;          // first entry in column $t$
2 next $\leftarrow$ Mcptr $[t+1]$;     // first entry in column $t+1$
  /* iterate over face $t$                                    */
3 for $k \leftarrow$ first to next $-1$ do
4     $i \leftarrow$ Mrids $[k]$;
5     $j \leftarrow$ nextVertInFace($i$, $k$);
6     off $\leftarrow$ Ecptr $[i]$+ offsets $[k]$ ;        // global offset
7     Erids [off] $\leftarrow j$;
8     Evals [off] $\leftarrow i < j$ ; // entry in lower tri. of $E$?
```

GPU SpGEMM is commonly performed in two steps. First, in the symbolic pass, the column pointer of the resulting matrix is determined. The multiplication is performed without generating the result, but only counts the number of non-zeros in each resulting column. A parallel prefix sum over those yields the column pointer of the resulting matrix. Second, the row indices and values are filled by running the multiplication routine again with the numeric operations. As SpGEMM must support arbitrary matrices, it is expensive on parallel devices and we want to avoid SpGEMM if possible.

As E stores the number of shared faces for any pair of vertices in the mesh, we can avoid the explicit SpGEMM of Eq. (4): First, the number of non-zeros in each column does not have to be computed. Each column is linked to a vertex with entries equal to the vertex' valence. A parallel prefix sum over the already available valences $\mathbf{n}$ yields the column pointer of E. Second, we directly compute the row indices and values of E, as outlined in Fig. 4 and Alg. 2: A column i in the resulting matrix has a non-zero entry in row j if vertices i and j share an edge. Thus, the row indices of E can be determined by inspecting the row indices of $\mathcal{M}$. We use one thread t per column of $\mathcal{M}$ (ln. 1-2). Each thread iterates over its column (ln. 4), creates an entry in E's row indices (ln. 8) for each edge i, j and writes a 1 to a temporary array if $i < j$ and 0 otherwise (ln. 9). To determine the position of an entry in the two arrays, the offsets from Alg. 1 are used, which enumerate the entries of each row in $\mathcal{M}$ (ln. 7)—each column in E has the same number of entries as a row in $\mathcal{M}$. A parallel prefix sum over the temporary value array

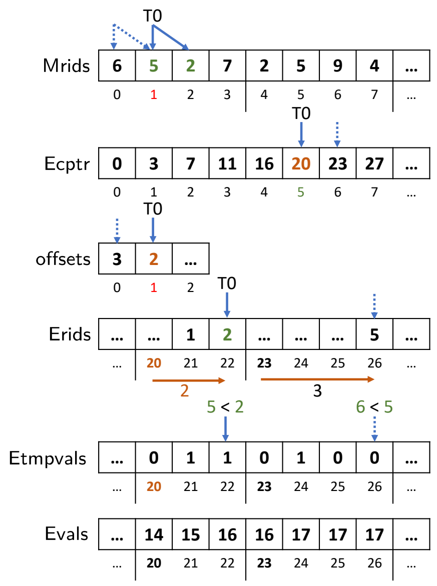


Figure 4: *Thread k operates on a single column of $\mathcal{M}$ and creates entries in E for each edge (i, j) of face k. The thread uses the column pointer of E and the offset array to determine the position p for the new entry in the row indices and values of E. The row index j is written to the row indices at position p. A subsequent write to a temporary value array at position p indicates whether $i < j$,* i.e.*, if the new entry is in the lower triangle of E. A prefix sum over the temporary value array yields the actual value array of E.*

yields the values of E, containing the unique edge indices in the lower triangular part of the resulting matrix. The index of edge i, j is stored in $E(\max(i, j), \min(i, j))$, which we denote as $E(i, j)$ in the following.

4.2. Topology Refinement

Commonly, we parallelize over the non-zeros of $\mathcal{M}$. For topology refinement, we need to know which face a non-zero belongs to. Using the CSC format, this would require a search in the column pointer of $\mathcal{M}$. To avoid this search, we attach an array to $\mathcal{M}$ holding the corresponding column/face index for each non-zero, effectively creating a hybrid with the coordinate format (COO) of $\mathcal{M}$.

Next, the subdivided topology can be created as in Eq. (12). Again, a straightforward implementation misses high performance goals. In Eq. (12) a collision, causing the lambda to return a face of the refined mesh, happens whenever two vertices are connected in CCW order in a face. This information is already contained in $\mathcal{M}$. Two non-zeros in the same column with consecutive values create a face of the refined mesh. Thus, we replace the mapped SpGEMM of Eq. (12) with a custom kernel, detailed in Alg. 3.

We parallelize over the non-zeros of $\mathcal{M}$, such that each thread builds a single face of the refined mesh. The original vertex and face index are trivially obtained from the previously computed COO form (ln. 1-2). To determine the vertex index of the next and previous index, we read the cyclic next and previous entry in the column

ALGORITHM 3: Refining the topology

```
input  : $\mathcal{M}$, $E$, Mcids
output : refined topology Mrids_ref
1 $v_0 \leftarrow$ Mrids $[t]$ ;                                  // vertex id
2 $f_0 \leftarrow$ Mcids $[t]$ ;                                    // face id
3 $v_n$ = nextVertInFace ($v_0$, $f_0$) ;
4 $v_p$ = prevVertInFace ($v_0$, $f_0$) ;
5 $v_1 \leftarrow E(v_0, v_n) + |v| + |f|$ ;                // out-edgepoint id
6 $v_2 \leftarrow f_0 + |v|$;                                  // in-edgepoint id
7 $v_3 \leftarrow E(v_p, v_0) + |v| + |f|$ ;                     // facepoint id
8 writeVec (Mrids_ref $[4t]$, vec4 ($v_0$, $v_1$, $v_2$, $v_3$)) ;
```

of $\mathcal{M}$ (ln. 3-4). As neighboring threads access the same column, these reads are often cached. We then compute the remaining vertex indices according to Eq. (12), using two entries from E (ln. 5-7). The number of vertices $|v|$ and faces $|f|$ are trivially tracked from one subdivision iteration to the next from the size of $\mathcal{M}$ and are directly provided to the specialized kernel. As each refined edge is a quad, we write the result using an efficient vector-store operation (ln. 8) with a perfect write memory pattern among threads.

4.3. Vertex Data Refinement

The face-points, *cf.* Eq. (3), are calculated in a single kernel given in Alg. 4. Each thread t is assigned to a column of $\mathcal{M}$ and averages all referenced vertices (ln. 2-4). As we store face-points next to one another in memory, the stores are coalesced (ln. 5).

ALGORITHM 4: Face-point calculation

```
input  : $\mathcal{M}$, vertex data
output : facepoints
1 val $\leftarrow 0$;
  /* iterate over face t                                            */
2 for $k \leftarrow$ Mcptr $[t]$ to Mcptr $[t+1] - 1$ do
3 |   $v \leftarrow$ Mrids $[k]$ ;                                // vertex id
4 |   val $\leftarrow$ val + data $[v]$;        // accumulate vertex data
5 facepoints $[t] \leftarrow \frac{\text{val}}{\text{Mcptr}[t+1]-\text{Mcptr}[t]}$ ;          // average data
```

An edge-point is given by the average of the two adjacent face-points and the two edge endpoints. To access those points the SpLA version uses the matrices E and F (Eq. (4) and (6)). However, relying on the topology information computed for the refined mesh (Alg. 3), we completely avoid the creation of F, as shown in Alg. 5. Each thread t is assigned a non-zero entry of the mesh matrix $\mathcal{M}$ and thus a face in the refined mesh (compare to Alg. 3). Using the original vertex index (ln. 2) and the already computed face-points (ln. 3), each thread adds its contribution to the edge-point on its outgoing edge (ln. 4). Thus, the computation of each edge-point is distributed to two threads and requires atomics (which show hardly any congestion).

The vertex update is the sum of three terms, see Eq. (11). We parallelize the first component-wise division over the elements and initialize the updated vertex array. To efficiently compute the second mapped SpMV we could again rely on atomics. However, as the sparsity pattern of E is symmetric, we instead multiply with the

ALGORITHM 5: Edge-point calculation

```
input  : refined topology Mrids_ref, vertex data, facepoints
output : edgepoints
   /* indices_ref = x:vp, y:ep_out, z:fp, w:ep_in */
1  indices_ref ← readVec(Mrids_ref [4$t$], 4); // refined face
2  vp ← data [indices_ref$_x$];                // vertex data
3  fp ← facepoints [indices_ref$_z$];          // facepoint data
   /* add contribution to edge-point */
4  atomicAdd(edgepoints [indices_ref$_y$], $\frac{1}{4}$(vp + fp));
```

vertex data from the left and thus parallelize over E's columns. In this way, atomics are avoided and writes are coalesced (and vectorized). For the third summand, we use the mapped SpMV parallelized over the non-zeros with atomics similar to Alg. 1.

4.4. Quadrilateral-Mesh Refinement

As the Catmull-Clark scheme exclusively produces quadrilaterals, further specialization of the subdivision kernels are possible, even if the input mesh has arbitrary face orders. As the number of faces increases exponentially with subdivision iterations, it is of particular importance to maximize the throughput for quadrilateral meshes. To keep the discussion concise, we only describe changes compared to the previous version.

Specialized Adjacency, Offset and Mapping The computation of row indices and values of E (Alg. 2) can be parallelized on a finer granularity: over the non-zeros instead of columns of $\mathcal{M}$. This eliminates the traversal of columns and thereby the access to the column pointer—because face i in a quad mesh starts at position $4i$ in the row indices of $\mathcal{M}$. Furthermore, the read access to the row indices of $\mathcal{M}$ improves because consecutive threads read consecutive entries. Each thread then reads the row index it is assigned and the next vertex index in the face and creates the respective entry in E, see Alg. 6. Additionally, we omit the creation of the COO format, as the mapping of a non-zero entry to a column directly follows from its position in the row index array; entries $4i$ to $4(i+1)-1$ belong to face i.

ALGORITHM 6: Filling non-zero entries of E in a quad mesh

```
input  : row indices of $\mathcal{M}$, column pointer of $E$, offsets
output : row indices of and temporary values of $E$
1  $v_0$ ← Mrids [$t$];
2  $v_n$ ← shuffleFromNextThreadInFace($v_0$);
3  off ← Ecptr [$v_0$] + offsets [$t$];
4  Erids [off] ← $v_n$;
5  Etmpvals [off] ← $v_0 < v_n$;
```

Specialized Topology Refinement In the topology refinement stage, we now use four threads to work cooperatively to subdivide an input quad. We still assign one thread to each non-zero element in $\mathcal{M}$. However, as four consecutive threads are guaranteed to execute on the same SIMD unit, they can communicate efficiently using so-called *shuffle* instructions. In the polygon refinement kernel, each thread originally read three row indices: its own and the two adjacent non-zeros in the same face (next and previous vertex index). Furthermore, two threads working on the same face queried E for the same edge-point index on the edge connecting the two vertices. For quad input meshes, we replace those additional accesses by shuffle instructions, as shown in Alg. 7. The index of the next vertex in the face v_n is shuffled from the (cyclic) next thread in the face (ln. 2). The face index is not read from the mapping as in the polygon subdivision kernel, but is computed from the thread index (ln. 4). As the incoming edge corresponds to the outgoing edge of the previous vertex, this edge index is also obtained using shuffling (ln. 3). Overall, each thread of the quad kernel reads two values, compared to five reads in the general kernel.

ALGORITHM 7: Refining the topology in a quad mesh

```
input  : Mrids, $E$
output : refined topology Mrids_ref
1  $v_0$ ← Mrids [$t$];
2  $v_n$ ← shuffleFromNextThreadInFace($v_0$);
3  $v_1$ ← $E(v_0, v_n) + |v| + |f|$;
4  $v_2$ ← $\lfloor \frac{t}{4} \rfloor + |v|$;
5  $v_3$ ← shuffleFromPrevThreadInFace($v_1$);
6  writeVec(Mrids_ref [4$t$], vec4($v_0$, $v_1$, $v_2$, $v_3$));
```

Specialized Vertex Data Refinement To calculate a face-point on a quad, vertex data of four vertices is averaged. Edge-points combine two vertices with two face-points, which means most of the data to calculate an edge-point is already available in the face-point computations. Thus, we fuse both into a single kernel to increase performance, as shown in Alg. 8. We use one thread per non-zero element in the mesh matrix. Each thread reads the non-zero row index and the corresponding vertex data (ln. 1-2). Four consecutive threads' data are summed, essentially performing a reduction using shuffle instructions (ln. 4-5). The sum is broadcast to all other threads assigned to the same face (ln. 6), as initially only the first of the four threads has the correct sum. Then, the final face-point is calculated and stored (ln. 7-8). For the edge-point, each thread combines its vertex data and the computed face-point and adds this contribution to the edge-point on the outgoing edge using an atomic addition (ln. 9-10). That way, two threads from two adjacent faces contribute to the edge-point on the shared edge.

ALGORITHM 8: Face- and Edge-points in a quad mesh

```
input  : $\mathcal{M}$, refined row indices Mrids_ref, vertex data
output : facepoints, edgepoints
1  $v_0$ ← Mrids[$t$];
2  vp ← data[$v_0$];
3  fp ← vp;
4  fp ← fp + shuffleDown(fp, 2, 4);
5  fp ← fp + shuffleDown(fp, 1, 4);
6  fp ← shuffle(fp, 0, 4);
7  fp ← $\frac{fp}{4}$;
8  facepoints[$\frac{t}{4}$] ← fp
9  $e_0$ ← Mrids_ref[$4t+1$] − $|v|$ − $|f|$;
10 atomicAdd(edgepoints [$e_0$], $\frac{vp+fp}{4}$);
```

In the vertex update, the average of face-points around each vertex (the third summand) can now omit the traversal of each column of $\mathcal{M}$, as each column has exactly four entries (Alg. 9). Each thread reads the four non-zeros of its column using a single vectorized load instruction and loads the corresponding face-point (ln. 1-2). It then proceeds to weight them and adds the result to the correct output vector element using atomics (ln. 3-6).

ALGORITHM 9: Calculation of $\mathbf{P}_{sub}$ in a quad mesh

```
input  : $\mathcal{M}$, facepoints, valences
output : $s_3$
1 face ← readVec(Mrids[$4t$], $4$);
2 fp ← facepoints[$t$];
3 atomicAdd($s_3$[face.first], fp/valences[face.first]$^2$);
4 atomicAdd($s_3$[face.second], fp/valences[face.second]$^2$);
5 atomicAdd($s_3$[face.third], fp/valences[face.third]$^2$);
6 atomicAdd($s_3$[face.fourth], fp/valences[face.fourth]$^2$);
```

5. Extensions through Efficient Kernels

Subdivision surfaces found their way from the lab to production when extensions where proposed. In this section, we extend our approach to address relevant aspects of such extensions.

5.1. Creases

Sharp and semi-sharp creases have become indispensable to describe piecewise smooth and tightly curved surfaces [DKT98]. Creases are realized as edges tagged by a (not necessarily) integer sharpness value and updated according to a special rules during subdivision. For an in-depth description of creases, we refer the reader to DeRose et al. [DKT98]; we present an efficient integration into our approach.

To support creases, we use a sparse, symmetric crease matrix C of size $|v| \times |v|$, with $C(i, j) = \sigma_{ij}$ being the sharpness value of the crease between vertex i and j. To subdivide creases, we need the crease valency $\mathbf{k}$, *i.e.*, number of creases incident to a vertex, and the vertex sharpness $\mathbf{s}$, *i.e.*, average of all incident crease sharpnesses. Both can be described by the same SpMV with different maps:

$$\mathbf{k} = \underset{val \to 1}{C\mathbf{1}}, \qquad \mathbf{s} = \underset{val_{i,j} \to \frac{val_{i,j}}{k_i}}{C\mathbf{1}}. \tag{13}$$

As C is symmetric, we again perform the summations over the columns rather than the rows. Furthermore, we merge both into a single kernel to reduce memory accesses. With the computed vectors $\mathbf{k}$ and $\mathbf{s}$ and the adjacency information in E, we correct crease vertices in parallel using the corresponding rules [DKT98], as an additional step after standard subdivision. Treating creases as a separate step avoids thread divergence and increases performance.

After each iteration, a new crease matrix with the updated sharpness values is required. We determine the crease values according to a variation of Chaikin's edge subdivision algorithm [Cha74] that decreases sharpness values [DKT98]:

$$\sigma_{ij} = \max\left(\frac{1}{4}\left(\sigma_i + 3\sigma_j\right) - 1, 0\right), \tag{14}$$

$$\sigma_{jk} = \max\left(\frac{1}{4}\left(3\sigma_j + \sigma_k\right) - 1, 0\right), \tag{15}$$

where σ_i, σ_j and σ_k are sharpness values of three adjacent parent creases i, j and k. σ_{ij} and σ_{jk} are the sharpness values of the two child creases of j. To allocate the memory for the new crease matrix, we count the number of resulting non-zeros in parallel over all columns and compute a prefix sum. In a second step, we perform the same computations again, but write the updated crease values. If all crease sharpness values decrease to zero, the subsequent subdivision steps are carried out identically as for a smooth mesh. Note that the core of the subdivision process remains the same; the crease matrix is created additionally in the build step. During evaluation, we re-evaluate and overwrite vertices influenced by a crease.

5.2. Selective and Feature Adaptive Subdivision

Our approach cannot only be used to describe uniform subdivision, but also selective processing. Consider feature adaptive subdivision, where only regions around irregular vertices are subdivided recursively, which is interesting for hardware-supported rendering [Pix19, NLMD12]: Using our scheme, extraordinary vertices are easily identified from Eq. (10), *i.e.*, where valency is $\neq 4$. To identify the regions around the extraordinary vertices, we start with a vector $\mathbf{s_0}$ spanning the number of vertices. $\mathbf{s_0}$ is 0 everywhere except for extraordinary vertices, where it is 1. To determine the surrounding faces, we propagate this information with the mesh matrix $\mathcal{M}$. The neighboring faces are determined as the non-zeros of

$$\mathbf{q_i} = \mathcal{M}^T \mathbf{s_i} \tag{16}$$

and their vertices can be revealed as the non-zero entries of

$$\mathbf{s_{i+1}} = \mathcal{M}\mathbf{q_i}. \tag{17}$$

We construct the matrix S_i which corresponds to an identity matrix with deleted rows according to $\mathbf{s_{i+1}}$. The extraction of the vertex-data is then performed by the SpMV

$$\mathbf{P_i'} = S_i \mathbf{P_i}. \tag{18}$$

To extract the mesh topology, the matrix $\mathring{S}_i$—analogue to S_i—is created from the information acquired in the propagation step. $\mathring{S}_i$ can again be created from the identity matrix by, in contrast to S_i, deleting *columns* corresponding to faces that should be disregarded during extraction. This information is readily available in $\mathbf{q_i}$. The extracted mesh matrix is then determined via

$$\mathcal{M}' = S_i \mathcal{M} \mathring{S}_i. \tag{19}$$

This can similarly be described as a mapped SpGEMM, replacing the two extraction matrices by identity matrices and mapping rows/columns to zero according to $\mathbf{s_i}/\mathbf{q_i}$. This would not explicitly reduce the size of the mesh matrix as Equation 19, but would set rows and columns to zero, that are not part of the extracted mesh.

5.3. Meshes with boundaries

In practice, meshes often contain boundaries, which require specialized subdivision rules. Catmull-Clark subdivision places edge-points on boundary edges on the edges' mid-points. A boundary vertex p_i is only influenced by adjacent boundary vertices

$$S(p_i) = \frac{3}{4}p_i + \frac{1}{8}(p_{i-1} + p_{i+1}). \tag{20}$$

Similar to creases, we handle mesh boundaries in a compute and overwrite fashion. First, the refined vertex-data is computed as usual. In a subsequent step, boundary vertices can be conveniently identified from E, and are replaced in parallel according to Eq. (20). Edge-points on edges connecting external vertices are set to the edge-mid points. Their indices can again be obtained from the enumeration of the non-zeros in the lower triangular part of E.

6. Operational Mode

Applications exhibit different requirements concerning subdivision approaches. Our method is versatile and can adapt to the current use-case by balancing computational cost between preprocessing (build) and vertex-data refinement (eval). However, we distinguish two main categories on opposite ends of the spectrum: dynamic and static topology of the control mesh.

6.1. Dynamic topology

Dynamic topology is ubiquitous in 3D modeling and CAD applications during the content creation process. Faces, vertices and edges are frequently added, modified and removed, which poses a great challenge to existing approaches, which rely on expensive preprocessing, as it has to be repeated on every topological update. This fact has led to the use of different subdivision approaches for model preview and production rendering, resulting in discrepancies between the two. Due to the efficiency of our complete approach, we can avoid any preprocessing and alternate between build steps and eval steps, computing one complete subdivision step before the next. As additional data like E_i is only needed for a single iteration, memory requirements are low.

6.2. Static topology

Static topology is common, *e.g.*, in production rendering, where vertex attributes, like positions, change over time but the mesh connectivity is invariant. Mesh connectivity information can be prepared upfront and does not require re-computation every frame, which reduces the overall production time. We factor all computations dealing with mesh connectivity throughout all subdivision levels into the build step, *i.e.*, generate all E_i and M_{i+1}; $\mathbf{s_i}$ and C_i in case of selective subdivision and creases; and F_i for the SpLA version. As control polygon vertices are updated, only the vertex position computations throughout all levels are computed during eval.

6.3. Single SpMV evaluation

Given that each iteration of eval is a sequence of mapped SpMVs, it is also possible to capture the entire sequence in a single sparse vertex subdivision matrix R_i. R_i captures the eval step of a single subdivision from level i to $i+1$:

$$\mathbf{P_{i+1}} = R_i\mathbf{P_i}. \tag{21}$$

Each column in R_i corresponds to a vertex at subdivision level i and each row corresponds to one refined vertex at level $i+1$. Thus, the entire evaluation from the first level to a specific level i can be written as a sequence of matrix-vector products as follows:

$$\mathbf{P_i} = R_{i-1}R_{i-2}\dots R_1R_0\mathbf{P_0} = \mathcal{R}\mathbf{P_0}. \tag{22}$$

Thus, the *whole eval step* can be rewritten as a *single SpMV* with the subdivision matrix $\mathcal{R}$, regardless of the subdivision depth.

Building Vertex Subdivision Matrices To construct $\mathcal{R}$, we first build the individual R_i, using the precomputed information that would otherwise be used in the normal eval step. First, we determine the number of non-zero entries in each row. A row in R_i reflects the influence of the previous iteration vertices on the vertices in the subdivided mesh. Thus, working with the compressed sparse rows (CSR) format, the transpose equivalent to CSC, is more efficient for constructing R_i. Consequently, the number of non-zero entries in a row is equal to the number of vertices that contribute to a vertex in the refined mesh. The first $|v|$ rows correspond to the updated original vertices and they receive a contribution from $1+\sum_i^n(c_i-2)$ vertices, where n is the number of neighboring faces and c_i their order. These entries can be obtained from the mapped SpMV

$$\mathbf{1} + \underset{val_{*,i}\to(c_i-2)}{\mathcal{M}\mathbf{1}}. \tag{23}$$

In a quad mesh the number of entries is $1+2n_i$, which we compute directly from $\mathbf{n}$. The next $|f|$ rows correspond to face-points, which are a linear combination of c_i vertices, which we have computed before (Eq. (2)) and is always 4 in a quad mesh. The remaining rows contain coefficients for edge-points, computed from c_l+c_r-2 control vertices, with c_l and c_r being the order of the two adjacent faces. These counts are computed with one thread per non-zero element in $\mathcal{M}$ and each thread in face i adds c_i-1 to the non-zero count of the edge-point on the outgoing edge. In a quad mesh the number of non-zeros in each row corresponding to a face-point is 6. As with CSC matrices, we use a parallel prefix sum over the non-zero counts to create the pointers array of R_i.

To fill the sparsity pattern with values, we again parallelize over the non-zeros of the mesh matrix. Each thread distributes the influence of the referenced vertex to the rows of all influenced vertices in the refined mesh, as shown in Figure 5. The indices of the influenced vertices are again directly taken from the refined topology and their contributions are given as follows:

- a face-point on an adjacent face is influenced with $f_d = \frac{1}{c}$.
- a vertex influences its updated location directly with $v_{s1} = 1-\frac{2}{n}$.
- a vertex influences its updated location indirectly via a single adjacent face-point with $v_{s2} = \frac{1}{n^2c}$
- a vertex connected via an edge with $v_d = \frac{1}{n^2}\left(1+\frac{1}{c_l}+\frac{1}{c_r}\right)$.
- a vertex connected via a face but no edge with $v_i = \frac{1}{n^2c}$.
- an edge-point on an incident edge with $e_d = \frac{1}{4}+\frac{1}{4c_l}+\frac{1}{4c_r}$.
- a non-incident edge-point of an adjacent face with $e_i = \frac{1}{4c}$.

SpMV Subdivision We construct $\mathcal{R}$ computing all SpGEMMs from Eq. (22). As $\mathcal{R}$ captures the combination of many mapped SpMVs, there is usually no common structure to exploit. However, we also use the CSR format to store $\mathcal{R}$ to allow efficient row access and perform the SpMV without atomic operations. Furthermore, we pad the row indices and value arrays, such that each row is 16 Byte aligned, to enable vectorized loads. For the same reason we also pad the vertex-data vector. For evaluation, we assign eight non-zeros to a single thread, which performs the multiplication with eight padded entries in the vertex array, *i.e.*, 32 values. Each thread needs to know

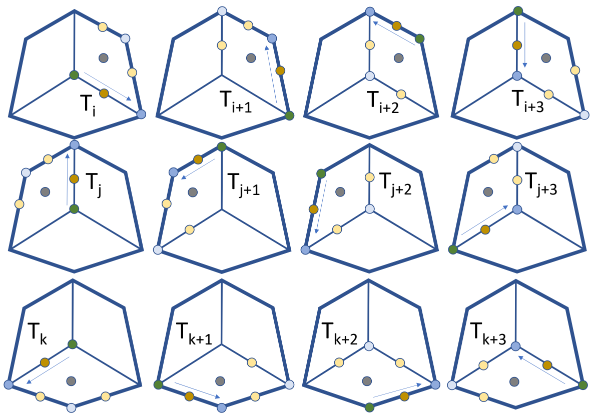


Figure 5: *The refinement matrix is built in parallel with one thread per non-zero element of the mesh matrix. Each thread adds one entry in each row that is influenced by its assigned vertex. Each thread's vertex contributes to an edge-point on the outgoing edge with e_d (orange), to edge-points on edges of the face that are not incident to the thread's vertex with e_i (yellow), to the face-point on the face the assigned non-zero element belongs to with f_d (gray), to the vertex it shares the outgoing edge with v_d (dark blue), to all other vertices of the face it does not share an edge with v_i (light blue) and finally to its own vertex with $v_{s1} + v_{s2}$ (green).*

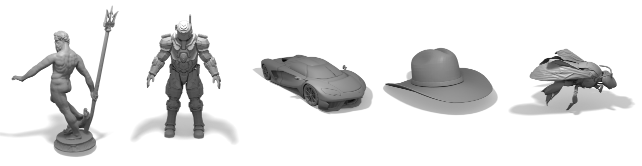


Figure 6: *Selection of evaluation meshes: Neptune, ArmorGuy (c/o DigitalFish), Car, Hat and Eulaema Bee (c/o The Smithsonian).*

the row index of its eight non-zero entries—this information is precomputed with $\mathcal{R}$ as neither the matrix nor the assignment changes during consecutive evaluations. We then use shuffle operations to merge results that correspond to the same row spread across multiple threads on the same SIMD unit. We collect data in on-chip memory using atomics and finally write the data coalesced to global memory.

7. Evaluation

We evaluate two implementations of the method presented in Section 3. The first approach uses common Linear Algebra Kernels (*LAK*) extended by action maps. The second approach uses the Specialized Linear Algebra Kernels (*SLAK*) as described in Section 4. While there is literature on parallel subdivision, there are hardly any implementations available for comparison. Thus, we compare to the current industry standard, OpenSubdiv (OSD), which is based on the approach by Nießner et al. [NLMD12] and splits subdivision into three steps. First, a symbolic subdivision is performed to create refined topology, which is then used in a second step to precompute the stencil tables. We summarize these two steps as *preprocessing*. The stencil tables are then used to perform the *evaluation* of refined vertex data, *i.e.*, vertex positions. While OpenSubdiv executes its *evaluation* on the GPU, *preprocessing* is entirely CPU-based. To provide a comparison to a complete GPU approach, we compare against Patney et al. [PEO09].

All tests are performed on an Intel Core i7-7700 with 32GB of RAM and an Nvidia GTX 1080 Ti. The provided measurements are the sum of all kernel timings required for the subdivision, averaged over several runs. We perform a variety of experiments on differently sized meshes (examples in Fig. 6) in order to evaluate subdivision performance. As LAK/SLAK can adapt to the specific needs of an applications, we distinguish two use cases which are on opposite sides of the full spectrum: "modeling" and "rendering".

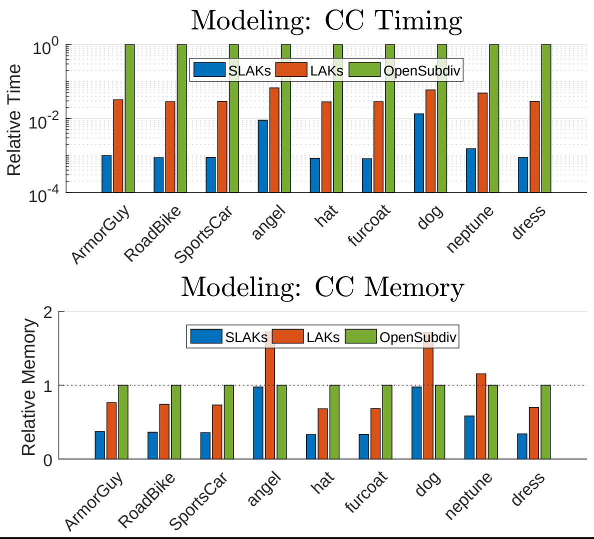
7.1. Catmull-Clark Modeling

This use case represents all applications in which the mesh connectivity changes frequently. This poses a challenge to approaches relying on precomputed data, *e.g.*, subdivision tables, as they have to be recomputed. Due to this fact, modeling software, like Blender, do not support OpenSubdiv in edit mode and instead rely on proprietary implementations. Similarly to using different subdivision levels for preview and rendering, proprietary solutions may show large visual differences between preview and final render.

Results for the modeling use case are shown in Fig. 7, where we evaluate the subdivision duration and peak memory consumption of the different approaches after a presumed topological change, *i.e.*, when the subdivision is re-initialized. LAK outperforms OpenSubdiv with an average speed-up of 26.6×, indicating that a complete GPU implementation is significantly faster than the split CPU-GPU approach of OpenSubdiv. SLAK is more than one order of magnitude faster than LAK and outperforms OpenSubdiv by more than two orders of magnitude, underlining that our specializations are highly effective. We did not include OpenSubdiv's memory transfer times between CPU and GPU, which would reduce it performance even further. Note that this is not the main use case of OpenSubdiv, which targets static topology. However, there is no efficient solution for this use case, underlining the importance of a complete parallelization to the problem. LAK needs similar or slightly more memory than OpenSubdiv's stencil tables due to the memory consumed by all matrices. SLAK reduces the memory of the mesh matrices and avoids the explicit creation of F and thus stays significantly below the memory requirements of the other two approaches.

Fig. 1 shows the clear advantage of using our approach in dynamic editing. Even at level six subdivision, SLAK yields results in real-time, whereas OpenSubdiv breaks the interactive modeling experience due to costly serial reprocessing. The accompanying video shows these circumstances for the same modeling sequence. Furthermore, as we perform all computations on the GPU, it is sufficient to transfer the updated geometry to the GPU instead of the complete subdivision tables. As our approach is instantaneous, it is also faster than previewing workarounds while being accurate.

As OpenSubdiv is more focused on efficient evaluation than optimizing the whole subdivision pipeline, we also compare to the GPU-based implementation by Patney et al. [PEO09], which we



			SLAK		LAK		OpenSubdiv		
Mesh	c_f	r_f	t	m	t	m	total	GPU	m
ArmorG.	8.6k	35.2M	39.1	1.8	1273.4	3.7	39506	24.4	4.8
RoadB.	53.9k	13.4M	14.4	0.7	470.0	1.4	16459	10.0	1.9
S.Car	149.7k	38.5M	40.3	2.0	1320.5	4.0	45471	34.8	5.5
angel	474k	1.4M	4.5	0.1	33.4	0.1	495	0.6	0.1
hat	4.4k	18.1M	20.3	0.9	684.5	1.9	24229	14.1	2.8
coat	5.6k	22.8M	24.3	1.2	850.5	2.4	29753	17.7	3.5
dog	994k	2.98M	18.8	0.2	83.41	0.3	1401	2.8	0.2
neptune	4.0M	48.1M	39.3	2.5	1263.0	4.9	25775	31.7	4.2
dress	2.3k	9.2M	10.6	0.5	350.6	1.0	12074	6.6	1.4

Figure 7: *Catmull-Clark subdivision time in ms after a topology changing operation was applied to the mesh. SLAK and LAK relative to OpenSubdiv and relative memory requirements (in GB). Details are given in the table. The number of control mesh faces c_f and the number of refined mesh faces r_f are provided as well.*

configured to perform uniform subdivision. We could only test small quad-only meshes, as their implementation fails when generating more geometry and on meshes with triangles. Nevertheless, as seen in Tab. 1, SLAK is about $2.6 - 5.6\times$ faster than the patch-based implementation of Patney et al.. We attribute this fact to our highly streamlined formulations and optimizations, which avoid redundant work and result in efficient memory movements. Aside from improved performance, we still have access to a fully connected mesh after subdivision compared to disconnected patches provided by Patney et al. These adjacency information is often required for further global processing or simulation of the subdivided mesh.

7.2. Catmull-Clark Rendering

In contrast to *modeling*, we consider topology static in the *rendering* use case. This enables efficient evaluation, as information that depends on mesh connectivity can be precomputed. Evaluation reuses this information to subdivide the vertex data in every render frame, *e.g.*, when replaying an animation. Using the modularity of SLAK, we rely on the split of *build* and *eval* to adapt to this use case.

Mesh	c_f	r_f	SLAK	Patney	$\uparrow$
bigguy	1.5k	371k	1.15	4.17	3.6
complex	1.4k	346k	0.96	4.07	4.2
cupid	29k	458k	0.72	3.65	5.1
frog	1.3k	331k	0.81	4.51	5.6
pig	381	390k	1.34	5.54	4.1
blocks	18	18k	1.07	2.75	2.6

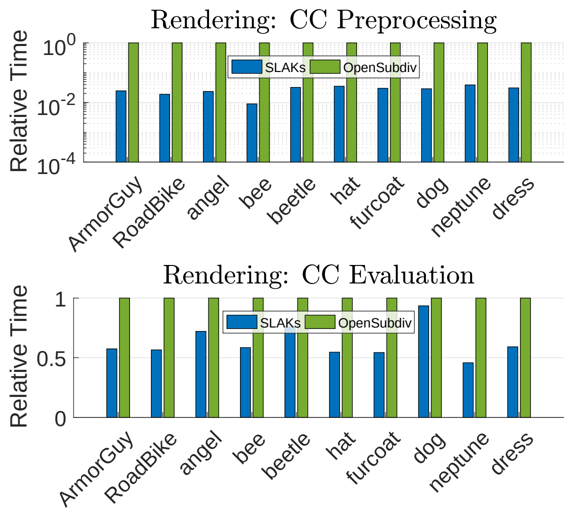
Table 1: *Comparison of SLAK with the GPU-based approach by Patney et al. for uniform subdivision from a given input mesh in ms. Our approach is* $2.6 - 5.6\times$ *faster* ($\uparrow$).

We present a detailed comparison of SLAK and OpenSubdiv as well as relative runtime and memory consumption in Fig. 8. We omit LAK in these figures to reduce clutter. However, in summary, LAK is on average $29.5\times$ better in preprocessing than OpenSubdiv, but $3.6\times$ slower in evaluation. Overall, SLAK leads the performance chart throughout all test cases—preprocessing and evaluation. In the preprocessing step, SLAK computes the refined mesh connectivity, assembles the sparse subdivision matrix and computes parameters for load balancing, exclusively on the GPU, outperforming OpenSubdiv's CPU preprocessing by more than an order of magnitude. In the evaluation phase, SLAK only performs a single SpMV, which is optimized using the precomputed load balancing scheme, outperforming OpenSubdiv by $1.6\times$. Note that OpenSubdiv's evaluation kernels have been optimized by NVIDIA, further underlining the efficiency of our evaluation step.

Our approach has similar memory requirements as OpenSubdiv in the uniform case, as the subdivision matrix and OpenSubdiv's stencil tables capture the same information. The memory requirement of SLAK is slightly higher if only one or two iterations are performed (Bee and Neptune). For higher subdivision levels, SLAK needs slightly less memory than OpenSubdiv.

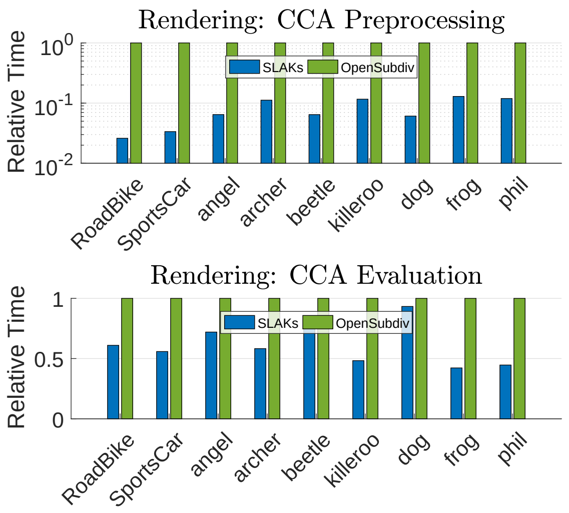
To reach real-time rendering performance, hardware-supported tessellation can be used for regular regions of the control mesh. However, regions around irregular vertices require full subdivision. To demonstrate that our approach can be used in this setting, we compare to the feature adaptive Catmull-Clark implementation of OpenSubdiv, which is based on the approach proposed by Nießner [NLMD12]. In this evaluation, regions around irregular vertices are successively subdivided and regular patches are split from the subdivision process.

Fig. 9 compares performance and required memory of our approach with OpenSubdiv. Formulating the extraction of irregular regions as a sequence of SpMVs that can directly be integrated into the global subdivision matrix is very efficient. Together with the parallel topology refinement, assembly and accumulation of the subdivision matrix SLAK performs preprocessing on average $15.5\times$ faster than OpenSubdiv. For evaluation, OpenSubdiv uses its stencil tables on the GPU, while we perform a single SpMV. While both approaches are similar in their nature, the simple static load balancing applied to SLAK's SpMV evaluation reflects in a speed-up of $1.7\times$ compared to OpenSubdiv's eval. Compared to the uniform subdivision from before, we observe that our optimizations work even better with the generally smaller subdivision matrices



Mesh	c_f	r_f	Prepro. SLAK	Prepro. OSD	Eval. SLAK	Eval. OSD	Memory SLAK	Memory OSD
ArmorG.	8.6k	35.2M	971	39481	14.0	24.4	4.57	4.93
RoadB.	53.9k	13.4M	311	16449	5.7	10.0	1.83	1.93
angel	474k	1.4M	12	494	0.4	0.6	0.08	0.08
bee	16.9M	50.8M	120	13290	9.4	16.0	2.91	2.72
beetle	2.0M	6.0M	93	2904	5.4	7.1	0.35	0.32
hat	4.4k	18.1M	850	24215	7.7	14.1	2.50	2.84
furcoat	5.6k	22.8M	892	29735	9.6	17.7	3.14	3.56
dog	994k	2.98M	40	1398	2.6	2.8	0.17	0.16
neptune	4.0M	48.1M	1000	25743	14.5	31.7	4.64	4.33
dress	2.3k	9.2M	373	12067	3.9	6.5	1.26	1.40

Figure 8: *Catmull-Clark subdivision: Evaluation of preprocessing and evaluation performance (in ms) as well as memory requirements (in GB) of SLAK and OpenSubdiv. c_f gives the number fo control mesh faces and r_f the refined mesh faces.*



Mesh	c_f	r_f	Prepro. SLAK	Prepro. OSD	Eval. SLAK	Eval. OSD	Memory SLAK	Memory OSD
RoadB.	54k	564k	101	3896	0.38	0.62	231	251
S.Car	150k	1.3M	306	9089	0.89	1.59	540	607
angel	474k	1.4M	52	799	0.41	0.58	82	76
archer	1.6k	11.7k	9	76	0.01	0.02	4	5
beetle	2.0M	6.0M	348	5400	5.41	7.14	346	320
k.roo	2.9k	9.9k	7	63	0.01	0.02	4	4
dog	994k	2.98M	154	2536	2.57	2.76	172	159
frog	1.3k	9.9k	8	60	0.01	0.02	3	4
phil	3.0k	9.9k	8	65	0.01	0.02	4	4

Figure 9: *Adaptive Catmull-Clark: preprocessing and evaluation performance (in ms) as well as memory requirements (in MB) of SLAK and OpenSubdiv. c_f gives the number fo control mesh faces and r_f the refined mesh faces.*

in the adaptive case. We believe this is due to our load balancing strategies for the single SpMV evaluation, which allows to draw more parallelism from the operations and thus increases relative performance for small matrices. SLAK performance increase is less pronounced for beetle and dog. On closer inspection we found that these model have a particularly bad layout, causing a high number of scattered memory accesses, which seems to have more influence on SLAK. Memory requirements are similar for both approaches.

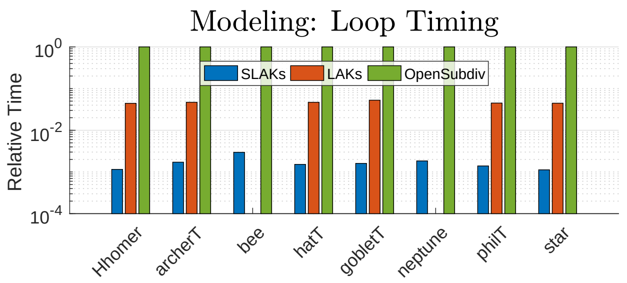
Considering the sum of these results, SLAK seems to be a suitable drop-in replacement for OpenSubdiv in both use cases, virtually removing preprocessing costs and increasing evaluation performance.

7.3. Loop and $\sqrt{3}$ Performance

The linear algebra machinery underlying our work naturally extends to other subdivision schemes. As a proof of concept, a brief algorithmic outline for $\sqrt{3}$ and Loop subdivision is given in Appendix A and B, respectively. The kernel specializations devised earlier can be applied to both schemes as well. A performance comparison for the modeling use case for Loop subdivision is given in Fig. 10, again showing that LAK and SLAK, both running completely on the GPU, outperform the partially CPU-based OpenSubdiv. LAK runs out of memory for the larger archerT and neptune models. $\sqrt{3}$ performance is shown in Tab. 2 comparing LAK and SLAK to the CPU-based OpenMesh. LAK is about one order of magnitude faster than OpenMesh and SLAK is about 20× faster than LAK. These results highlight that the speedups achieved for Catmull-Clark subdivision also carry over to other subdivision schemes.

8. Conclusion

We revisited Catmull-Clark subdivision from the ground up in the light of sparse linear algebra. To maintain an expressive and concise notation, we introduced lambda functions, which alter the result of matrix multiplications on the fly and thereby greatly increase flexibility and versatility of these operations. Using our extended formalism enables us to describe the full algorithm as a series of mapped SpMVs and SpGEMMs on the GPU. While our formalism can be implemented with minor adjustments to existing linear algebra kernels, we showed that the key to top-of-the-shelve performance is the combination of high-level domain knowledge with low-level knowledge about the execution platform.



Mesh	c_f	r_f	SLAK t	SLAK m	LAK t	LAK m	OpenSubdiv total	OpenSubdiv GPU	OpenSubdiv m
Hhomer	10.2k	41.8M	26.1	1.8	1003.0	2.2	22633	11.8	2.6
archerT	3.2k	13.1M	12.2	0.6	334.3	0.7	7117	3.7	0.8
bee	16.9M	67.8M	27.5	2.4	-	-	9289	11.2	2.0
hatT	8.8k	36.2M	29.0	1.6	894.1	1.9	19073	9.8	2.2
gobletT	1.0k	4.1M	3.6	0.2	117.4	0.2	2232	1.5	0.3
neptune	4.0M	64.1M	30.2	2.7	-	-	16383	16.5	2.8
philT	6.1k	24.9M	19.3	1.1	622.1	1.3	13840	7.6	1.6
star	10.4k	42.5M	25.5	1.8	1009.9	2.2	22650	11.5	2.6

Figure 10: *Loop subdivision time after a topology changing operation for SLAK and LAK relative to OpenSubdiv. Details in the table include timing in ms and memory in GB, the number of control mesh faces c_f and the number of refined mesh faces r_f.*

Mesh	c_f	r_f	SLAK	LAK	OpenMesh
fox	622	453.4k	1.24	35.16	127.40
girl_bust	61.3k	44.7M	93.18	1283.62	15394.00
goblet	1.0k	729.0k	1.49	42.23	207.64
Hhomer	10.2k	7.4M	12.59	207.25	2474.35
star	10.4k	7.6M	12.36	214.10	2481.43
bee	16.9M	50.8M	16.91	-	7520.05
neptune	4.0M	36.1M	22.64	-	7454.20

Table 2: $\sqrt{3}$*-subdivision time in ms for the GPU-based LAK and SLAK and the CPU-based OpenMesh implementation. c_f is the number fo control mesh faces and r_f the refined mesh faces.*

Our approach virtually removes idle times during subdivision surface design that are caused by expensive preprocessing in current approaches. Using our approach, modelers can modify the mesh topology and see an accurate preview of the subdivision surface instantly. Our experiments suggest that the applicability of our approach goes beyond the dynamic mesh connectivity setting, *i.e.*, modeling, as it outperforms the industry standard OpenSubdiv on static connectivity scenarios, *i.e.*, rendering, as well. Our approach operates fully on graphics hardware without requiring trips to the CPU. Furthermore, our formulation is open for extension with design features, as we showed for creases and selective subdivision. Thus, our approach can be readily integrated in all stages of the production pipeline. By open sourcing SLAK, we hope to inspire further developments in high performance geometry processing: `https://github.com/GPUPeople/SLAK`

9. Acknowledgments

This research was supported partly by the Max Planck Center for Visual Computing and Communication, the German Research Foundation (DFG) grant STE 2565/1-1, and the Austrian Science Fund (FWF) grant I 3007. Open access funding enabled and organized by Projekt DEAL. [Correction added on 24 March 2021, after first online publication: Projekt Deal funding statement has been added.]

References

[Bau72] BAUMGART B. G.: *Winged Edge Polyhedron Representation.* Tech. rep., Stanford University, Stanford, CA, USA, 1972. 2

[BFK*16] BRAINERD W., FOLEY T., KRAEMER M., MORETON H., NIESSNER M.: Efficient GPU Rendering of Subdivision Surfaces Using Adaptive Quadtrees. *ACM Trans. Graph. 35*, 4 (July 2016), 113:1–113:12. 3

[BS02] BOLZ J., SCHRÖDER P.: Rapid Evaluation of Catmull-Clark Subdivision Surfaces. In *Proceedings of the Seventh International Conference on 3D Web Technology* (New York, NY, USA, 2002), Web3D '02, ACM, pp. 11–17. 2

[BS03] BOLZ J., SCHRÖDER P.: Evaluation of Subdivision Surfaces on Programmable Graphics Hardware. 2

[CC78] CATMULL E., CLARK J.: Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design 10*, 6 (1978), 350 – 355. 2, 3

[Cha74] CHAIKIN G. M.: An algorithm for high speed curve generation. *Computer Graphics and Image Processing 3* (1974), 346–349. 2, 8

[CKS98] CAMPAGNA S., KOBBELT L., SEIDEL H.-P.: Directed edges-A scalable representation for triangle meshes. *Journal of Graphics tools 3*, 4 (1998), 1–11. 2

[Coo84] COOK R. L.: Shade Trees. *SIGGRAPH Comput. Graph. 18*, 3 (Jan. 1984), 223–231. 3

[CRW05] CASTILLO P., RIEBEN R., WHITE D.: FEMSTER: An Object-oriented Class Library of High-order Discrete Differential Forms. *ACM Trans. Math. Softw. 31*, 4 (Dec. 2005), 425–457. 3

[Dee95] DEERING M.: Geometry Compression. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1995), SIGGRAPH '95, ACM, pp. 13–20. 2

[DKT98] DEROSE T., KASS M., TRUONG T.: Subdivision Surfaces in Character Animation. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1998), SIGGRAPH '98, ACM, pp. 85–94. 2, 3, 8

[Doo78] DOO D.: A subdivision algorithm for smoothing down irregularly shaped polyhedrons. In *Proced. Int'l Conf. Ineractive Techniques in Computer Aided Design* (1978), pp. 157–165. Bologna, Italy, IEEE Computer Soc. 2

[Dri14] DRISCOLL M.: *Subdivision Surface Evaluation as Sparse Matrix-Vector Multiplication.* Master's thesis, EECS Department, University of California, Berkeley, Dec 2014. 3

[DS78] DOO D., SABIN M.: Behaviour of Recursive Division Surfaces Near Extraordinary Points. *Computer-Aided Design 10* (Sept. 1978), 356–360. 2

[GS85] GUIBAS L., STOLFI J.: Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi. *ACM Trans. Graph. 4*, 2 (Apr. 1985), 74–123. 2

[Hop99] HOPPE H.: Optimization of Mesh Locality for Transparent Vertex Caching. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1999), SIGGRAPH '99, ACM Press/Addison-Wesley Publishing Co., pp. 269–276. 2

[Kob00] KOBBELT L.: $\sqrt{3}$-subdivision. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2000), SIGGRAPH '00, ACM Press/Addison-Wesley Publishing Co., pp. 103–112. 14

[Lie94] LIENHARDT P.: N-dimensional generalized combinatorial maps and cellular quasi-manifolds. *International Journal of Computational Geometry & Applications 4*, 03 (1994), 275–324. 2

[Loo87] LOOP C.: Smooth subdivision surfaces based on triangles. 15

[LS08] LOOP C., SCHAEFER S.: Approximating Catmull-Clark Subdivision Surfaces with Bicubic Patches. *ACM Trans. Graph. 27*, 1 (Mar. 2008), 8:1–8:11. 2

[MRAS17] MUELLER-ROEMER J. S., ALTENHOFEN C., STORK A.: Ternary Sparse Matrix Representation for Volumetric Mesh Subdivision and Processing on GPUs. *Computer Graphics Forum 36*, 5 (2017), 59–69. 3

[Nas87] NASRI A. H.: Polyhedral Subdivision Methods for Free-form Surfaces. *ACM Trans. Graph. 6*, 1 (Jan. 1987), 29–73. 3

[NL13] NIESSNER M., LOOP C.: Analytic Displacement Mapping Using Hardware Tessellation. *ACM Trans. Graph. 32*, 3 (July 2013), 26:1–26:9. 3

[NLMD12] NIESSNER M., LOOP C., MEYER M., DEROSE T.: Feature-adaptive GPU Rendering of Catmull-Clark Subdivision Surfaces. *ACM Trans. Graph. 31*, 1 (Feb. 2012), 6:1–6:11. 2, 3, 8, 10, 11

[PEO09] PATNEY A., EBEIDA M. S., OWENS J. D.: Parallel View-dependent Tessellation of Catmull-Clark Subdivision Surfaces. In *Proc. HPG '09* (2009), ACM, pp. 99–108. 2, 10

[Pet00] PETERS J.: Patching Catmull-Clark Meshes. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2000), SIGGRAPH '00, ACM Press/Addison-Wesley Publishing Co., pp. 255–258. 2

[Pix19] PIXAR G. T.: OpenSubdiv, 2019. 1, 2, 8

[PS96] PULLI K., SEGAL M.: Fast Rendering of Subdivision Surfaces. In *Rendering Techniques '96* (Vienna, 1996), Pueyo X., Schröder P., (Eds.), Springer Vienna, pp. 61–70. 2

[Saa94] SAAD Y.: *SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations*. Tech. rep., Computer Science Department, University of Minnesota, Minneapolis, MN 55455, June 1994. 5

[SJP05] SHIUE L.-J., JONES I., PETERS J.: A Realtime GPU Subdivision Kernel. In *ACM SIGGRAPH 2005 Papers* (New York, NY, USA, 2005), SIGGRAPH '05, ACM, pp. 1010–1015. 2

[SRK*15] SCHÄFER H., RAAB J., KEINERT B., MEYER M., STAMMINGER M., NIESSNER M.: Dynamic Feature-adaptive Subdivision. In *Proc. i3D '15* (2015), pp. 31–38. 3

[Sta98] STAM J.: Exact Evaluation of Catmull-Clark Subdivision Surfaces at Arbitrary Parameter Values. In *Proc. SIGGRAPH '98* (1998), ACM, pp. 395–404. 2

[TPO10] TZENG S., PATNEY A., OWENS J. D.: Task Management for Irregular-parallel Workloads on the GPU. In *Proc. HPG '10* (2010), pp. 29–37. 2

[ZHR*09] ZHOU K., HOU Q., REN Z., GONG M., SUN X., GUO B.: RenderAnts: Interactive Reyes Rendering on GPUs. In *ACM SIGGRAPH Asia 2009 Papers* (New York, NY, USA, 2009), SIGGRAPH Asia '09, ACM, pp. 155:1–155:11. 2

[ZSS17] ZAYER R., STEINBERGER M., SEIDEL H.-P.: A GPU-Adapted Structure for Unstructured Grids. *Computer Graphics Forum 36*, 2 (May 2017), 495–507. 2, 3, 5

Appendix A: $\sqrt{3}$-Subdivision

The $\sqrt{3}$-subdivision scheme is specialized for triangle meshes and is based on a uniform split operator which introduces a new vertex for every triangle of the input mesh [Kob00]. It defines a natural stationary subdivision scheme with stencils of minimum size and maximum symmetry.

The subdivision process involves two major steps. The first one inserts a new vertex f_i at the center of every triangle i. Each new vertex is then connected to the vertices of its triangle and an edge flip is applied to the original edges, see Figure 11. In the second step, the positions of the old vertices are updated according to the smoothing rule

$$S(p_i) = (1-\alpha_i)p_i + \frac{\alpha_i}{n_i}\sum_{1}^{n_i} p_j, \tag{24}$$

where n_i is the valence of vertex p_i and p_j are vertices in its 1-ring neighborhood. The valency dependent factor α_n is obtained from the eigenstructure of the local subdivision matrix:

$$\alpha_i = \frac{1}{9}\left(4 - 2\cos\left(2\pi/n_i\right)\right). \tag{25}$$

Similar to other subdivision schemes, the topological operations involved in the $\sqrt{3}$-algorithm anticipate an edge-based mesh representation. All the implementations we are aware of rely on the half-edge data structure.

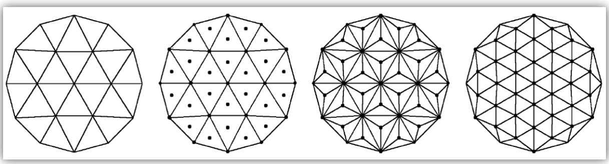


Figure 11: *Description of the $\sqrt{3}$-subdivision scheme. First a new vertex is inserted at every face of the given mesh. Second, an edge flip applied to the original mesh edges yields the final result, which is a 30 degree rotated regular mesh. Applying this scheme twice leads to a 1-to-9 refinement of each input triangle. Original image from [Kob00], copyright ACM.*

In order to adapt this subdivision scheme to our matrix algebra framework, we reinterpret the whole process in a slightly different manner. By reasoning only on triangles as detailed in Figure 12, the topological operations get simplified and the subdivision scheme can be abstracted using sparse matrix algebra. In fact, we need only a good bookkeeping of triangle-triangle adjacency to obtain new triangulations and update vertex positions. Please note that the boundary can be treated by adequate smoothing [Kob00] using similar ideas to the outline given earlier for the Catmull-Clark scheme, but we omit it here to keep the presentation succinct.

New vertex points A new vertex is added to each triangle's barycenter. The average of triangle vertices can be calculated using the mapped SpMV

$$\mathbf{f} = \underset{(1,2,3)\to\frac{1}{3}}{\mathcal{M}^T\mathbf{P}} \tag{26}$$

Required adjacency information The $\sqrt{3}$ scheme adds a vertex to each triangle and connects it to the new vertices on the three neighboring triangles. To find these neighbors efficiently, we can

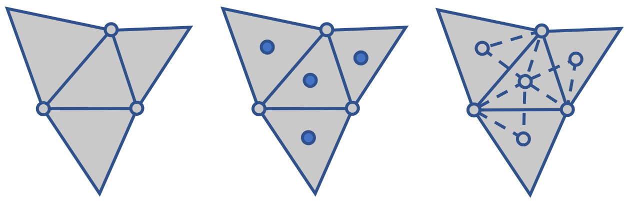


Figure 12: *After inserting the new vertices (blue), each triangle contributes three new triangles to the refined mesh by connecting its vertices to their left and right neighboring new vertices.*

again use the oriented graph adjacency matrix to store the index of the adjacent face to any given edge, as in Equations 6 and 7.

Vertex update The first term in 24 can be done in parallel in custamary ways. The second term can be efficiently computed in parallel by the mapped SpMV

$$\underset{val_i \to \frac{\alpha_i}{n_i}}{F\mathbf{P}}, \tag{27}$$

which, for each vertex, computes a linear combination of the vertex data in its 1-ring neighborhood.

Topology refinement For a mesh with $|v|$ vertices, each vertex (k,l,m) of a given triangle with index i contributes a new triangle to the refined mesh. For instance, vertex k contributes the triangle consisting of k itself, the barycenter on an adjacent triangle, which takes index $F(l,k)+|v|$ and the face-point on triangle i which can be conveniently indexed by $i+|v|$. The topology refinement can be performed efficiently in parallel with one thread per non-zero element in $\mathcal{M}$, each assembling one of the refined faces.

Appendix B: Loop Subdivision

This scheme is a triangle mesh subdivision method which was introduced by Charles Loop [Loo87]. It refines a mesh by inserting a new vertex on every edge as illustrated in Figure 13 (left). These edge-points are used to perform a 1-to-4 split of each input triangle. The original vertex positions are then smoothed by local weighted averaging as shown in Figure 13 (right). The weighted average in the vertex update step is defined as

$$S(p_i) = (1 - n_i\beta_i)p_i + \beta_i \sum_{1}^{n_i} p_j, \tag{28}$$

where n_i is the valence of vertex p_i and p_j are its incident vertices. The factor β_i depends on the vertex valence and is computed by

$$\beta_i = \frac{1}{n_i} \cdot \left(\frac{5}{8} - \left(\frac{3}{8} + \frac{1}{4} \cdot \cos\left(\frac{2\pi}{n_i} \right) \right)^2 \right). \tag{29}$$

In the following, we briefly describe the algebraic machinery we use to capture the topological modifications intrinsic to this scheme.

New vertex points To compute an edge-point on a given edge, the vertex insertion requires the edge end-points and the two vertices opposite to the edge in adjacent triangles. We can encode this information in the sparse adjacency matrix of the directed graph

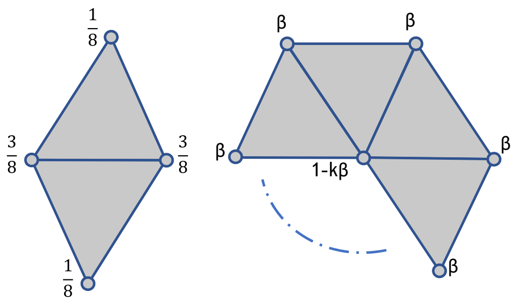


Figure 13: *For each edge, the Loop scheme inserts a new vertex, computed by a weighted sum of the vertices of the adjacent triangles (left). Original positions are updated using a β-weighted combination of the 1-ring neighborhood (right).*

of the mesh. For each (directed) edge, we store the index of the remaining triangle vertex as a non-zero value. This computations can be encoded as a mapped SpGEMM augmented with a lambda function:

$$G = \underset{\{Q_3\}[\lambda]}{\mathcal{M}\mathcal{M}^T} \qquad \lambda(i,j) = \begin{cases} k & if \quad Q_3 = 1 \\ 0 & else \end{cases} \tag{30}$$

where k is the vertex opposite to edge (i,j). Before the computation of edge-points can be completed, a unique index needs to be assigned to each edge. These can be obtained by summing G with its transpose G^T, which yields a matrix with the same sparsity pattern as the undirected adjacency matrix of the mesh. Incrementally assigning indices to the non-zeros of the lower triangular part of $G+G^T$ enumerates the edge-points, similarly to how it is done in the context of Catmull-Clark subdivision. The new vertex locations can then be obtained by looking up the edge-point indices and for each edge (i,j), determine the opposite vertices as $G(i,j)$ and $G(j,i)$ and performing the summation as given in Figure 13 (left).

Vertex update The first term in Equation 28 can be parallelized efficiently in a per element-fashion. The second term can be encoded and computed using the mapped SpMV

$$\underset{val_i \to \beta_i}{G\mathbf{P}}, \tag{31}$$

where the action map substitutes values in row i by β_i.

Topology refinement For a triangle with vertex indices (k,l,m) in the control mesh, three new triangles of the refined mesh are simple arrangements of an original vertex and two new edge-points. The fourth triangle is only composed of the three new edge-points. The original vertices' indices are k, l, m and unique edge indices can be obtained from the lower triangular part of $G+G^T$. With this information, the refined mesh matrix can be constructed efficiently in parallel. Each thread is assigned to a non-zero of $\mathcal{M}$ and three threads collaborate to write the four new triangle for each input face. For that, every thread determines the assigned vertex index and the index of the edge-point on the outgoing edge. Each of the three outer triangles is then constructed by two of the three threads. To build the center triangle each thread contributes the determined edge-point index.