Bisection Based Triangulation of Catmull Clark Subdivision

Jonathan Dupuy
Unity Technologies

Thomas Deliot
Unity Technologies

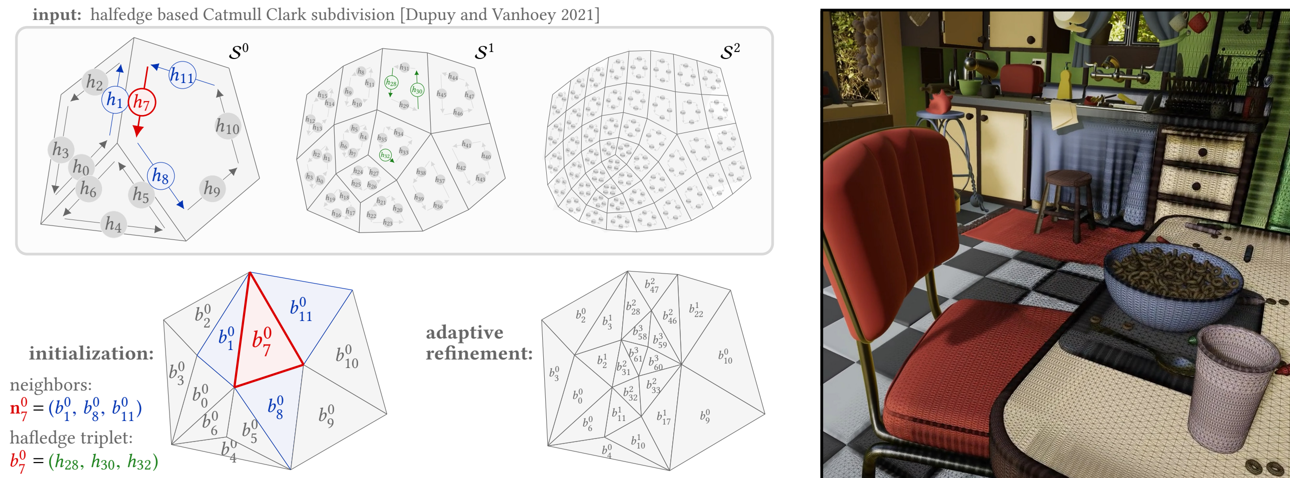


Figure 1: Overview of our triangulation method. Our method takes (top) a halfedge based subdivision as input and produces (bottom) adaptive bisection based triangulations by implicitly mapping bisectors to halfedge triplets. The resulting triangulation effectively acts as an anisotropic sampler over the subdivision, with the guarantee to produce a conforming topology.

ABSTRACT

Concurrent binary trees are a recent GPU-friendly data-structure suitable for generating bisection based terrain tessellations, i.e., adaptive triangulations over square domains. In this document, we introduce simple mappings and algorithms that bring such adaptive triangulations to Catmull Clark subdivision surfaces. Our resulting implementation is straightforward and allows to render densely triangulated subdivision surfaces in a few milliseconds on a modern GPU. We showcase production scenes rendered in real time within the Unity game engine thanks to our method.

KEYWORDS

GPU tessellation; compute shader; subdivision

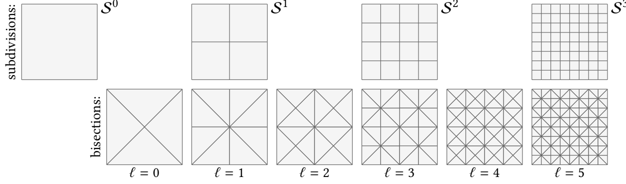
1 INTRODUCTION

Motivation. Many popular GPU rendering techniques require multiple passes over geometric data, e.g., shadow mapping, Z prepassing, path-tracing, visibility buffering, etc. Obviously, the speed of such techniques greatly depends on that of geometric data reads. In this context, tessellation shaders are not ideal: they either require re-generating up to thousands of triangles per pass or caching the vertex data it generates without the ability to index it (thus producing highly redundant data). To circumvent these issues, we introduced a compute based adaptive triangulation algorithm that provides efficient caching for terrain rendering, i.e., square domains [2020]. In this talk, we show how to extend these adaptive triangulations to geometric data produced by our recently published halfedge based Catmull Clark refinement algorithm [2021].

Contributions and Outline. Our approach is illustrated in Figure 1: We generate adaptive triangulations by refining what we refer to as bisectors, i.e., triangles that recursively split into two new ones. Each bisector references exactly three halfedges produced by our Catmull Clark algorithm. In turn, these halfedges provide access to vertex points, thus fully determining the geometry of its referent bisector. In this setting, the key to computing adaptive triangulations free of ⊤-junctions lies in the ability to determine the neighbors of each bisector at any refinement level. In the following sections, we provide an initialization method along with a simple refinement rule that fulfil this requirement (see Sections 2, 3). Furthermore, our method represents each bisector as an integer value, which makes it suitable for our GPU-friendly concurrent binary tree (CBT) data-structure (Section 4). We refer the interested reader to our supplemental video for real-time rendering results.

2 BISECTION OPERATORS

Bisector Initialization. Our triangulation method draws inspiration from that of Velho and Zorin [2001], who suggest initializing two bisectors per quad after one Catmull Clark refinement step. Our approach differs from theirs in that we initialize a bisector for each halfedge of the control cage. This construction provides two triangulations per Catmull-Clark subdivision as illustrated below:



For a control cage with $H_0 \geq 3$ halfedges, we index initial bisectors by $h \in [0, H_0)$ and set their halfedge reference to the following triplet (see red and green symbols in Figure 1):

$$b_h^0 = (4h,\ 4h + 2,\ 4\,\textsc{Next}(h)). \tag{1}$$

Bisector Refinement Rule. We refine bisectors depending on whether the current refinement level is even or odd. For even levels, we have:

$$\begin{aligned} b_k^\ell = (h_0,\ h_1,\ h_2) &\mapsto b_{2k+0}^{\ell+1} = (h_0,\ h_0 + 1,\ h_0 + 2) \\ &\mapsto b_{2k+1}^{\ell+1} = (h_2 + 2,\ h_2 + 3,\ h_2), \end{aligned} \tag{2}$$

and for odd bisection levels:

$$\begin{aligned} b_k^\ell = (h_0,\ h_1,\ h_2) &\mapsto b_{2k+0}^{\ell+1} = (4h_0,\ 4h_0 + 2,\ 4h_1) \\ &\mapsto b_{2k+1}^{\ell+1} = (4h_1,\ 4h_1 + 2,\ 4h_2). \end{aligned} \tag{3}$$

Bisector Vertex Points. We retrieve the bisector's vertex points by querying each halfedge it references. Given a bisector with halfedges h_0, h_1, and h_2, the vertex points are simply $\textsc{Vert}(h_0)$, $\textsc{Vert}(h_1)$, and $\textsc{Vert}(h_2)$.

3 ADAPTIVE TRIANGULATIONS

Conforming Triangulations. In order to produce adaptive triangulations, we simply refine bisectors non-uniformly. In this non-uniform setting, we guarantee conforming triangulations, i.e., free of ⊤-junctions, as follows: whenever we refine a bisector from refinement level ℓ to $\ell + 1$, we propagate the refinement over up to ℓ neighboring bisectors according to Algorithm 1. We determine the neighboring bisectors thanks to the equations derived next.

Neighboring Bisectors. The initial bisector with index $h \in [0, H_0)$ has the three following neighbors (see blue symbols in Figure 1):

$$\mathbf{n}_h^0 = (\textsc{Twin}(h),\ \textsc{Next}(h),\ \textsc{Prev}(h)). \tag{4}$$

Then, neighbors evolve according to the following refinement rule:

$$\begin{aligned} \mathbf{n}_b^\ell = (n_0,\ n_1,\ n_2) &\mapsto \mathbf{n}_{2b+0}^{\ell+1} = (2n_2 + 1,\ 2b + 1,\ 2n_0 + 1) \\ &\mapsto \mathbf{n}_{2b+1}^{\ell+1} = (2n_1 + 0,\ 2n_0 + 0,\ 2b + 0). \end{aligned} \tag{5}$$

Note that this rule applies to both even and odd refinement levels. Furthermore, our rule is such that the first neighbor is always the one for which refinement must be propagated to obtain conforming triangulations (see lines 4 to 6 in Algorithm 1).

Algorithm 1 Conforming triangulations via bisection

```
1:  procedure ConformingBisection(b^ℓ: bisector)
2:      Bisect(b^ℓ)
3:      n^ℓ ← Neighbors(b^ℓ)                    ▷ see Equation (5)
4:      while n_0^ℓ ≥ 0 and ℓ ≥ 0 do
5:          p^(ℓ−1) ← ParentBisector(n_0^ℓ)
6:          Bisect(p^(ℓ−1))
7:          Bisect(n_0^ℓ)
8:          n^ℓ ← Neighbors(p^(ℓ−1))
9:      end while
10: end procedure
```

4 CBT IMPLEMENTATION

Bisectors as Integers. We encode bisectors implicitly as unique integer values. Each integer value is a binary heap index: the most significant bit gives the bisector refinement level ℓ, and the following bits provide the path from the initial bisector to refinement level ℓ in most to least significant bit order. Specifically, the value for initial bisector $h \in [0, H_0)$ is

$$b_h^0 \mapsto 2^{\lceil \lg H_0 \rceil} + h. \tag{6}$$

After one refinement, the two children are twice the integer value plus zero and one respectively. This approach allows the use of a CBT to encode each triangle as a leaf node, as we describe next.

CBT Initialization. In order to encode all possible triangulations, we create a CBT with maximum depth $\lceil \lg H_0 \rceil + 2D - 1$, where D denotes the depth of the Catmull Clark subdivision. We then initialize all leaf nodes at depth $\lceil \lg H_0 \rceil$ to get a heap index per initial bisector. Note that this produces $2^{\lceil \lg H_0 \rceil} - H_0$ extra indexes that don't map to bisectors so we discard those in practice. This procedure effectively encodes all initial bisectors for the triangulation.

CBT Update and Rendering. To update the triangulation, we iterate over the heap indexes encoded by the CBT and decode them as bisectors. To determine the halfedge triplet of each bisector, we iterate over the bits of heap index and apply Equations (2, 3). We then retrieve the vertex points of the bisector and decide whether to refine or decimate it based on an arbitrary metric that controls the triangulation. We render each bisector in the same fashion.

Results and Performances. We implemented our method as a standalone C++ code and on top of our existing CBT implementation in the Unity game engine [Deliot et al. 2021]. On production assets, our triangulation takes a few milliseconds and adds negligible overhead compared to computing their Catmull-Clark subdivision see, e.g., the timings in our supplemental video.

REFERENCES

T. Deliot, Y. Xialing, J. Dupuy, and K. Rijnen. 2021. Experimenting With Concurrent Binary Trees for Large-scale Terrain Rendering. In *ACM SIGGRAPH 2021 Courses.*

Jonathan Dupuy. 2020. Concurrent Binary Trees (with Application to Longest Edge Bisection). *Proc. ACM Comput. Graph. Interact. Tech.* (2020). https://doi.org/10.1145/3406186

J. Dupuy and K. Vanhoey. 2021. A Halfedge Refinement Rule for Parallel Catmull-Clark Subdivision. *Computer Graphics Forum* (2021). https://doi.org/10.1111/cgf.14381

Luiz Velho and Denis Zorin. 2001. 4-8 Subdivision. *Comput. Aided Geom. Des.* 18, 5 (jun 2001), 397–427. https://doi.org/10.1016/S0167-8396(01)00039-5