High-Performance Graphics 2021
N. Binder and T. Ritschel
(Guest Editors)

Volume 40 (2021), Number 8

A Halfedge Refinement Rule for Parallel Catmull-Clark Subdivision

J. Dupuy and K. Vanhoey

Unity Technologies

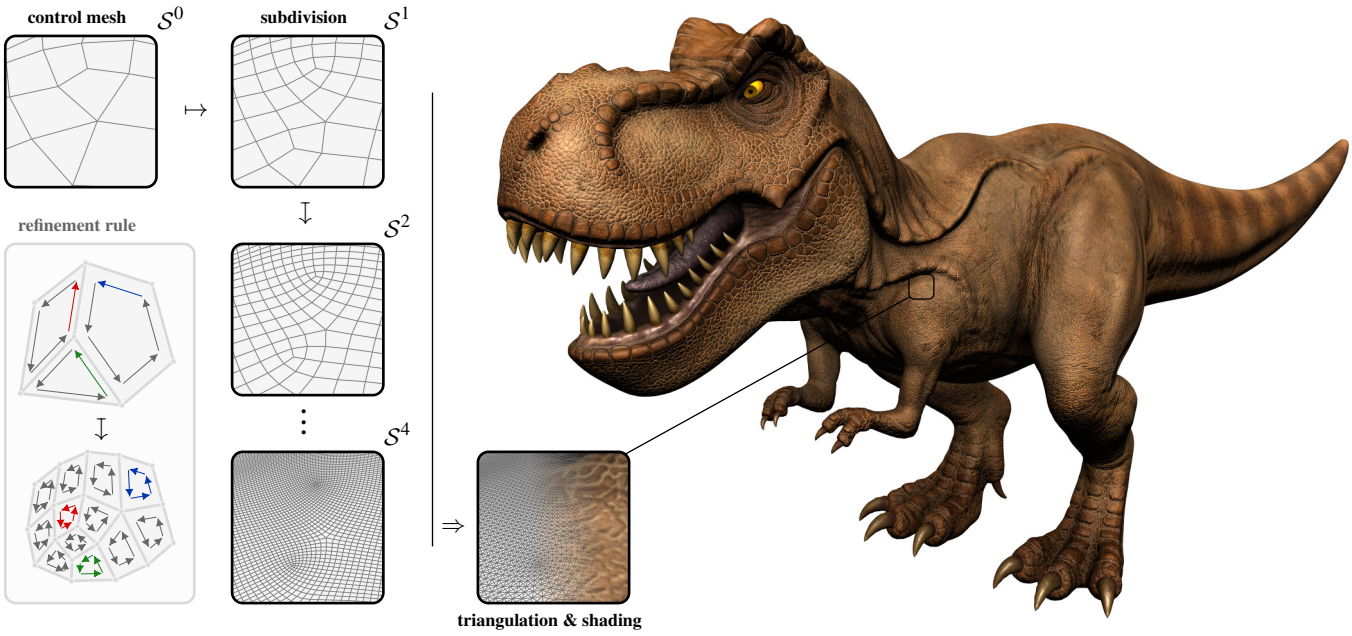


Figure 1: *We derive a halfedge refinement rule for Catmull-Clark subdivision. The rule is illustrated on the left: Catmull-Clark subdivision splits each halfedge into exactly 4 new ones independently from the face within which the subdivision operates (see highlighted halfedges). We leverage this rule in a novel GPU implementation that runs at state-of-the-art performances. For instance, the control mesh of this illustrated T-Rex production model consists of ∼11.5k faces and vertices. We compute its subdivision down to level 4, which produces ∼2.9M faces and vertices, in less than three milliseconds on an NVIDIA RTX 2080 GPU.*

Abstract

We show that Catmull-Clark subdivision induces an invariant one-to-four refinement rule for halfedges that reduces to simple algebraic expressions. This has two important consequences. First, it allows to refine the halfedges of the input mesh, which completely describe its topology, concurrently in breadth-first order. Second, it makes the computation of the vertex points straightforward as the halfedges provide all the information that is needed. We leverage these results to derive a novel parallel implementation of Catmull-Clark subdivision suitable for the GPU. Our implementation supports non-quad faces, extraordinary vertices, boundaries and semi-sharp creases seamlessly. Moreover, we show that its speed scales linearly with the number of processors, and yields state-of-the-art performances on modern GPUs.

CCS Concepts

• ***Computing methodologies*** → ***Massively parallel algorithms; Rendering;***

1. Introduction

Motivation First published back in 1978, Catmull-Clark subdivision [CC78] is one of the most durable and successful algorithms of computer-graphics: it lies at the very foundation of any modern surface-modeling tool and has been stimulating a wide-variety of research[†] up to the present day. Among the vast literature dedicated to Catmull-Clark subdivision (see, e.g., [Cas12] for a thorough survey), one of the most active research areas over the past two decades focuses on parallelization. This particular area is motivated by two main reasons. First, the geometric complexity of modern scenes makes it hard for a sequential implementation to cope with the exponential complexity induced by the algorithm. Second, there is a strong desire within the industry for interactive GPU-based implementations supporting both rendering and modeling scenarios [BFK*16, MWS*20]. In this work, we contribute to this research area via a novel parallel method for uniform subdivision on halfedge meshes suitable for both CPUs and GPUs.

Positioning While halfedges are ubiquitous in mesh processing [KUJ*14, SC*19, SB20, The21], they have been discarded by previous work for being inefficient in the context of parallel subdivision [SJP05, MWS*20]. Our work effectively disproves this belief by showing that halfedges lead *in fact* to a simple and fast parallel implementation that rivals performance-wise with existing methods based on more sophisticated implementations. From a practical standpoint, we also mention that our work is the only one (apart from OpenSubdiv) to support and distribute code for semi-sharp creases, an often overlooked yet crucial feature in practice.

Contributions and Outline We arrived at our algorithm by initially looking at how Catmull-Clark's refinement rule behaves with respect to the halfedges of the input mesh. In this setting, we observed that Catmull-Clark refinement multiplies the number of halfedges by exactly four; Figure 1 (see refinement rule inset) illustrates this property. Based on this simple insight, we introduce an invariant rule for the halfedges of the mesh under Catmull-Clark subdivision. Our rule always splits halfedges into four new copies whose attributes can be computed in breadth-first order via simple algebraic expressions. This makes parallelization trivial while providing all the topological information necessary to compute the vertex points produced by the subdivision. In the remainder of this paper, we derive our new invariant refinement rule and show how it straightforwardly leads to an efficient parallel implementation:

- In Section 3, we provide some fundamentals on Catmull-Clark subdivision and halfedge meshes. The reader already familiar with these concepts may look into Figure 2 for intuitions and immediately move to the next section.
- In Section 4, we introduce our invariant halfedge refinement rule, as well as the straightforward parallel implementation it leads to.
- In Section 5, we show how to support the semi-sharp crease extension of DeRose et al. [DKT98].
- In Section 6, we position our contributions with respect to previous work and compare our implementation's performances against state-of-the-art methods for different polygon meshes.

[†] Google Scholar references over 2849 citations, which seems to be the second most cited computer-graphics paper after Kajiya's (3568 citations).

2. Previous Work

In this section, we position our work with respect to existing parallelization methods for Catmull-Clark subdivision. We distinguish three main categories: patch-based, breadth-first, and direct evaluation, each of which is discussed in a dedicated paragraph. Note that for the sake of clarity, we defer discussions on adaptive refinement to the last paragraph of this section.

Patch-Based Refinement Patch-based refinement methods [BS02, Bun05, PO08, ZHR*09] decompose the input polygon-mesh into patches, each of which consists of a face and its one-ring neighborhood. In turn, these patches are refined independently down to the required subdivision level. This approach makes per-patch parallelization trivial since data-dependencies are avoided via duplication. Reciprocally though, it incurs redundant vertex-point computations. This is wasteful both in terms of memory and computations but worse is that, due to floating point inaccuracies, refined edges may differ between patches, yielding cracks [BS02]. Note that floating-point inaccuracies can be alleviated whenever computations solely consist of commutative operations, i.e., additions and multiplications [SJP05, NLMD12]. Depending on the approach however, this may not be feasible. Unlike patch-based refinement methods, ours is free from redundant vertex-point computation and wasteful memory allocations. We achieve this via a breadth-first approach, which we discuss next.

Breadth-First Refinement A breadth-first subdivision method successively refines a mesh data-structure down to the required subdivision level. The advantage of such an approach is that it makes vertex-point computations straightforward since all the necessary topological information is available. The downside is that it has traditionally been difficult to parallelize all refinement steps, especially when the input mesh has non-quad faces. For instance, the first parallel approach due to Shiue et al. [SJP05] requires two sequential refinement steps to first quadrangulate the input mesh and then isolate extraordinary vertices. Patney et al. [PO08] later alleviate the need to isolate extraordinary vertices but still require quad-only meshes. Recently, Mlakar et al. [MWS*20] introduced the very first method capable of parallelizing all refinement steps at unprecedented speed. Their method relies on a sparse data-structure [ZSS17] capable of coping with the fact that non-quad faces produce irregular numbers of faces during the first refinement step (see Figure 1). Our approach can be seen as a halfedge-based alternative to Mlakar et al. that avoids sparse data-structures entirely. This is thanks to the fact that Catmull-Clark refinement regularly splits halfedges into four new ones even in the presence of non-quad faces (see again Figure 1). We argue that our resulting implementation is much simpler than that of Mlakar et al. while yielding very close performances albeit at slightly higher memory costs. We also emphasize that each aforementioned publication requires a custom data-structure, while ours relies on the popular halfedge data-structure. Interestingly, halfedges have been discarded for being too complex and/or costly for parallel processors [SJP05, MWS*20]. This is probably because halfedges are often understood as linked-lists; in Section 3.3, we provide a pointerless approach that hopefully clarifies this misunderstanding.

Direct Evaluation Rather than relying on successive refinement to compute the subdivision, direct-evaluation methods focus on evaluating the limit-surface directly [NLMD12,SRK*15,BFK*16]. Note that the OpenSubdiv library builds directly on this idea, and more specifically on the work of Nießner et al. [NLMD12]. The original inspiration for such methods come from the fact that Catmull-Clark subdivision generalizes bicubic patches, for which direct evaluation is known. Later, Stam [Sta98] and Nießner et al. [NLG12] showed that certain non-regular configurations also produce a limit surface that can be directly evaluated. For other additional configurations, approximations also exist [LS08, KMDZ09, LSNCn09]. From a practical standpoint, the main advantage of direct evaluation lies in its ability to be evaluated by GPU tessellation-shaders, which provide hardware acceleration as well as adaptive triangulations within each tessellation-patch. Unfortunately though, direct evaluation remains restricted to specific geometric configurations and has yet to be proven applicable to the general case. Therefore, existing methods require a preprocessing stage that adaptively subdivides the input mesh into those geometries that lead to direct evaluation. Currently, this preprocessing stage is difficult to parallelize and/or memory-intensive, which makes it impractical for modeling scenarios [MWS*20]. We also mention that while direct-evaluation is often advertised over breadth-first approaches for their lower memory bandwidth requirements [NLMD12, BFK*16], the performance measurements of Mlakar et al. and those we report in Section 6 do not show any evidence that this leads to consistent performance advantages.

On Adaptive Subdivision Several works on parallel subdivision emphasize support for *adaptive* subdivision, i.e., the ability to refine subsets of an input polygon-mesh. This is usually relevant for GPU rasterization, which scales poorly in the presence of sub-pixel (triangle-based) tessellations. While such concerns are entirely justified, we argue that *subdivision* and *tessellation* are two distinct problems. In the case of feature-adaptive subdivision [NLMD12] for instance, a sequential *subdivision* is precomputed around, e.g., extraordinary vertices to create a set of patches that support direct evaluation and thus adaptive *tessellation* in parallel on the GPU. In our case and that of Mlakar et al. [MWS*20], nothing prevents us from applying our refinement rules non-uniformly, but the difficult problem of producing conforming and/or watertight mesh surfaces in parallel remains entirely open. We thus emphasize that we solely target the problem of computing Catmull-Clark subdivision surfaces in parallel and do not address the problem of adaptively tessellating them. We defer this latter problem to future work.

3. Preliminaries

In this section, we provide the fundamentals for computing Catmull-Clark subdivisions. First, we recall how Catmull-Clark subdivision operates on polygon meshes (Section 3.1). Next, we provide a self-contained formalism for halfedge meshes, which we leverage to process Catmull-Clark subdivision (Section 3.2). Finally, we describe the data-structure we use to represent our halfedge meshes in memory (Section 3.3). As stated in the introduction, we recall that the reader already familiar with these concepts may simply look into Figure 2 for intuitions and immediately move to the next section.

3.1. Catmull Clark Subdivision

Catmull-Clark subdivision applies a specific set of refinement rules on an input polygon mesh $\mathcal{S}^0$ to produce a denser, quad-only mesh $\mathcal{S}^1$. In turn, this new mesh may be used for subdivision to produce an even denser mesh $\mathcal{S}^2$. Repeating this operation $D \geq 1$ times produces the quad-only mesh $\mathcal{S}^D$, which converges towards a smooth surface as D increases. We provide below the specific set of refinement rules that characterize Catmull-Clark subdivision. Note that for the sake of clarity, we provide the original 1978 formulation of Catmull and Clark and describe the semi-sharp crease extension in Section 5.

Vertex Point Calculation Given an input polygon mesh $\mathcal{S}^{d\geq 0}$, Catmull and Clark determine the vertices of the new quad-only mesh $\mathcal{S}^{d+1}$ according to the following (A, B, C) rules:
(A) New face points – the average of all of the old points defining the face
(B.1) New boundary edge points – the midpoint of the old edge
(B.2) New smooth edge points – the average of the point produced by the boundary edge rule with the average of the two new face points of the faces sharing that edge
(C.1) New boundary vertex points – old vertex point
(C.2) New smooth vertex points – the average

$$\frac{Q}{n} + \frac{2R}{n} + \frac{S(n-3)}{n}, \tag{1}$$

where

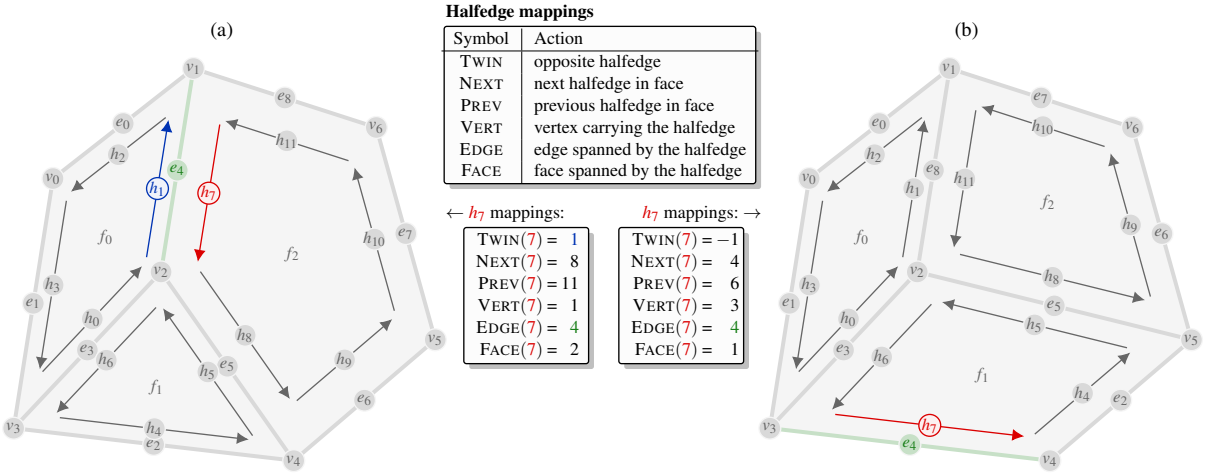
Q	=	the average of the new face points of all faces adjacent to the old vertex point.
R	=	the average of the midpoints of all edges incident to the old vertex point.
S	=	old vertex point.
n	=	valence of the old vertex point.

Topological Rules After the new vertex points have been computed, the edges of the new mesh $\mathcal{S}^{d+1}$ are formed by:

- connecting each face point to the new edge points of the edges defining the old face
- connecting each new vertex point to the new edge points of all old edges incident on the old vertex point

The faces of $\mathcal{S}^{d+1}$ are then defined as those enclosed by the new edges.

Implementation Prerequisite In order to process Catmull-Clark subdivision on a computer, the refinement rules impose strict requirements upon the data-structure used to represent the input mesh. Specifically: the data-structure should provide a mechanism to iterate over the faces, edges, and vertices of the mesh. In addition, it should provide support for vertex and face neighborhood queries. In the following subsection, we describe a well-known mesh decomposition suitable for producing a data-structure that complies with such requirements.



Label	Content											
h	0	1	2	3	4	5	6	7	8	9	10	11
TWIN	6	7	-1	-1	-1	8	0	1	5	-1	-1	-1
NEXT	1	2	3	0	5	6	4	8	9	10	11	7
PREV	3	0	1	2	6	4	5	11	7	8	9	10
VERT	3	2	1	0	3	4	2	1	2	4	5	6
EDGE	3	4	0	1	2	5	3	4	5	6	7	8
FACE	0	0	0	0	1	1	1	2	2	2	2	2

Label	Content (= quad-only analytic expression)											
h	0	1	2	3	4	5	6	7	8	9	10	11
TWIN	6	11	-1	-1	-1	8	0	-1	5	-1	-1	1
NEXT	$h \bmod 4 == 3 \,?\, h-3 : h+1$											
PREV	$h \bmod 4 == 0 \,?\, h+3 : h-1$											
VERT	3	2	1	0	4	5	2	3	2	5	6	1
EDGE	3	8	0	1	2	5	3	4	5	6	7	8
FACE	$h/4$											

Figure 2: *Two different polygon meshes with same vertex points and their respective halfedge buffer.*

3.2. Halfedge Mesh Formalism

A polygon mesh is a high-level concept. As such, it is not immediately amenable to computer processing without a prior decomposition into simpler, lower-level concepts. A halfedge mesh [Wei85, Ket99, BKP*10] provides one such decomposition. Specifically, a halfedge mesh decomposes a polygon mesh $\mathcal{S}$ into two sets: a set $\mathcal{V}$ of vertex points and a set $\mathcal{H}$ of halfedges, so that we can write $\mathcal{S} = \mathcal{H} \cup \mathcal{V}$. The vertex points encode the positional information for the mesh, while the halfedges–from which edges, and faces emanate naturally–encode its topology. We provide below our formal definition[‡] for the concept of halfedge and describe the mappings they provide; these mappings are summarized in Figure 2.

Halfedge Definition We define a halfedge simply as an oriented edge between two vertices of the mesh; Figure 2 (a) provides an illustration for this construction where, e.g., the red halfedge h_7 is the oriented edge carried by vertex v_1 and pointing towards v_2.

Face and Edge Construction Based on the halfedge definition, both the edges and faces of the mesh arise as follows:

- Each edge $e \in \mathcal{S}$ of the mesh $\mathcal{S}$ arises either as a pair of opposite halfedges, which we refer to as twins, or as a single isolated halfedge in the case of a boundary. More formally, we write that $\text{EDGE}(h) = \text{EDGE}(\text{TWIN}(h))$, where $h \in \mathcal{H}$ denotes a halfedge of the mesh $\mathcal{S}$ (note that in the case of a boundary, the notation $\text{TWIN}(h)$ does not refer to an existing halfedge). Figure 2 (a) provides an illustration for the edge construction where, e.g., the edge e_4 arises from the halfedge pair h_7 and $h_1 = \text{TWIN}(h_7)$, and e_1 is a boundary edge formed by the isolated halfedge h_3.
- Each face $f \in \mathcal{S}$ of the mesh $\mathcal{S}$ arises as a closed cycle of halfedges. More formally, we write that $\text{FACE}(h_0) = \cdots = \text{FACE}(h_{m>2})$, where the halfedges $h_0, \cdots, h_m \in \mathcal{H}$ form a closed cycle f within the mesh $\mathcal{S}$ – the resulting face is an m-gon. In addition, we respectively define $\text{NEXT}(h)$ and $\text{PREV}(h)$ as the next and previous halfedge within the (unique) cycle to which the halfedge h contributes. Figure 2 (a) provides an illustration for the face construction where, e.g., the face f_1 arises from the halfedge cycle h_4, $h_5 = \text{NEXT}(h_4)$, $h_6 = \text{PREV}(h_4)$, therefore forming a 3-gon, i.e., a triangle.

Halfedge-to-Vertex Mapping The halfedge-based constructions for edges and faces imply that each vertex $v \in \mathcal{V}$ spans at least $n \geq 2$ halfedges, where n denotes the valence of the vertex. More formally, we write that $\text{VERT}(h_0) = \cdots = \text{VERT}(h_n)$ where the halfedges $h_0, \cdots, h_m \in \mathcal{H}$ are those carried by v. Figure 2 (a) provides an illustration for the halfedge-to-vertex mapping where, e.g., the vertex v_2 carries the halfedges h_1, h_6 and h_8, i.e., $\text{VERT}(h_1) = \text{VERT}(h_6) = \text{VERT}(h_8)$.

Implementation Prerequisite The halfedge construction describes mesh topology solely through the use of halfedges and their mappings. Implementing a data-structure that evaluates these mappings is the key to computing Catmull-Clark subdivision. We describe such a data-structure in the next subsection.

[‡] We are not aware of a standard halfedge mesh formalism hence we provide our own.

3.3. Our Halfedge Mesh Data-Structure

There are multiple ways to transform formalism into an actual implementation. Here, we rely on a generalization of directed-edges [CKS98] to arbitrary polygon meshes (as opposed to triangle meshes originally). Similarly to directed-edges, we represent polygon mesh topology as array indices, which is the most suitable form for parallel processing, especially on a GPU. We provide below the details about the actual data we store.

Floats for Vertex Points Following directed-edges, we store the spatial coordinates of each 3D vertex point in a floating-point buffer of size $3V$, where $V \geq 3$ denotes the number of vertices of the mesh.

Array-Indices for Topology We represent the mesh topology $\mathcal{H}$ using a halfedge buffer. This buffer stores each fundamental halfedge mapping as listed in Figure 2 using $H \geq 3$ indices, where H denotes the number of halfedges of the mesh; Figure 2 (a) shows the content of the halfedge buffer for a specific mesh (N.B.: we encode a border half-edge h by setting $\textsc{Twin}(h) < 0$, without loss of generality). Note that, similarly to the directed-edges data-structure, we store the halfedges that span each face contiguously in memory. This is particularly useful when dealing with semi-regular meshes, as we discuss in the next paragraph.

Semi-Regular Mesh Optimization Catmull-Clark subdivision works for arbitrary polygon meshes but always produces quad-only meshes. We leverage this property for the subdivided meshes $\mathcal{S}^1, \cdots, \mathcal{S}^D$ by storing the halfedges forming a face contiguously. This makes the mappings NEXT, PREV, and FACE analytic; Figure 2 (b) provides the expression of the resulting mappings. In practice, we represent the control mesh topology $\mathcal{H}^0$ using the explicit representation of Figure 2 (a) and those produced by subdivision $\mathcal{H}^1, \cdots, \mathcal{H}^D$ using the representation of Figure 2 (b).

4. Parallel Catmull Clark Subdivision

In this section, we derive our parallel implementation of Catmull-Clark subdivision based on the halfedge mesh representation. We take an arbitrary polygon mesh $\mathcal{S}^0 = \mathcal{H}^0 \cup \mathcal{V}^0$ as control cage input and produce a quad-only mesh $\mathcal{S}^D = \mathcal{H}^D \cup \mathcal{V}^D$ by successive application of Catmull-Clark rules over $D \geq 1$ steps. We start by deriving our novel halfedge refinement rule and leverage it to successively compute the topologies $\mathcal{H}^1, \cdots, \mathcal{H}^D$ (Section 4.1). Next, we show how to leverage this topology information to successively produce the vertex points $\mathcal{V}^1, \cdots, \mathcal{V}^D$ (Section 4.2). Finally, we provide implementation details and performance evaluation on multiprocessors (Section 4.3).

4.1. Halfedge Refinement

Catmull-Clark subdivision always produces quad-only meshes. This implies a simple yet powerful consequence for halfedges: the subdivision splits the halfedges of the input mesh into exactly 4 new ones; Figure 3 highlights this effect on the red and blue halfedges. In this section, we introduce a stationary halfedge refinement rule that produces the topology $\mathcal{H}^{d+1}$ given $\mathcal{H}^d$ for any depth $d \in [0, D]$.

Refinement Rule We derive a breadth-first halfedge refinement rule: the h-th halfedge in $\mathcal{H}^d$ produces four halfedges in $\mathcal{H}^{d+1}$ indexed by

$$\begin{aligned} h &\mapsto 4h+0 \\ &\mapsto 4h+1 \\ &\mapsto 4h+2 \\ &\mapsto 4h+3. \end{aligned}$$

Note that it follows trivially that at depth d, the number of halfedges H_d produced by the subdivision is

$$H_d = 4^d H_0, \tag{2}$$

where H_0 denotes the number of halfedges of the control mesh. By convention, we choose that the four new halfedges form a new face in $\mathcal{H}^{d+1}$ and that the $(4h+0)$-th halfedge in $\mathcal{H}^{d+1}$ cuts the h-th halfedge in $\mathcal{H}^d$ in half; Figure 3 highlights this specific rule for the red and blue halfedges. From this particular construction, we determine refinement rules for the halfedge operators TWIN, NEXT, PREV, and FACE; Figure 3 (a, b, c, f) compiles these rules and provides some illustrated examples for the halfedges highlighted in red and blue. The remaining VERT and EDGE operators require additional conventions, and we focus on them next.

Edge Operator Catmull-Clark subdivision splits existing edges into two new ones. It also produces an extra number of H_d edges within each face of the input mesh. Therefore, the number of edges E_{d+1} produced by the subdivision follows the recurrence relation

$$E_{d+1} = 2E_d + H_d. \tag{3}$$

By convention, we choose that the e-th edge in $\mathcal{S}^d$ produces two edges in $\mathcal{S}^d$ indexed by

$$\begin{aligned} e &\mapsto 2e+0 \\ &\mapsto 2e+1. \end{aligned} \tag{4}$$

Figure 3 provides an illustration for this construction, where the green edge e_1, which is formed by the halfedge pair h_0 and h_4, splits into the green edges e_2 and e_3. We then label each of the H_d additional edges as $2E_d + h$ in such a way that each halfedge produces exactly one additional edge. From this particular construction we determine the refinement rule for the operator EDGE; Figure 3 (e) provides this specific rule. Note that this rule requires to evaluate the number of edges E_d. We retrieve this number by solving the recurrence relation of Equation (3). This gives

$$E_d = 2^{d-1}\left(2E_0 + (2^d - 1)H_0\right), \tag{5}$$

where E_0 denotes the number of edges of the control mesh $\mathcal{S}^0$.

Vertex Operator Catmull-Clark subdivision adds an extra vertex for each face and edge of the input mesh. Therefore, the number of vertices V_d produced by the subdivision at depth d follows the recurrence relation

$$\begin{cases} V_{d+1} &= V_d + F_d + E_d \\ F_{d+1} &= H_d \end{cases}, \tag{6}$$

where F_d denotes the number of faces at depth d. By convention, we label the new vertices as follows: we index each vertex produced

by the f-th face of $\mathcal{S}^d$ as $V_d + f$, and those produced by the e-th edge of $\mathcal{S}^d$ as $V_d + F_d + e$. Thanks to this particular construction we determine the refinement rule for the operator VERT; Figure 3 (d) provides this specific rule. Note that this rule requires to evaluate the number of faces F_d and vertices V_d. We retrieve these numbers by solving the recurrence relation of Equation (6) for $d \geq 1$

$$\begin{cases} V_d &= V_1 + 2^{d-1}\left(E_1 + (2^d - 1)F_1\right) \\ F_d &= 4^{d-1}H_0 \end{cases}. \tag{7}$$

Halfedge Refinement Algorithm Thanks to the refinement rules derived in this subsection, we straightforwardly compute the topologies $\mathcal{H}^1, \cdots, \mathcal{H}^D$; Algorithm 1 provides pseudocode for an implementation – note the simplicity of the actual pseudocode.

Algorithm 1 Halfedge refinement

```
1:  procedure RefineHalfedges(H^0: input, H^{1,...,D}: output)
2:      for all d ∈ [0, D) do
3:          for all h ∈ [0, H_d) do
4:              // apply rule Fig. 3 (a):
5:              Twin(H^{d+1}[4h+0]) ← 4 Next(Twin(H^d[h])) + 3
6:              Twin(H^{d+1}[4h+1]) ← 4 Next(H^d[h]) + 2
7:              Twin(H^{d+1}[4h+2]) ← 4 Prev(H^d[h]) + 1
8:              Twin(H^{d+1}[4h+3]) ← 4 Twin(Prev(H^d[h])) + 0
9:              // apply rules Fig. 3 (b, ···, f):
10:             ···
11:         end for
12:     end for
13: end procedure
```

4.2. Vertex Point Calculation

We now leverage $\mathcal{H}^0, \cdots, \mathcal{H}^D$ to compute the vertex points as defined in Section 3.1. Specifically, we compute the vertex points $\mathcal{V}^{d+1}$ using $\mathcal{V}^d$ and an iteration over the halfedges of $\mathcal{H}^d$. In this setting, each halfedge atomically adds a specific contribution to the computation. We describe these contributions in the following paragraphs.

Face Points Algorithm 2 provides pseudocode for an implementation of the face point calculation. The h-th halfedge weights the vertex-point that carries it by m, where m denotes the number of halfedges within the face FACE(h). Then, it adds this value inside the location of the new face point $i = V_d + \text{FACE}(h)$ (see Algorithm 2, lines 3-5). We determine the m value for the control mesh $\mathcal{S}^0$ by computing the length of the cycle formed by the halfedges. Note that for $d > 1$, we have $m = 4$ since the mesh is solely composed of quads.

Edge Points There are two ways of computing the new edge points depending on whether the edge is a boundary or not (see Section 3.1). We determine which specific way to follow while iterating over each halfedge h based on the sign of TWIN(h). The evaluation of the sharp edge point rule is trivial, so we focus here on the smooth edge point rule; Algorithm 3 provides pseudocode for an implementation. The h-th halfedge weights the sum of the vertex point that carries it and the new face point it produced using the face rule. It then adds this value at the location of the new edge point $i = V_d + F_d + \text{EDGE}(h)$ (see Algorithm 2, lines 3-6).

Vertex Points As for the edge points, there are two ways of computing the new vertex points. We determine which specific way to follow for the h-th halfedge based on whether the vertex that carries it lies on a boundary. The evaluation of the sharp vertex point rule is trivial, so we focus here on the smooth rule. Rather than directly evaluating Equation (1) through the halfedge iteration, we first rewrite it as the alternative form

$$-\frac{Q}{n} + \frac{4R'}{n} + \frac{S(n-3)}{n}, \tag{8}$$

where R' now denotes the average of the new edge points induced by all edges incident to the old vertex point; we are not aware of the existence of this particular expression in the literature so we believe it is new. Compared to the original expression, ours is more cache-friendly as it re-uses the newly computed face and edge points in $\mathcal{V}^{d+1}$, which are closer memory-wise to the new vertex point. Now, the h-th halfedge of the input mesh weights the sum of the vertex point that carries it with both the new face- and edge-point it produced. It then adds this value inside the location of the new vertex point $v = \text{VERT}(h)$ (see Algorithm 2, lines 3-7).

Algorithm 2 Face points (according to (1, A))

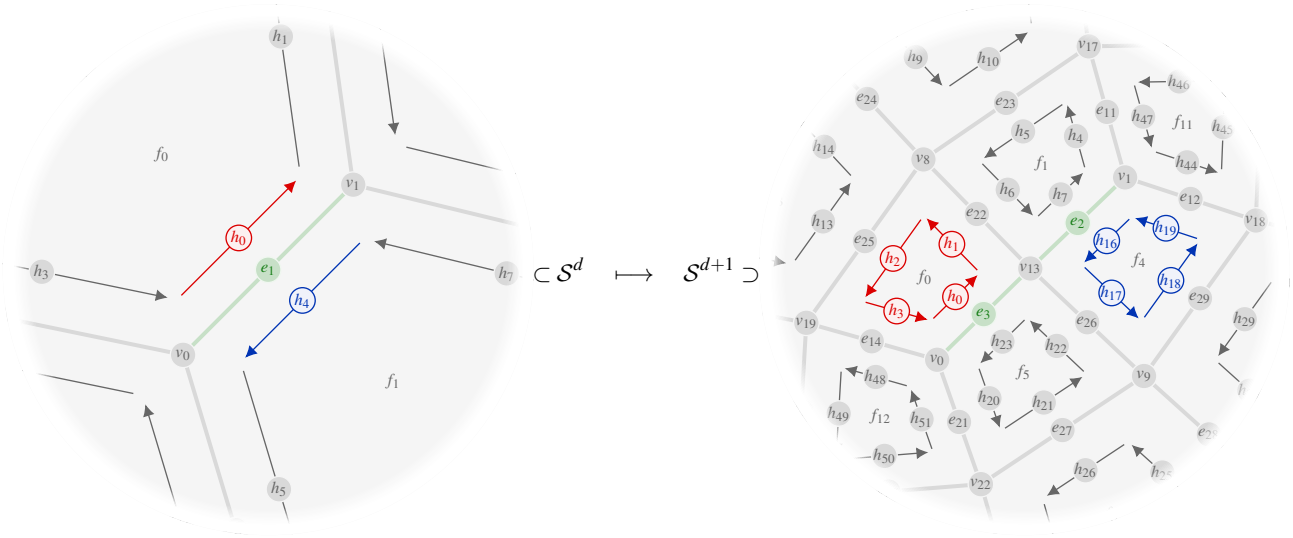
```
1: procedure FacePoints(S^d: input mesh, V^{d+1}: points)
2:     for all h ∈ [0, H_d) do
3:         m ← CycleLength(S^d, h)                    ▷ see Alg. 7
4:         v ← Vert(h)                                ▷ halfedge vertexID
5:         i ← V_d + Face(h)                          ▷ new face point vertexID
6:         V^{d+1}[i] ← V^{d+1}[i] + V^d[v] / m
7:     end for
8: end procedure
```

Algorithm 3 Smooth edge points (according to (1, B.2))

```
1: procedure EdgePoints(S^d: input mesh, V^{d+1}: points)
2:     for all h ∈ [0, H_d) do
3:         v ← Vert(h)                                ▷ halfedge vertexID
4:         i ← V_d + Face(h)                          ▷ new face point vertexID
5:         j ← V_d + F_d + Edge(h)                    ▷ new edge point vertexID
6:         V^{d+1}[j] ← V^{d+1}[j] + (V^d[v] + V^{d+1}[i]) / 4
7:     end for
8: end procedure
```

Algorithm 4 Smooth vertex points (according to (1, C.2))

```
1: procedure VertexPoints(S^d: input mesh, V^{d+1}: points)
2:     for all h ∈ [0, H_d) do
3:         n ← Valence(S^d, h)                        ▷ see Alg. 6
4:         v ← Vert(h)                                ▷ halfedge vertexID
5:         i ← V_d + Face(h)                          ▷ new face point vertexID
6:         j ← V_d + F_d + Edge(h)                    ▷ new edge point vertexID
7:         V^{d+1}[v] ← V^{d+1}[v] + (4V^{d+1}[j] − V^{d+1}[i] + (n−3)V^d[v]) / n^2
8:     end for
9: end procedure
```



Halfedge refinement rules

(a) halfedge's twin rule	(b) halfedge's next rule	(c) halfedge's previous rule
$\text{TWIN}(h) \mapsto \text{TWIN}(4h+0) = 4\,\text{NEXT}(\text{TWIN}(h)) + 3$ $\mapsto \text{TWIN}(4h+1) = 4\,\text{NEXT}(h) + 2$ $\mapsto \text{TWIN}(4h+2) = 4\,\text{PREV}(h) + 1$ $\mapsto \text{TWIN}(4h+3) = 4\,\text{TWIN}(\text{PREV}(h)) + 0$ *examples:* $\text{TWIN}(0) \mapsto \{23, 6, 13, 48\}$ $\text{TWIN}(4) \mapsto \{7, 22, 29, 44\}$	$\text{NEXT}(h) \mapsto \text{NEXT}(4h+0) = 4h+1$ $\mapsto \text{NEXT}(4h+1) = 4h+2$ $\mapsto \text{NEXT}(4h+2) = 4h+3$ $\mapsto \text{NEXT}(4h+3) = 4h+0$ *examples:* $\text{NEXT}(0) \mapsto \{1, 2, 3, 0\}$ $\text{NEXT}(4) \mapsto \{17, 18, 19, 16\}$	$\text{PREV}(h) \mapsto \text{PREV}(4h+0) = 4h+3$ $\mapsto \text{PREV}(4h+1) = 4h+0$ $\mapsto \text{PREV}(4h+2) = 4h+1$ $\mapsto \text{PREV}(4h+3) = 4h+2$ *examples:* $\text{PREV}(0) \mapsto \{3, 0, 1, 2\}$ $\text{PREV}(4) \mapsto \{19, 16, 17, 18\}$
(d) halfedge's vertex rule (*using* $h' := \text{PREV}(h)$)	(e) halfedge's edge rule (*using* $h' := \text{PREV}(h)$)	(f) halfedge's face rule
$\text{VERT}(h) \mapsto \text{VERT}(4h+0) = \text{VERT}(h)$ $\mapsto \text{VERT}(4h+1) = V_d + F_d + \text{EDGE}(h)$ $\mapsto \text{VERT}(4h+2) = V_d + \text{FACE}(h)$ $\mapsto \text{VERT}(4h+3) = V_d + F_d + \text{EDGE}(h')$ *examples:* $\text{VERT}(0) \mapsto \{0, 13, 8, 19\}$ $\text{VERT}(4) \mapsto \{1, 13, 9, 18\}$	$\text{EDGE}(h) \mapsto \text{EDGE}(4h+0) = \begin{cases} 2\,\text{EDGE}(h) & \text{if } h > \text{TWIN}(h) \\ 2\,\text{EDGE}(h)+1 & \text{otherwise} \end{cases}$ $\mapsto \text{EDGE}(4h+1) = 2E_d + h$ $\mapsto \text{EDGE}(4h+2) = 2E_d + h'$ $\mapsto \text{EDGE}(4h+3) = \begin{cases} 2\,\text{EDGE}(h')+1 & \text{if } h' > \text{TWIN}(h') \\ 2\,\text{EDGE}(h') & \text{otherwise} \end{cases}$ *examples:* $\text{EDGE}(0) \mapsto \{3, 22, 25, 14\}$ $\text{EDGE}(4) \mapsto \{2, 26, 29, 12\}$	$\text{FACE}(h) \mapsto \text{FACE}(4h+0) = h$ $\mapsto \text{FACE}(4h+1) = h$ $\mapsto \text{FACE}(4h+2) = h$ $\mapsto \text{FACE}(4h+3) = h$ *examples:* $\text{FACE}(0) \mapsto \{0, 0, 0, 0\}$ $\text{FACE}(4) \mapsto \{4, 4, 4, 4\}$

Crease refinement rules

(g) creases's sharpness rule	(h) crease's next rule (*using* $c' := \text{NEXT}(c)$)	(i) crease's previous rule (*using* $c' := \text{PREV}(c)$)
$\sigma(c) \mapsto \sigma(2c+0) = \langle \frac{\sigma(\text{PREV}(c)) + 3\sigma(c)}{4} - 1 \rangle$ $\mapsto \sigma(2c+1) = \langle \frac{\sigma(\text{NEXT}(c)) + 3\sigma(c)}{4} - 1 \rangle$ *example:* $\sigma(1) = 1.8 \mapsto \{0.8, 0.8\}$	$\text{NEXT}(c) \mapsto \text{NEXT}(2c+0) = 2c+1$ $\mapsto \text{NEXT}(2c+1) = \begin{cases} 2c' & \text{if } c' \neq c = \text{PREV}(c') \\ 2c'+1 & \text{otherwise} \end{cases}$ *example:* $\text{NEXT}(1) = 1 \mapsto \{3, 3\}$	$\text{PREV}(c) \mapsto \text{PREV}(2c+0) = \begin{cases} 2c'+1 & \text{if } c' \neq c = \text{NEXT}(c') \\ 2c' & \text{otherwise} \end{cases}$ $\mapsto \text{PREV}(2c+1) = 2c$ *example:* $\text{PREV}(1) = 1 \mapsto \{2, 2\}$

Figure 3: *Halfedge and crease refinement rules induced by Catmull Clark subdivision. A Catmull-Clark subdivision step splits each halfedge into 4 new ones and each crease into 2 new ones. The algebraic rules to compute the attributes of the new halfedges and creases are summarized inside the tables.*

4.3. Implementation Details

We now focus on a parallel implementation of the algorithms we have presented so far. In order to compute $\mathcal{S}^{D\geq 1}$ given $\mathcal{S}^0$, we proceed in two steps. First, we compute all the necessary topology information $\mathcal{H}^1,\cdots,\mathcal{H}^D$ using Algorithm (1). Second, we compute the vertex points $\mathcal{V}^1,\cdots,\mathcal{V}^D$ using Algorithms (2, 3, 4). We first describe how much memory our algorithm requires and how we allocate it in practice. Then, we describe our parallel CPU-based implementation, followed by our GPU-based one.

Memory Allocation Our implementation only requires memory for topology $\mathcal{H}^1,\cdots,\mathcal{H}^D$ and vertex points $\mathcal{V}^1,\cdots,\mathcal{V}^D$. Note that unlike the approaches of [PEO09] and [MWS*20], we do not require any extra temporary memory as our buffers provide all the necessary information to compute the subdivision. For the topology, we allocate a halfedge buffer that stores the operators TWIN, VERT, and EDGE as signed, 32-bit integers; we neglect the remaining operators NEXT, PREV, and FACE as we compute them analytically as shown in Figure 2 (b). Using Equation (2), we determine the size of the halfedge buffer as

$$\sum_{d=1}^{D} H_d = \frac{4^D-1}{3} H_1, \tag{9}$$

where $H_1 = 4H_0$ denotes the number of halfedges of $\mathcal{S}^1$. Note that we also use Equation (9) to determine the pointer to the d-th topology $\mathcal{H}^{d\leq D}$. For the vertex points, we allocate a vertex buffer that stores the 3D point coordinates as 32-bit floating point values. Using Equation (7), we determine the size of the vertex buffer as

$$\sum_{d=1}^{D} V_d = (2^D-1)(E_1-2F_1) + \frac{4^D-1}{3} F_1 + D(F_1-E_1+V_1). \tag{10}$$

Since our implementation increments the values of this buffer in parallel, it is important to set its values to zero. Note that we also use Equation (10) to determine the offset to the d-th vertex points $\mathcal{V}^{d\leq D}$. We believe we are the first to provide explicit memory footprint formulas, for computing Catmull Clark subdivision.

Parallel-for CPU Implementation Our CPU implementation is written in C and runs halfedge iterations as OpenMP parallel-for loops (see Algorithm (1) line 3 for halfedge refinement and Algorithms (2, 3, 4) line 2 for vertex point calculations). The vertex point calculations lead to concurrent memory accesses over the vertex buffers and so we use an OpenMP atomic instruction for the (single) memory write operation executed by each thread; for more details we refer the reader to the C code provided in our supplemental material. In order to assess our implementation's scalability across thread count, we provide performance measurements for the subdivision of the ArmorGuy asset; Figure 4 provides the results of our measurements using 1, 2, 4, and 8 concurrent threads of an AMD Ryzen Threadripper 3960X. As demonstrated by the plotted curves, the processing speed of our implementation increases linearly with the number of threads. It is interesting to note that for higher thread counts, we sometimes observed a decrease in scalability because memory bandwidth becomes the main bottleneck; we discuss memory bandwidth issues in more detail in Section 6 and provide exhaustive CPU performance measurements in a dedicated supplemental document.

GPU Implementation Thanks to the pointerless nature of our halfedge data-structure, our GPU implementation is a straightforward port of our CPU implementation to GLSL compute shaders. In practice we rely on core GLSL450 shaders augmented with the NVIDIA extension `GL_NV_shader_atomic_float`. We wrote two variants of four compute shaders that respectively implement the halfedge refinement, face points, edge points, and vertex points routines discussed so far. The first variant serves to compute subdivision level one and supports non-quad faces. The second variant serves to compute deeper subdivision levels and leverages the analytic nature of the NEXT, PREV, and FACE halfedge operators of quad-only meshes. In addition, we also rely on a dedicated kernel to set the vertex points to zero before refinement. In practice, this kernel takes the form of a simple OpenGL ClearBuffer command. We refer the reader to the shaders provided in supplemental for more details. In order to compute a subdivision down to an arbitrary depth, we successively call these shaders followed by a memory barrier operation. To quantify gains due to parallelization, we compared the performances of our GPU-based implementation against our CPU-based one; Figure 4 provides the results of this comparison. The reported numbers show an important speed-up over our CPU implementation thanks to the high processor counts and memory bandwidth offered by modern GPUs. Note that we perform a more thorough performance analysis in Section 6.

5. Semi-Sharp Creases

In this section, we extend our algorithms to support the semi-sharp crease extension of DeRose et al. [DKT98]. We start by recalling the modified refinement rules induced by the extension (Section 5.1). Next, we provide a refinement rule for creases that allows extending our algorithm to support semi-sharp creases (Section 5.2). Finally, we provide some implementation details regarding the additional memory required to support the extension (Section 5.3).

5.1. Background

Building upon the idea of Hoppe et al. [HDD*94], DeRose et al. tag the edges of the control mesh $\mathcal{S}^0$ with arbitrary sharpness values $\sigma \geq 0$. In turn, these sharpness values alter the vertex point calculations of the original Catmull-Clark subdivision. We provide below the modified rules induced by this extension.

Semi-sharp Creases Formalism The introduction of sharp edges leads to consider specific topological configurations. To this end, we denote $\bar{\sigma} \geq 0$ the average edge-sharpness incident to a vertex and refer to any edge with sharpness $\sigma > 0$ as a crease $c \in \mathcal{S}$ so that we write $\sigma(c) > 0$. Whenever $\sigma(c) \in [0,1)$, we say that the crease is a blending crease. Additionally, we refer to a vertex with valence $n = 2$ or with more than two incident creases as a corner vertex. In the case where the vertex has exactly two incident creases, we refer to it as either a creased vertex if $\bar{\sigma} \geq 1$, or a blended vertex otherwise. Finally, we mention that a crease $c \in \mathcal{S}$ may act as a link within a path of connected creases. In this case, we respectively denote its two neighbors $\text{PREV}(c) \in \mathcal{S}$ and $\text{NEXT}(c) \in \mathcal{S}$.

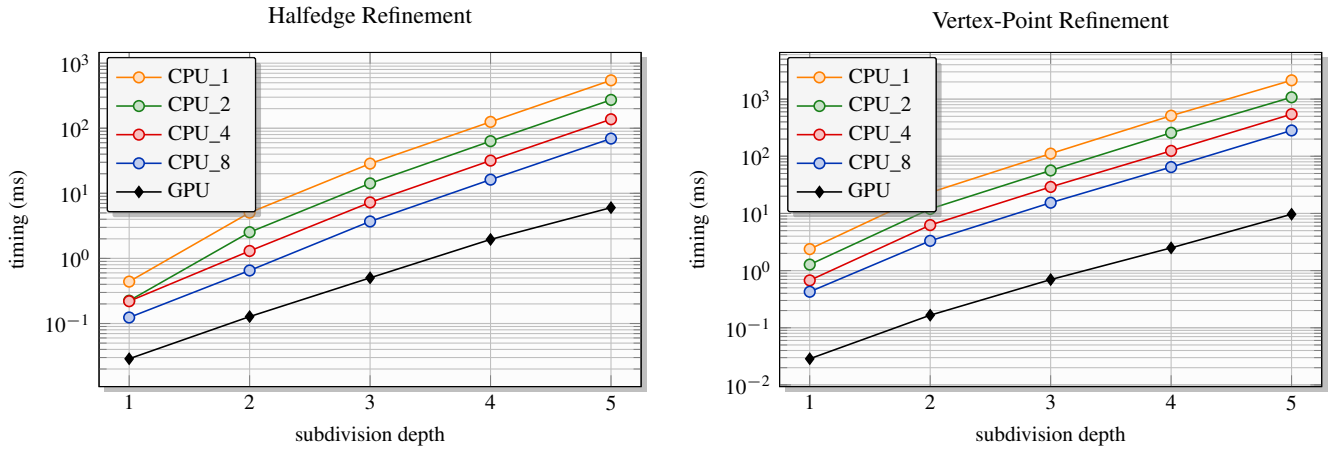


Figure 4: *Per-thread scalability for the computation of (left) halfedge-, and (right) vertex-point refinement as a function of depth with ArmorGuy as control cage. The CPU is an AMD Ryzen Threadripper 3960X (24-cores), and the GPU an NVIDIA RTX 2080.*

Vertex Point Calculation The following rules apply in addition to the previous ones (see Section 3.1):
(B.3) New crease points – same as rule (B.1)
(B.4) New blending crease points – the linear interpolation of point rules (B.1) and (B.2) with weight $\sigma \in [0,1)$
(C.3) New corner vertex points – same as rule (C.1)
(C.4) New blended vertex points – the linear interpolation of point rules (C.3) and (C.5) with weight $\bar{\sigma} \in [0,1)$
(C.5) New creased vertex points – the average

$$\frac{A+6S+B}{8}, \tag{11}$$

where
A = the vertex point that forms the first crease incident to the old vertex point S.
B = the vertex point that forms the second crease incident to the old vertex point S.

Topological Rules The rules for edge and face construction of the new mesh $\mathcal{S}^{d+1}$, remain unchanged. The creases of the new mesh $\mathcal{C}^{d+1} \subset \mathcal{S}^{d+1}$ are determined according to the curve subdivision algorithm of Chaikin [Cha74]: Each crease c creates two new creases with sharpness values

$$\sigma_1 = \langle \frac{\sigma(\textsc{Prev}(c)) + 3\sigma(c)}{4} - 1 \rangle,$$
$$\sigma_2 = \langle \frac{\sigma(\textsc{Next}(c)) + 3\sigma(c)}{4} - 1 \rangle,$$

where we use the notation $\langle x \rangle := \max(0, x)$. It follows trivially that the resulting number of creases C_{d+1} follows the recurrence relation

$$C_{d+1} = 2C_d, \tag{12}$$

where C_d denotes the number of creases of the mesh $\mathcal{S}^d$.

5.2. Refinement Rules

The semi-sharp crease extension refines creases into two new ones. We leverage this property to derive a breadth-first algorithm based on a refinement rule for the creases $\mathcal{C}^d$ of an input mesh $\mathcal{S}^d$ in the spirit of the one we used for the halfedges $\mathcal{H}^d$. This allows us to compute all $\mathcal{C}^1, \cdots, \mathcal{C}^D$. In turn, we use this information to evaluate the modified vertex-point rules for $\mathcal{V}^1, \cdots, \mathcal{V}^D$. Note that we avoid the method of Nießner et al. [NLG12] as it requires isolated creases, which is inapplicable in the general case without preprocessing the input mesh first.

Crease refinement As for halfedges, we represent each crease through its fundamental operators σ, NEXT, and PREV. By convention, we label each crease according to the edge-labelling scheme introduced in Equation (4). Thanks to this approach, we determine the refinement crease rules; Figure 3 (g, h, i) compiles these rules and provides an illustration for the edge highlighted in green. Algorithm 5 provides pseudocode for an implementation.

Vertex Points As for regular Catmull-Clark subdivision, we still iterate over the halfedges $\mathcal{H}^d$ of the input mesh to compute the vertex points $\mathcal{V}^{d+1}$. In order to determine which of the (A, B, C) rules apply for the h-th halfedge, we lookup its associated sharpness value $\sigma(\textsc{Edge}(h))$; Algorithm 8 provides pseudocode for an actual implementation. Thanks to this construction, we apply the proper rules for face points, edge points and vertex points. Note that for the creased vertex rule (C.5), we use an alternative form of Equation (11). Specifically,

$$\frac{A'+2S+B'}{4}, \tag{13}$$

where A' and B' respectively denote the new crease points produced by the old creases incident to S. As for Equation (8), Equation (13) allows to re-use the newly-computed neighboring vertex points.

Algorithm 5 Crease refinement

```
1: procedure RefineCreases(C^0: input, C^{1,...,D}: output)
2:     for all d ∈ [0, D) do
3:         for all c ∈ [0, 2^d E_0) do
4:             // apply rule Fig. 3 (g):
5:             σ(C^{d+1}[2c+0]) ← ⟨(σ(Prev(C^d[c])) + 3σ(C^d[c]))/4 − 1⟩
6:             σ(C^{d+1}[2c+1]) ← ⟨(σ(Next(C^d[c])) + 3σ(C^d[c]))/4 − 1⟩
7:             // apply rules Fig. 3 (h, i):
8:             ...
9:         end for
10:    end for
11: end procedure
```

5.3. Implementation Details

Our extended algorithm for semi-sharp creases remains mostly unchanged from the original one. As such, its implementation is straightforward. For the sake of completeness, we discuss here how much memory requires our extended implementation and how we allocate it in practice.

Memory Allocation In order to support semi-sharp creases, our implementation requires memory for the creases $\mathcal{C}^1, \cdots, \mathcal{C}^D$. For this we allocate memory for a crease buffer that stores the operators σ, NEXT, PREV as a 32-bit floating point, and two 32-bit signed integers, respectively. The crease buffer has size

$$\sum_{d=1}^{D} C_d = 2C_0\,(2^D - 1), \tag{14}$$

where $C_0 = E_0$ denotes the number of edges of $\mathcal{S}^0$. Note that we also use Equation (14) to determine the pointer to the d-th crease buffer $\mathcal{C}^{d \le D}$. As for our original implementation, our extension for semi-sharp creases has tractable memory requirements.

6. Performance Evaluation

With the exposition of our subdivision algorithm complete, we now turn to its evaluation against existing parallel implementations. To this end, we provide performance comparisons of our implementation against publicly available implementations.

Methodology In order to position our implementation with respect to previous work, we provide performance-measurement comparisons against publicly available implementations. We conduct our performance-measurements under the two following scenarios:

- Interactive modeling: we compute both the topology and the vertex points. This scenario is relevant for interactive modelling, when the topology and/or the creases of the control mesh are modified.
- Vertex point update: we only compute the vertex points using precomputed topology information. This scenario is relevant for rendering, e.g., skinned meshes, which maintains control mesh topology constant.

In both experiments, we report the timing for computing subdivisions on a variety of control meshes shown in Figure 5. Each control mesh exhibits different features such as semi-sharp creases, boundaries, and/or high numbers of non-quad faces. Each timing we report represents the median of the delta between the first and last operation required for the computation over 50 runs. Our timings thus naturally include shader/kernel execution time, memset instructions, state changes, and CPU-GPU synchronizations. Note that we provide timings for the tessellation-shader based implementation of [NLMD12]. For this specific algorithm, we measure the total mesh rendering time when moved offscreen (so as to minimize rasterization overhead). We use the GLSL compute shader backend for OpenSubdiv. We compile the timings for computing up to 6 subdivision levels in our supplemental material. In the following paragraphs, we focus on the measurements for subdivision level 4, which corresponds to a practical setting for both modelling and rendering in production.

	[MWS*20]	Ours
Bigguy	19.8 MiB	28.2 MiB
Monsterfrog	17.7 MiB	25.1 MiB
Imrod	73.4 MiB	104.1MiB
T-Rex	155.1 MiB	220.0MiB

Table 1: *Memory requirements for subdivision down to level 4.*

A Comment on Semi-sharp Crease Support Among all the publicly available open-source implementations we tested, we found our implementation to be the only one to support semi-sharp creases in addition to that of OpenSubdiv. Therefore, we separated creased-mesh performances from the other ones for fair comparisons. Note that Nießner et al. [NLMD12] and Mlakar et al. [MWS*20] claim support for semi-sharp creases in their respective papers, but we were unable to reproduce such results using their source code. Since our implementation is publicly available and provides straightforward support for semi-sharp creases, we position ourselves as the only practical alternative to OpenSubdiv.

Interactive Modeling Figure 7 compiles the timings for the interactive modeling scenario for subdivision depth 4. As demonstrated by the reported numbers, our method runs on par with the state-of-the-art method of Mlakar et al. [MWS*20] for non-creased assets and significantly outperforms OpenSubdiv for creased assets. The slow performances for both OpenSubdiv and Nießner et al. [NLMD12] are due to the preprocessing stage they require, which is purely sequential. Note that the performance numbers for [PEO09] are missing for the Imrod and T-Rex assets as the implementation lacks support for non-quad faces.

Vertex Point Update Figure 8 compiles the timings for the vertex point update scenario for subdivision depth 4. As demonstrated by the reported numbers, our method again performs on par with state-of-the-art methods. Note that OpenSubdiv outperforms our method only once for the ArmorGuy asset. We explain this result by the fact that this particular asset forces OpenSubdiv's to significantly pre-subdivide it around its features, leaving only very little geometry to process.

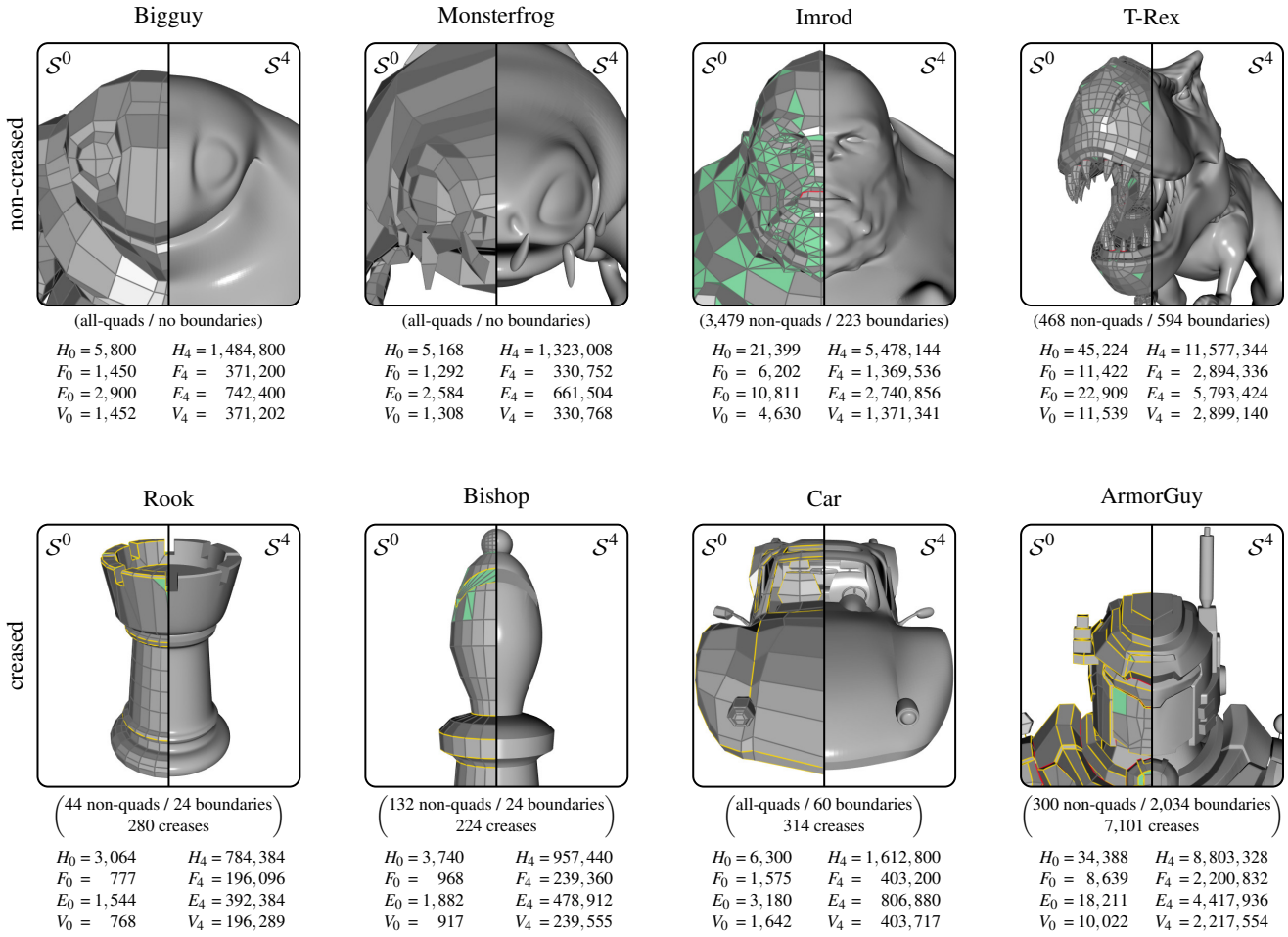


Figure 5: *The subdivision surfaces we used for performance measurements along with their properties. Non-quad faces, boundaries, and semi-sharp creases are respectively highlighted in green, magenta, and yellow.*

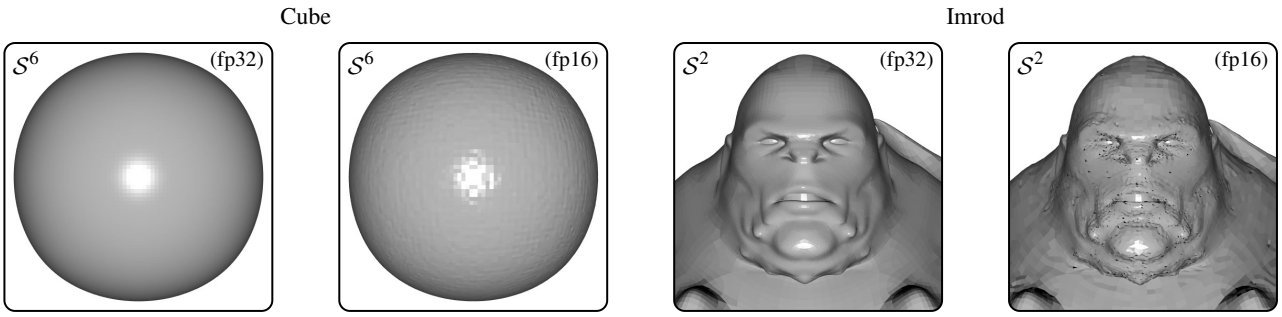


Figure 6: *Discretization artifacts due to 16-bit floating point precision compared to 32-bit floating point precision.*

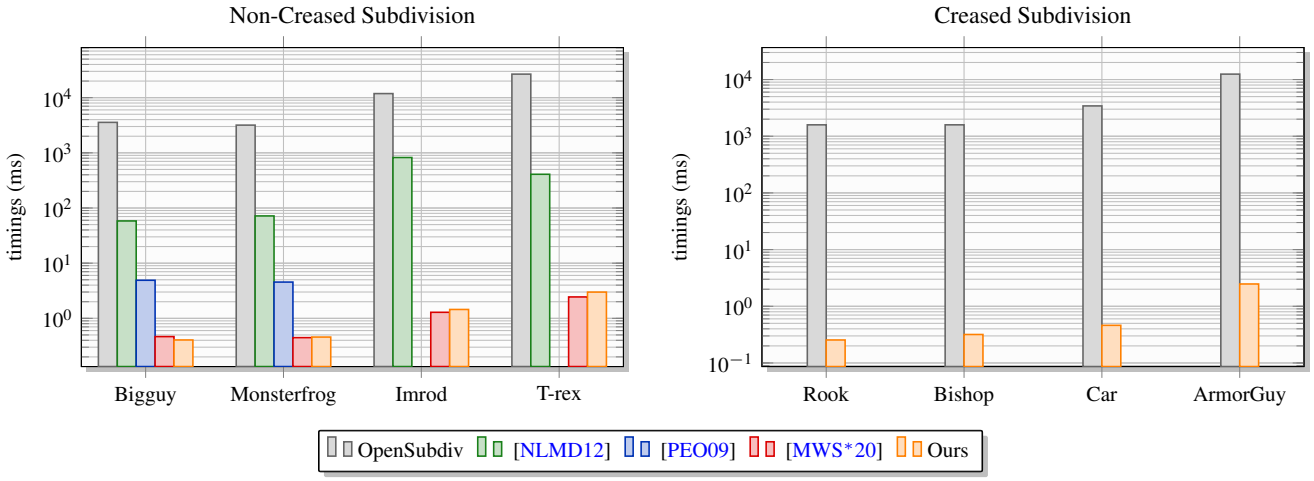


Figure 7: *End-to-end subdivision timings for subdivision down to level 4.*

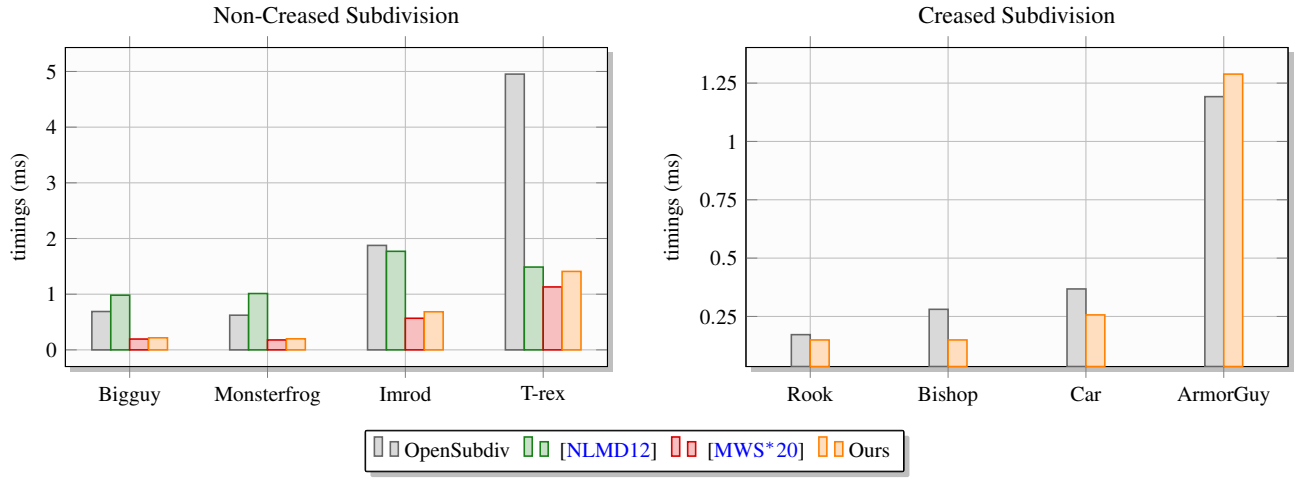


Figure 8: *Vertex point computation timings for subdivision down to level 4.*

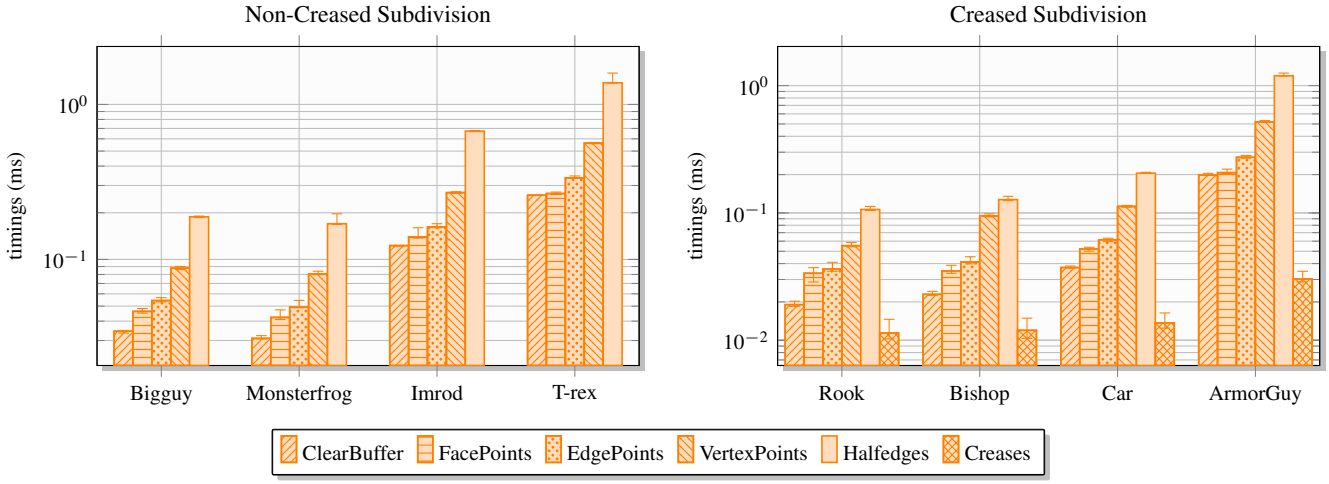


Figure 9: *Per-Kernel computation timings for subdivision down to level 4.*

Per-Kernel Timings For the sake of completeness we report timings for each GPU kernel executed by our implementation in Figure 9. The error bars shown in the plot correspond to the minimum and maximum timings measured over our 50 runs and are shown to emphasize performance stability. In practice, the halfedge kernel is consistently the slowest one, followed by the vertex-point kernel. Conversely, the crease kernel is negligible with respect to the other kernels. This opens up the possibility for real-time crease (in addition to vertex points) animation, which is not currently supported by OpenSubdiv as creases require heavy preprocessing.

Memory Consumption The memory requirements of our algorithm are straightforward to assess thanks to Equations (9, 10). For instance, computing the subdivision down to level 4 of the Monsterfrog asset given its control mesh statistics shown in Figure 5 requires 1,757,120 halfedges (see Equation (9)) and 439,344 vertex points (see Equation (10)). Since we store each halfedge as three 32-bit integers and each vertex point as three 32-bit floats, we thus require a total of 25.1 MiBytes. We compare our memory requirements with those of Mlakar *et al.* [MWS*20] in Table 1. The reported numbers demonstrate that their memory consumption is 70% that of ours, which makes their method more efficient in terms of memory. We add that for creased meshes, our implementation requires an additional crease buffer of size given by Equation (14) and consists of one 32-bit floating point and two 32-bit integers as discussed in Section 5.3.

Memory Bandwidth as Performance Bottleneck We observed negligible performance gains when disabling crease support even though this incurs non-negligible shader logic simplifications. This is due to the fact that our implementation is primarily bottlenecked by memory bandwidth. In order to emphasize this fact, we conducted the following experiment: We ran our vertex-point kernels at 16-bit floating point precision (rather than 32-bit) using another GLSL extension by NVIDIA namely `GL_NV_shader_atomic_fp16_vector`. This extension reduces the memory footprint of vertex points from $3 \times 4 = 12$ Bytes to $4 \times 2 = 8$ Bytes (the extension forces 4-component vectors), and consistently results in speedups ranging from $\times 1.15$ to $\times 1.25$ across our test meshes. In practice however, 16-bit floating points produce obvious discretization artifacts illustrated in Figure 6. While 16-bit floating points thus turn out to be impractical, they demonstrate the benefits of compressed vertex-point formats for faster subdivision. An interesting addition from hardware vendors could be to provide a similar GLSL extension for, e.g., 16-bit normalized coordinates. Such an extension would also be beneficial for more general compute tasks and image load/store operations.

7. Conclusion

We introduced a novel parallel algorithm suitable for computing Catmull-Clark subdivision using a halfedge mesh data-structure. Our algorithm yields state-of-the-art performances and its implementation is straightforward. In future work, we plan to derive the halfedge refinement rules induced by other subdivision schemes such as Loop. We are also investigating a gather-based implementation to support GPUs that lack support for atomic float operations.

Acknowledgements

We credit Bay Raitt for the Bigguy and Monsterfrog models, and Dmitry Parkin for the Imrod model. The T-Rex asset was modeled by Olivier Roos for Studio Manette. The ArmorGuy model is courtesy of ©2014 DigitalFish, Inc. Finally, the Rook, Bishop and Car assets are from OpenSubdiv. We thank Anjul Patney, Daniel Mlakar, and Lionel Untereiner for answering technical questions on parallel subdivision. Lastly, we thank our colleagues Laurent Belcour, Eric Heitz, Frédéric Larue, Mathieu Muller and Cyril Jover. Eric, Laurent, and Frédéric proofread drafts of the paper, and Mathieu and Cyril put us in touch with Studio Manette.

References

[BFK*16] BRAINERD W., FOLEY T., KRAEMER M., MORETON H., NIESSNER M.: Efficient gpu rendering of subdivision surfaces using adaptive quadtrees. URL: https://doi.org/10.1145/2897824.2925874, doi:10.1145/2897824.2925874. 2, 3

[BKP*10] BOTSCH M., KOBBELT L., PAULY M., ALLIEZ P., LÉVY B.: *Polygon Mesh Processing*. AK Peters / CRC Press, Sept. 2010. URL: https://hal.inria.fr/inria-00538098. 4

[BS02] BOLZ J., SCHRÖDER P.: Rapid evaluation of catmull-clark subdivision surfaces. Web3D '02, Association for Computing Machinery, pp. 11–17. URL: https://doi.org/10.1145/504502.504505, doi:10.1145/504502.504505. 2

[Bun05] BUNNELL M.: Adaptive tessellation of subdivision surfaces with displacement mapping. In *GPU Gems 2*, Pharr M., (Ed.). Addison-Wesley, 2005, pp. 109–122. 2

[Cas12] CASHMAN T. J.: Beyond catmull-clark? a survey of advances in subdivision surface methods. *Computer Graphics Forum 31*, 1 (2012), 42–61. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2011.02083.x, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2011.02083.x, doi:https://doi.org/10.1111/j.1467-8659.2011.02083.x. 2

[CC78] CATMULL E., CLARK J.: Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer-Aided Design 10*, 6 (1978), 350 – 355. URL: http://www.sciencedirect.com/science/article/pii/0010448578901100, doi:https://doi.org/10.1016/0010-4485(78)90110-0. 2

[Cha74] CHAIKIN G. M.: An algorithm for high-speed curve generation. *Computer Graphics and Image Processing 3*, 4 (1974), 346 – 349. URL: http://www.sciencedirect.com/science/article/pii/0146664X74900288, doi:https://doi.org/10.1016/0146-664X(74)90028-8. 9

[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. doi:10.1080/10867651.1998.10487494. 5

[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, Association for Computing Machinery, pp. 85–94. URL: https://doi.org/10.1145/280814.280826, doi:10.1145/280814.280826. 2, 8

[HDD*94] HOPPE H., DEROSE T., DUCHAMP T., HALSTEAD M., JIN H., MCDONALD J., SCHWEITZER J., STUETZLE W.: Piecewise smooth surface reconstruction. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1994), SIGGRAPH '94, Association for Computing Machinery, pp. 295–302. URL: https://doi.org/10.1145/192161.192233, doi:10.1145/192161.192233. 8

[Ket99] KETTNER L.: Using generic programming for designing a data structure for polyhedral surfaces. *Computational Geometry 13*,

1 (1999), 65 – 90. URL: http://www.sciencedirect.com/science/article/pii/S0925772199000073, doi:https://doi.org/10.1016/S0925-7721(99)00007-3. 4

[KMDZ09] KOVACS D., MITCHELL J., DRONE S., ZORIN D.: Real-time creased approximate subdivision surfaces. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2009), I3D '09, Association for Computing Machinery, pp. 155–160. URL: https://doi.org/10.1145/1507149.1507174, doi:10.1145/1507149.1507174. 3

[KUJ*14] KRAEMER P., UNTEREINER L., JUND T., THERY S., CAZIER D.: Cgogn: n-dimensional meshes with combinatorial maps. In *Proceedings of the 22nd International Meshing Roundtable* (Cham, 2014), Sarrate J., Staten M., (Eds.), Springer International Publishing, pp. 485–503. 2

[LS08] LOOP C., SCHAEFER S.: Approximating catmull-clark subdivision surfaces with bicubic patches. *ACM Trans. Graph. 27*, 1 (Mar. 2008). URL: https://doi.org/10.1145/1330511.1330519, doi:10.1145/1330511.1330519. 3

[LSNCn09] LOOP C., SCHAEFER S., NI T., CASTAÑO I.: Approximating subdivision surfaces with gregory patches for hardware tessellation. 1–9. URL: https://doi.org/10.1145/1618452.1618497, doi:10.1145/1618452.1618497. 3

[MWS*20] MLAKAR D., WINTER M., STADLBAUER P., SEIDEL H.-P., STEINBERGER M., ZAYER R.: Subdivision-specialized linear algebra kernels for static and dynamic mesh connectivity on the gpu. *Computer Graphics Forum 39*, 2 (2020), 335–349. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13934, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13934, doi:10.1111/cgf.13934. 2, 3, 8, 10, 12, 13

[NLG12] NIESSNER M., LOOP C., GREINER G.: Efficient Evaluation of Semi-Smooth Creases in Catmull-Clark Subdivision Surfaces. In *Eurographics 2012 - Short Papers* (2012), Andujar C., Puppo E., (Eds.), The Eurographics Association. doi:10.2312/conf/EG2012/short/041-044. 3, 9

[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). URL: https://doi.org/10.1145/2077341.2077347, doi:10.1145/2077341.2077347. 2, 3, 10, 12

[PEO09] PATNEY A., EBEIDA M. S., OWENS J. D.: Parallel view-dependent tessellation of catmull-clark subdivision surfaces. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, Association for Computing Machinery, pp. 99–108. URL: https://doi.org/10.1145/1572769.1572785, doi:10.1145/1572769.1572785. 8, 10, 12

[PO08] PATNEY A., OWENS J. D.: Real-time reyes-style adaptive surface subdivision. *ACM Trans. Graph. 27*, 5 (Dec. 2008). URL: https://doi.org/10.1145/1409060.1409096, doi:10.1145/1409060.1409096. 2

[SB20] SIEGER D., BOTSCH M.: The polygon mesh processing library, 2020. http://www.pmp-library.org. 2

[SC*19] SHARP N., CRANE K., ET AL.: geometry-central, 2019. www.geometry-central.net. 2

[SJP05] SHIUE L.-J., JONES I., PETERS J.: A realtime gpu subdivision kernel. *ACM Trans. Graph. 24*, 3 (July 2005), 1010–1015. URL: https://doi.org/10.1145/1073204.1073304, doi:10.1145/1073204.1073304. 2

[SRK*15] SCHÄFER H., RAAB J., KEINERT B., MEYER M., STAMMINGER M., NIESSNER M.: Dynamic feature-adaptive subdivision. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2015), i3D '15, Association for Computing Machinery, pp. 31–38. URL: https://doi.org/10.1145/2699276.2699282, doi:10.1145/2699276.2699282. 3

[Sta98] STAM J.: Exact evaluation of catmull-clark subdivision surfaces at arbitrary parameter values. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1998), SIGGRAPH '98, Association for Computing Machinery, pp. 395–404. URL: https://doi.org/10.1145/280814.280945, doi:10.1145/280814.280945. 3

[The21] THE CGAL PROJECT: *CGAL User and Reference Manual*, 5.2.1 ed. CGAL Editorial Board, 2021. URL: https://doc.cgal.org/5.2.1/Manual/packages.html. 2

[Wei85] WEILER K.: Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Applications 5*, 1 (1985), 21–40. doi:10.1109/MCG.1985.276271. 4

[ZHR*09] ZHOU K., HOU Q., REN Z., GONG M., SUN X., GUO B.: Renderants: Interactive reyes rendering on gpus. *ACM Trans. Graph. 28*, 5 (Dec. 2009), 1–11. URL: https://doi.org/10.1145/1618452.1618501, doi:10.1145/1618452.1618501. 2

[ZSS17] ZAYER R., STEINBERGER M., SEIDEL H.-P.: A gpu-adapted structure for unstructured grids. *Computer Graphics Forum 36*, 2 (2017), 495–507. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13144, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13144, doi:https://doi.org/10.1111/cgf.13144. 2

Appendix A: Supplemental Pseudocode

Algorithm 6 Vertex valence (non-boundary only)

```
function VALENCE(S^d: input, h: halfedgeID)
    n ← 1
    h' ← NEXT(TWIN(h))
    while h' ≠ h do
        n ← n + 1
        h' ← NEXT(TWIN(h'))
    end while
    return n
end function
```

Algorithm 7 Halfedge cycle length

```
function CYCLELENGTH(S^d: mesh, h: halfedgeID)
    m ← 1
    h' ← NEXT(h)
    while h' ≠ h do
        m ← m + 1
        h' ← NEXT(h')
    end while
    return m
end function
```

Algorithm 8 Halfedge sharpness

```
function SHARPNESS(S^d: input, C^d: input, h: halfedgeID)
    e ← EDGE(h)
    if e ≥ 2^d E_0 then
        return 0
    else
        return σ(C^d[e])
    end if
end function
```