A Halfedge Refinement Rule for Parallel Loop Subdivision

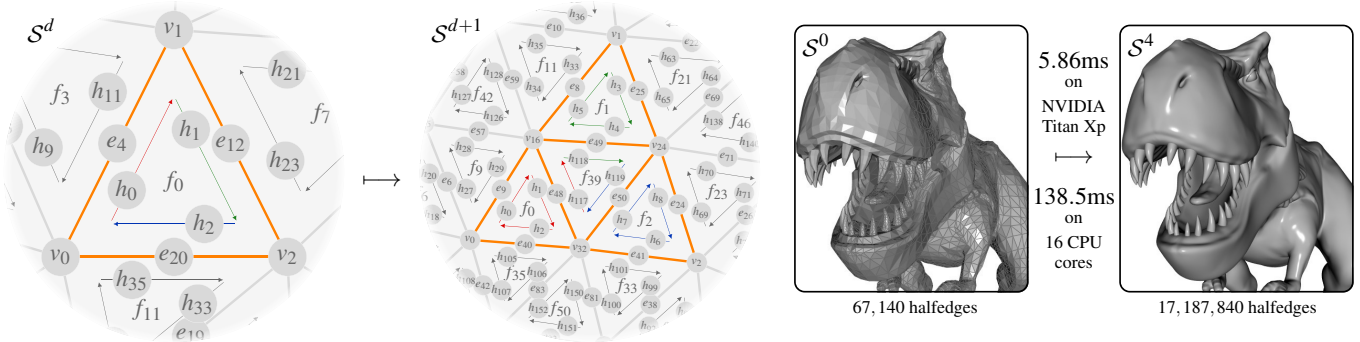K. Vanhoey [ID] and J. Dupuy [ID]

Unity Technologies

Figure 1: *We introduce a novel GPU-friendly parallel algorithm to compute Loop subdivision of production-ready assets in real-time (right). Our algorithm is based on the observation that the halfedges (arrows in $\mathcal{S}^d$, left) of the mesh under subdivision are invariably refined into four new ones (corresponding colors in $\mathcal{S}^{d+1}$, left). N.B.: in $\mathcal{S}^d$ (left), we have $H_d = 39$ halfedges, $V_d = 12$ vertices and $E_d = 24$ edges.*

Abstract

We observe that a Loop refinement step invariably splits halfedges into four new ones. We leverage this observation to formulate a breadth-first uniform Loop subdivision algorithm: Our algorithm iterates over halfedges to both generate the refined topological information and scatter contributions to the refined vertex points. Thanks to this formulation we limit concurrent data access, enabling straightforward and efficient parallelization on the GPU. We provide an open-source GPU implementation that runs at state-of-the-art performances and supports production-ready assets, including borders and semi-sharp creases.

CCS Concepts

• Computing methodologies → Computer graphics; Mesh models;

1. Introduction

Loop Subdivision for Interactive Modelling. Loop subdivision [Loo87] is a ubiquitous primitive supported by most interactive modelling tools through the industry-standard OpenSubdiv (OSD) library [Pix13]. The main advantage of OSD is that it achieves interactive rendering of complex geometry ($10M - 100M$ subdivided vertices) through GPU acceleration. However, these performances are only achieved under the assumption of static mesh topology. This naturally prevents interactive modelling experiences, where the user frequently edits mesh topology.

Limitations and Related Solutions. OSD is restricted to fixed topologies because it converts the input mesh into regular box-spline patches that can be evaluated in parallel [HFN*14]. Unfortunately, this takes several seconds to complete even for moderately-sized meshes ($100K - 1M$ vert.) and must be recomputed for each topology change. Recently, parallelization of the recursive subdivision rules was achieved for Catmull-Clark subdivision [MWS*20,

DV21], resulting in real-time performances for dynamic topologies. Yet, artists may create meshes specifically for Loop subdivision and these cannot easily be evaluated in real-time. The method of Mlakar *et al.* can be extended to Loop subdivision [MWS*20], but relies on a dedicated data-structure based on sparse matrices.

Contribution. Instead, we show how to generalize the simpler halfedge based method of Dupuy and Vanhoey [DV21] to Loop subdivision. We derive an invariant rule in which each halfedge splits into four new ones following our color-coding of Fig. 1 (arrows, left). Based on this observation, we provide a compact and complete set of definitions (Sec. 2) and algorithms (Sec. 3) for implementing Loop subdivision with semi-sharp crease support in parallel over halfedges. In Sec. 4, we show that this leads to an efficient algorithm that runs in real-time (Fig. 1, right) and greatly outperforms OSD for dynamic topologies, while still yielding similar performances for static ones. To our knowledge, our implementation is the first to lift uniform Loop subdivision to the class of subdivision schemes with real-time modelling performance.

2. Preliminaries

In this section, we provide the fundamental background to compute Loop subdivision using halfedges. In Sec. 2.1, we recall the Loop refinement rules as implemented in OpenSubdiv with support for semi-sharp creases. In Sec. 2.2, we describe the halfedge mesh data structure we leverage for our algorithm.

2.1. Loop Refinement

Loop subdivision applies the refinement rules defined by Loop [Loo87] on an input triangle mesh $\mathcal{S}^0$, yielding a denser mesh $\mathcal{S}^1$. Repeating this process D times yields denser triangle meshes $\mathcal{S}^1, \ldots, \mathcal{S}^D$, converging towards a smooth surface for higher D.

Topological Rules. The topological rule is illustrated in Fig. 1 (left, in orange): The new triangles of $\mathcal{S}^{d+1}$ originate from splitting each triangle of $\mathcal{S}^{d \geq 0}$ into four new ones.

Semi-Sharp Creases. Edges of the control mesh $\mathcal{S}^0$ can be tagged with arbitrary sharpness values $\sigma \geq 0$, which alters vertex point evaluation (see next paragraph). During subdivision, crease sharpness is decremented so that it starts smoothing after a number of subdivision levels. We introduce the following definitions (illustrated in our supplementary "cheat sheet", Sec. 1.1): A **smooth edge** is not a crease, *i.e.*, $\sigma = 0$. A **sharp crease edge** has $\sigma \geq 1$, and if $\sigma \in (0, 1)$, then it is a **blending crease edge**. Furthermore: a **smooth vertex** has zero or one adjacent crease, a **crease vertex** has exactly two adjacent creases, and a **corner vertex** has more than two adjacent creases. We define the vertex sharpness $\bar{\sigma}$ as the average of the sharpnesses of all its adjacent crease edges. Finally, a crease vertex having $\bar{\sigma} \in (0, 1)$ is called a **blending crease vertex**. Note that we follow Hoppe [HDD*94] in tagging all border edges as infinitely sharp creases to allow for boundary edge preservation.

Vertex Point Calculation. All vertices of the new triangle mesh $\mathcal{S}^{d+1}$ receive new attributes according to the following (E,V) rules:
(E.1) New creased edge points – the midpoint Q of the old edge,
(E.2) New smooth edge points – the weighted average $\frac{1}{4}(3Q+R)$,
(E.3) New blended crease edge points – the linear interpolation of point rules (E.1) and (E.2) with weight $\sigma \in (0, 1)$,
(V.1) New corner vertex points – the old vertex point V,
(V.2) New crease vertex points – the weighted average $\frac{1}{4}(3V+S)$,
(V.3) New smooth vertex points – the average $(1 - n\beta_n)V + \beta_n n \cdot T$,
(V.4) New blended vertex points – the linear interpolation of point rules (V.1) and (V.2) with weight $\bar{\sigma} \in (0, 1)$,
where (illustrations in our supplementary "cheat sheet", Sec 1.2):
V = the old vertex point,
R = the midpoint of the two vertex points in both incident triangles that are opposite to the edge,
S = the midpoint of the two vertex points that share a crease edge with V,
T = the average of the old neighboring vertices adjacent to V,
n = valence of the old vertex point, and
β_n = $\frac{1}{n}\left(\frac{5}{8} - \left(\frac{3}{8} + \frac{1}{4}\cos(\frac{2\pi}{n})\right)^2\right)$ is a function of n.

2.2. Halfedge Mesh Data-Structure

In this subsection we describe our memory representation. For more details, we provide source code as supplemental material.

Halfedge Buffer. To process Loop subdivision on a halfedge mesh, we require the following operators, as described by Dupuy and Vanhoey [DV21]: NEXT, PREV, FACE, TWIN, VERT and EDGE. To this end, we rely on the Directed Edges data structure [CKS98], which stores the halfedges that form a triangle sequentially in memory. In practice, we store each halfedge as three 32-bit integers which, respectively, store the result of the operators TWIN, VERT and EDGE. Since the halfedges forming a triangle are consecutive in memory, we retrieve the remaining operators analytically:

$$\text{NEXT}(h) = h \mod 3 == 2 \,?\, h-2 \,:\, h+1, \quad (1)$$

$$\text{PREV}(h) = h \mod 3 == 0 \,?\, h+2 \,:\, h-1, \quad (2)$$

$$\text{FACE}(h) = \lfloor h/3 \rfloor. \quad (3)$$

Note that we take care to preserve this halfedge contiguity for $\mathcal{S}^{d \geq 1}$ when deriving our halfedge refinement formulae in Sec. 3.

Vertex Buffer. We store the vertex points as three 32-bit floating point values in a vertex buffer. Each halfedge has access to its supporting vertex point through its VERT operator, which works as a regular index buffer for the vertex buffer.

Crease Buffer. We store creases as a 32-bit floating point sharpness value and two 32-bit integers. The former returns the result of the σ operator, while the remaining two return that of the successor crease NEXT and predecessor crease PREV.

3. Parallel Loop Subdivision

In this section, we present our halfedge-based loop refinement rule that leads to a parallel implementation. In our formalism, a mesh $\mathcal{S}^d$ is represented by a halfedge buffer $\mathcal{H}^d$ composed of H_d halfedges, a vertex buffer $\mathcal{V}^d$ of V_d vertices, and a crease buffer $\mathcal{C}^d$ of C_d creases. The input mesh $\mathcal{S}^0$ is refined into successive meshes $\mathcal{S}^1, \ldots, \mathcal{S}^D$ by D successive applications of Loop refinement. One refinement step yields $\mathcal{S}^{d+1}$ given $\mathcal{S}^d$, and comprises the steps of *halfedge refinement* ($\mathcal{H}^d \rightarrow \mathcal{H}^{d+1}$, Sec. 3.1), *crease refinement* ($\mathcal{C}^d \rightarrow \mathcal{C}^{d+1}$, Sec. 3.2) and *vertex refinement* ($\mathcal{V}^d \rightarrow \mathcal{V}^{d+1}$, Sec. 3.3).

3.1. Halfedge Refinement

Loop refinement splits each halfedge of a mesh into four: see the red, green and blue halfedges in Fig. 1 (left). So we write $H_{d+1} = 4H_d$, and it follows trivially that, at depth d, we have

$$H_d = 4^d H_0 \quad (4)$$

halfedges, where H_0 is the number of halfedges of the input mesh $\mathcal{S}^0$. Similarly, we have $F_{d+1} = 4F_d$, and it follows that $F_d = 4^d F_0$, where $F_0 = H_0/3$ is the number of faces of the input mesh. Unlike for Catmull-Clark subdivision [DV21], the four new halfedges do not form a single face. This thus requires a careful design of the halfedge numbering at subdivision, which we focus on next.

Refinement Rule. By convention, we choose to split the h-th halfedge in $\mathcal{H}^d$ into the four new halfedges in $\mathcal{H}^{d+1}$ indexed by

$$h \mapsto \{3h+0, 3h+1, 3h+2, 3H_d+h\}.$$

We arrange the halfedges in $\mathcal{H}^{d+1}$ as follows (see Fig. 1, left): The new $(3h+0)$-th halfedge cuts the old halfedge h in half and belongs to triangle number h, The new $(3h+1)$-th and $(3h+2)$-th halfedges

Algorithm 1 Halfedge refinement

1: **procedure** REFINEHALFEDGES($\mathcal{H}^d$: input, $\mathcal{H}^{d+1}$: output)
2:      **for all** $h \in [0, H_d)$ **do**
3:          $h', t, t' \leftarrow$ PREV(h), TWIN(h), TWIN(h')
4:          // Twin rules
5:          TWIN($\mathcal{H}^{d+1}[3h+0]$) $\leftarrow$ 3 NEXT(TWIN($\mathcal{H}^d[h]$)) + 2
6:          TWIN($\mathcal{H}^{d+1}[3h+1]$) $\leftarrow$ 3H_d + h
7:          TWIN($\mathcal{H}^{d+1}[3h+2]$) $\leftarrow$ 3 TWIN(PREV($\mathcal{H}^d[h]$))
8:          TWIN($\mathcal{H}^{d+1}[3H_d+h]$) $\leftarrow$ 3h + 1
9:          // Edge rules
10:         EDGE($\mathcal{H}^{d+1}[3h+0]$) $\leftarrow$ 2 EDGE(h) + ($h > t$? 0 : 1)
11:         EDGE($\mathcal{H}^{d+1}[3h+1]$) $\leftarrow$ 2E_d + h
12:         EDGE($\mathcal{H}^{d+1}[3h+2]$) $\leftarrow$ 2 EDGE(h') + ($h' > t'$? 1 : 0)
13:         EDGE($\mathcal{H}^{d+1}[3H_d+h]$) $\leftarrow$ 2E_d + h
14:         // Vert rules
15:         VERT($\mathcal{H}^{d+1}[3h+0]$) $\leftarrow$ $Vert(h)$
16:         VERT($\mathcal{H}^{d+1}[3h+1]$) $\leftarrow$ V_d + EDGE(h)
17:         VERT($\mathcal{H}^{d+1}[3h+2]$) $\leftarrow$ V_d + EDGE(PREV(h))
18:         VERT($\mathcal{H}^{d+1}[3H_d+h]$) $\leftarrow$ V_d + EDGE(PREV(h))
19:     **end for**
20: **end procedure**

follow within the same triangle, The new $(3H_d + h)$-th halfedge is the twin of the new $(3h+1)$-th halfedge. This construction preserves halfedge contiguity within faces, as required by Eqn. (1)–(3). It also induces an analytic formula for the subdivided halfedge's TWIN attribute, which we provide in Alg. 1 (lines 5–8).

Edge Operator. Loop subdivision splits existing edges in two, and adds three new edges per input triangle. It follows that the number of edges $E_{d+1} = 2E_d + 3F_d$, or equivalently using direct evaluation:

$$E_d = 2^d E_0 + 3(2^{2d-1} - 2^{d-1})F_0. \tag{5}$$

By convention, we choose that the e-th edge in $\mathcal{S}^d$ splits into two edges in $\mathcal{S}^{d+1}$ indexed by $e \mapsto \{2e+0, 2e+1\}$, and creates a new edge (contributing to the central triangle) labelled as $2E_d + h$. This way, each halfedge produces exactly one additional edge (see Fig. 1: *e.g.*, $h_0 \in \mathcal{S}^d$ in red generates $e_{48} \in \mathcal{S}^{d+1}$, $h_1 \in \mathcal{S}^d$ in green generates $e_{49} \in \mathcal{S}^{d+1}$, $h_2 \in \mathcal{S}^d$ in blue generates $e_{50} \in \mathcal{S}^{d+1}$). This construction yields the refinement rules of the halfedge's EDGE attribute, and we provide it in Alg. 1 (lines 10–13).

Vertex Operator. Loop subdivision adds an extra vertex for each edge of the input mesh. It follows that the number of vertices $V_{d+1} = V_d + E_d$, or equivalently:

$$V_d = V_0 + (2^d - 1)E_0 + (2^{2d-1} - 3 \cdot 2^{d-1} + 1)F_0. \tag{6}$$

By convention, we label the extra vertex produced by the e-th edge as $V_d + e$ (see Fig. 1: *e.g.*, $h_0 \in \mathcal{H}^d$ – spanning e_4 – produces $v_{16} \in \mathcal{S}^{d+1}$). This construction yields the refinement rules of the halfedge's VERT attribute, also provided in Alg. 1 (lines 15–18).

Halfedge Refinement Algorithm. Our algorithm for halfedge refinement (see Alg. 1) behaves similarly to Dupuy and Vanhoey's [DV21]: a breadth-first (*i.e.*, one depth level at a time) evaluation over halfedges of the halfedge refinement rules. As such, it is also trivial to parallelize via a parallel-for loop over the halfedges. We provide a detailed example in Sec. 2 of our supplementary "cheat sheet".

Algorithm 2 Vertex point calculation (Smooth points)

1: **procedure** SMOOTHPOINTS($\mathcal{S}^d$: input mesh, $\mathcal{V}^{d+1}$: points)
2:      $\mathcal{V}^{d+1} \leftarrow 0$ ▷ Initialize vertex buffer
3:      **for all** $h \in [0, H_d)$ **do**
4:          $v \leftarrow$ VERT(h)   ▷ halfedge vertexID
5:          $vn \leftarrow$ VERT(NEXT(h))   ▷ neighboring vertexID
6:          $vp \leftarrow$ VERT(PREV(h))   ▷ 3rd vertex in triangle
7:          $n \leftarrow$ VALENCE($\mathcal{S}^d$, h)
8:          $ve \leftarrow V_d +$ EDGE(h)   ▷ new edge point vertexID
9:          $\beta \leftarrow \frac{1}{n}\left(\frac{5}{8} - \left(\frac{3}{8} + \frac{1}{4}\cos\left(\frac{2\pi}{n}\right)\right)^2\right)$
10:         $\mathcal{V}^{d+1}[ve] \leftarrow \mathcal{V}^{d+1}[ve] + \frac{3\mathcal{V}^d[v] + \mathcal{V}^d[vp]}{8}$
11:         $\mathcal{V}^{d+1}[v] \leftarrow \mathcal{V}^{d+1}[v] + (\frac{1}{n} - \beta)\mathcal{V}^d[v] + \beta\mathcal{V}^d[vn]$
12:     **end for**
13: **end procedure**

3.2. Crease Refinement

A step of Loop subdivision splits each crease into two new ones covering the two edges the original one was split into. So we write $C_{d+1} = 2C_d$, and it follows trivially that, at depth d, we have $C_d = 2^d C_0$ creases, where $C_0 = E_0$ is the number of edges of the input mesh. The algorithm for refining the crease buffer is not Loop-specific: it is identical to that of Dupuy and Vanhoey [DV21] in which the attributes σ are decremented, and NEXT and PREV are updated consistently. As such, our implementation is identical too.

3.3. Vertex Point Calculation

There are two types of refined vertex points: the (new) edge points and the update of the (old) vertex points. We compute both using a single loop over halfedges where each halfedge scatters its contribution to a single edge-point and a single vertex-point. Since the calculation depends on the crease configurations (see Sec. 2.1), the resulting algorithm is quite long. Due to page-length constraints, we simply provide in Alg. 2 the smooth case (E.2 and V.3) and refer the interested reader to our supplementary source code. Notice how each halfedge additively scatters its contribution to a single edge point (line 10) and a single vertex point (line 11) following E.2 and V.3, respectively. Importantly, we mention that the vertex buffer $\mathcal{V}^{d+1}$ must be initialized to zero.

3.4. Implementation Details

Parallelization. We provide a CPU and a GPU implementation. We parallelize the computation over halfedges using C++ OpenMP parallel-for loops and GLSL Compute shaders, respectively. Alg. 2 leads to concurrent memory writes in lines 10 and 11. To handle race conditions, we respectively use an OpenMP `atomic` and the NVIDIA `GL_NV_shader_atomic_float` extension [DV21].

Memory Footprint. Our memory consumption can be simply evaluated with the following explicit formulae. Computing a subdivision down to depth D requires storing $\sum_{d=0}^{D} H_d = (4^{D+1} - 1)F_0$, and $\sum_{d=0}^{D} C_d = (2^{D+1} - 1)E_0$ elements for the halfedge and crease buffer entries, respectively. As for the vertex buffer, we allocate size for $\sum_{d=0}^{D} V_d = D \cdot V_0 + (2^{D+1} - D - 2)E_0 + (\frac{4^{D+1}-4}{6} - 3(2^D - 1) + D)F_0$ entries. All three values have to be multiplied by 12 bytes, as each entry of the buffers consists of three 4-byte integer or floating-point values. Tab. 1 provides some examples of memory consumption.

Asset	H_0	$E_0 = C_0$	F_0	V_0	GiB
Knight	1,902	951	634	365	0.14
BigguyT	8,700	4,350	2,900	1,452	0.63
T-rexT	67,140	33,867	22,380	11,539	4.83
ArmorGuyT	101,736	51,885	33,912	18,423	7.32

Table 1: *Memory requirements (GiB) for subdivision depth $D = 6$.*

4. Validation & Results

In this section, we position our method w.r.t. related work and demonstrate its effectiveness. We run our parallel implementation on four meshes and provide all data with a more thorough evaluation **in our supplemental documents**.

4.1. Scalability

In order to evaluate the parallel nature of our algorithm, we conducted the following experiment: We compute $\mathcal{S}^D$ given $\mathcal{S}^0$ using our algorithm and plot the performances for different numbers of processors in Fig. 2. As demonstrated by the reported numbers, performances scale proportionally to the number of CPU threads, and thus naturally benefit from GPU acceleration. We emphasize that the atomic operations we require per halfedge (Alg. 2, lines 10–11) have negligible impact on scalability. This is in part due to the fact that the number of concurrent writes are negligible: there are two writes for each edge point, and no more than $v \geqslant 2$ for each vertex point, where v denotes vertex valence.
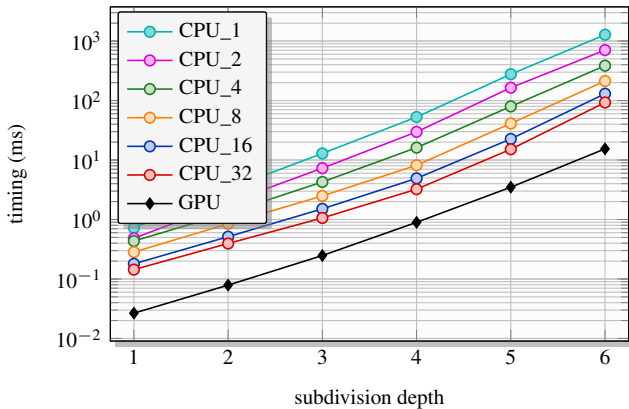

Figure 2: *Scalability w.r.t. thread count. End-to-end subdivision (halfedges, creases and vertex points) on the BigguyT model. Computed on an AMD Ryzen Threadripper 3960X CPU or NVIDIA Titan Xp GPU.*

4.2. Comparison

Positioning. Below, we compare performances with OSD, the only openly available implementation of fast Loop subdivision. Alternatives exist, but are limited. Stam [Sta99] relies on lookup tables that dismiss semi-sharp creases. Li *et al.* [LRZM11] trade quality for efficiency by computing an inexact limit surface. Mlakar *et al.* [MWS*20] propose a valid alternative: similar to our work, parallel breadth-first subdivision results in state of-the-art performances. Unfortunately, a fair comparison is impossible as

their method is complex to reproduce correctly and source code is unavailable. Parallel half-edge refinement was shown to be slightly slower than their method for Catmull-Clark subdivision [DV21]: it remains to be seen if this is similar for Loop refinement.

Performance Comparison. We compare to OSD exclusively on uniform subdivision and consider two scenarios: animation and modelling. The *animation* scenario assumes the topology of $\mathcal{S}^0$ to be static, and therefore $\mathcal{V}^D$ and $\mathcal{C}^D$ can be precomputed and fixed: only the vertex buffers $\mathcal{V}^{d \geqslant 1}$ are updated at runtime. In contrast, the *modelling* scenario requires to recompute all the buffers of $\mathcal{S}^{d \geqslant 1}$, which includes $\mathcal{H}^{d \geqslant 1}$, $\mathcal{C}^{d \geqslant 1}$ and $\mathcal{V}^{d \geqslant 1}$.

As demonstrated by Tab. 2, our method is roughly as efficient as OSD in the animation scenario. For the modelling scenario however, we obtain large speedups. This is because OSD relies on subdivision tables that have to be recomputed at each topology change [HFN*14], which is not parallelizable, as observed in previous works [MWS*20, DV21]. We emphasize that our implementation is fast enough for interactive modelling and rendering. Also note that OSD has a higher memory consumption: it runs out of memory for subdivision depths that our method handles well, as shown in our supplemental documents.

Modelling	Knight	BigguyT	T-rexT	ArmorGuyT
Ours	**0.75**ms	**3.58**ms	**25.58**ms	**38.07**ms
OSD	2.51s	13.26s	99.40s	124.91s
Animation				
Ours	0.43ms	**2.34**ms	**16.08**ms	23.83ms
OSD	**0.42**ms	2.73ms	17.45ms	**13.51**ms

Table 2: *Time for subdivision down to depth $D = 5$, computed on an Intel Core i5 4690K and an NVIDIA Titan Xp.*

References

[CKS98] CAMPAGNA S., KOBBELT L., SEIDEL H.-P.: Directed edges– a scalable representation for triangle meshes. *Journal of Graph. Tools 3*, 4 (1998), 1–11. doi:10.1080/10867651.1998.10487494. 2

[DV21] DUPUY J., VANHOEY K.: A Halfedge Refinement Rule for Parallel Catmull-Clark Subdivision. *Computer Graph. Forum 40*, 8 (2021). doi:10.1111/cgf.14381. 1, 2, 3, 4

[HDD*94] HOPPE H., DEROSE T., DUCHAMP T., HALSTEAD M., JIN H., MCDONALD J., SCHWEITZER J., STUETZLE W.: Piecewise smooth surface reconstruction. In *SIGGRAPH '94* (New York, NY, USA, 1994), ACM, pp. 295–302. doi:10.1145/192161.192233. 2

[HFN*14] HUANG Y., FENG J., NIESSNER M., CUI Y., YANG B.: Feature-adaptive rendering of loop subdivision surfaces on modern gpus. *J. Comput. Sci. Technol. 29*, 6 (2014), 1014–1025. doi:10.1007/s11390-014-1486-x. 1, 4

[Loo87] LOOP C.: *Smooth Subdivision Surfaces Based on Triangles*. PhD thesis, January 1987. 1, 2

[LRZM11] LI G., REN C., ZHANG J., MA W.: Approximation of loop subdivision surfaces for fast rendering. *IEEE Trans. on Visualization and Comp. Graphics 17*, 4 (2011). doi:10.1109/TVCG.2010.83. 4

[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 Graph. Forum 39*, 2 (2020). doi:10.1111/cgf.13934. 1, 4

[Pix13] PIXAR: Opensubdiv from research to industry adoption. In *ACM SIGGRAPH 2013 Courses* (New York, NY, USA, 2013), SIGGRAPH '13, ACM. doi:10.1145/2504435.2504451. 1

[Sta99] STAM J.: Evaluation of loop subdivision surfaces. *SIGGRAPH Course Notes* (1999). 4