Improved Triangle Encoding for Cached Adaptive Tessellation

Linus Horváth
linus.horvath@gmail.com
TU Wien

Bernhard Kerbl
kerbl@cg.tuwien.ac.at
TU Wien

Michael Wimmer
wimmer@cg.tuwien.ac.at
TU Wien

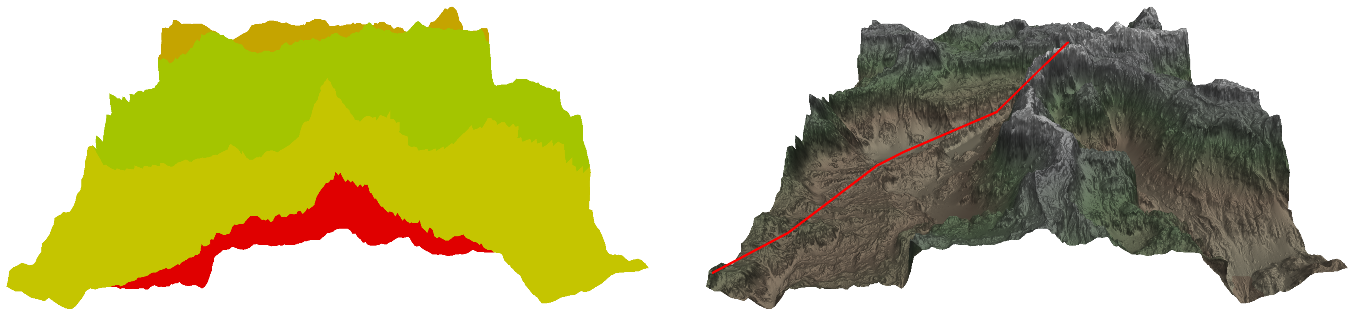

Figure 1: The terrain rendering setup by Khoury et al. which we use to evaluate our encoding. The input is a single quad consisting of two triangles. Individual geometry subdivision levels are adaptively generated over multiple frames (left). Our method preservers the visual results of the original method, but runs significantly faster on a simulated camera path (right).

CCS CONCEPTS

• **Computing methodologies** → **Rasterization**; **Mesh geometry models**; *Concurrent algorithms*; *Mesh models.*

KEYWORDS

real-time rendering, subdivision, adaptive tessellation, GPU

ACM Reference Format:
Linus Horváth, Bernhard Kerbl, and Michael Wimmer. 2020. Improved Triangle Encoding for Cached Adaptive Tessellation. In *Proceedings of High Performance Graphics, Poster Abstracts (online) (HPG '20 Posters).* ACM, New York, NY, USA, 2 pages.

1 INTRODUCTION

Hardware tessellation plays an important role in reducing CPU-GPU memory transfer overhead by offloading the generation of detailed geometry directly to the GPU. However, it is not without its limitations. Performance drops at high tessellation factors and inability to produce more than 64 splits per edge have driven developers and researchers to investigate alternative, software-based solutions. In this work, we present a modification to the adaptive GPU tessellation method presented by Khoury et al. [4], in order to increase its efficiency on current hardware. The authors' original design aims to avoid the above shortcomings of hardware tessellation. In fact, their approach enables a theoretical 2^{31} triangles to be generated from each input primitive. Instead of generating detailed geometry anew in every frame, tessellated primitives are produced incrementally by recursively splitting triangles over multiple frames, until a target criterion (e.g., projected edge length) is reached. Splits always occur at the midpoint of one triangle edge in a rotating fashion. The resulting triangles are written directly to a GPU buffer and can thereby remain cached over multiple frames. In order for this technique to be viable, generated triangles must be encoded in order to economize on-chip memory.

The authors' triangle encoding method requires only two integers (one for the original primitive and one tessellation key) per stored triangle and is derived from the compact representation by Gargantini for linear quadtrees [3], which have recently been ported to the GPU for real-time rendering [1, 2]. Specifically, the authors propose a key layout wherein each bit represents a transformation of barycentric coordinates. Each such transformation may apply scaling, translating, rotation or flipping operations. The key is in turn decoded by recursively applying these transformations to obtain barycentric coordinates that uniquely identify each tessellated subtriangle. For a detailed explanation of the original encoding and its application to cached tessellation, please refer to the original article [4]. However, such a recursive decoding method implies that rendering tessellated subtriangles becomes more compute-intensive as subdivision increases: a triangle that results from 30 recursive subdivisions requires 30 sequential matrix-vector multiplications to decode it for rendering, which is a poor fit for GPU hardware.

2 MODIFICATIONS

We propose an alternative triangle encoding method to avoid the recursive decoding procedure described above. Instead of exploiting transformation matrices, we try to map each tessellated subtriangle to a unique location in a grid overlaid with the original triangle in barycentric space. In doing so, we naturally waste one bit of the integer that we use as the key. However, we believe that the resulting performance gain is a viable argument for this tradeoff.

In order to support the full recursive splitting method by the authors, we must define two different states that each triangle can be in. We distinguish *2-state* and *4-state*. This nomenclature is easily explained: Consider an input triangle for tessellation. Assuming that we perform uniform subdivision down to a particular level by recursively splitting subtriangle edges in a rotating fashion, the result can be seen as the bottom-left, triangular half of a regular grid. Depending on the level, each grid cell then either contains 2 or 4 triangles (see Figure 3). Figure 2 outlines the full composition of the subtriangle key codes in both states.

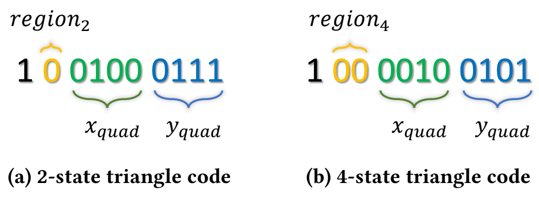

Figure 2: Illustration of the individual components of the subtriangle codes in 2-state and 4-state. Notice that the correct state can be detected by the position of the MSB.

We can interpret each triangle as part of a quad in a regular grid. From the position of the most significant bit (MSB), the state and quad grid resolution can be directly inferred. An even MSB position indicates 2-state, and an odd one indicates 4-state. Subtracting the proper number of region bits and dividing by 2 gives the quad grid resolution for this triangle in x and y. The quad coordinates are stored (x, y) in the corresponding lower bits. The region code interpretation is slightly more involved, since it depends on both quad coordinates in order to support fully recursive subdivision. Figure 3 illustrates how the region codes change while subtriangle encoding switches from 2-state to 4-state and back again.

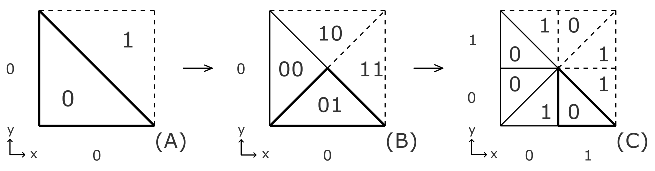

Figure 3: Transition from an initial, single quad in 2-state (A) to the same quad in 4-state (B). At the next subdivision (C), the quad is split into four with rotated configurations of the initial 2-state. Dotted lines mark triangles whose codes are never used, since they lie outside the input 0 triangle.

We have implemented alternative versions of the original article authors' methods as a drop-in replacement for computing keys that represent the next-lower/-higher level of subdivision and for decoding keys in constant time. Our splitting/merging of triangles is more complex, as in the original version, this requires only the concatenation/removal of a single bit. However, our results show that these changes are easily amortized by avoiding recursive matrix-vector multiplication and loop control logic for triangle decoding.

3 RESULTS

We have evaluated the performance of our new encoding scheme in the original OpenGL framework by Khoury et al. by modifying their implementation of adaptive tessellation via compute shaders. To create a challenging scenario with a high degree of subdivision, we have set the target triangle edge length to 1 pixel and the input geometry to a single quad. In this setup, the subdivision procedure routinely approaches the maximal possible level of 30 recursive splits. Figure 4 compares the rendering performance with our encoding scheme to the unaltered version. Timings were obtained by measuring the total GPU time in each frame over the animated camera path shown in Figure 1 with both methods separately. All timings were recorded at 1080p on an NVIDIA RTX 2070 GPU.

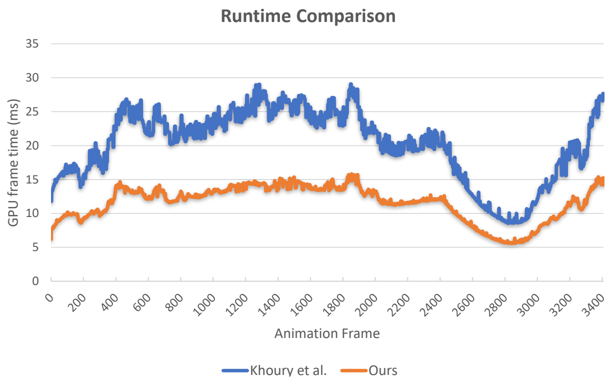

Figure 4: Measured total frame time for rendering the terrain along the camera path in Figure 1. Our constant-time decoding procedure can reduce frame times by up to 40%.

4 FUTURE WORK

The original work by Khoury et al. aimed to achieve fully adaptive tessellation. However, while their solution is straight-forward to implement, its simplicity limits its applicability. Since it includes no concrete method for resolving T-junctions, triangle subdivision must be governed by simple rules (e.g., distance to camera) to ensure that neighboring triangles differ by no more than one subdivision level. An additional pass that detects and resolves T-junctions (e.g., via hash maps) could provide a solution to this problem.

In the transition from a recursive to a grid-based encoding, one bit is wasted, since half of the possible codes represent triangles that lie outside the input triangle. Trivial solutions that reinterpret the corresponding codes introduce additional control paths and overhead. More reflection on the topic is required in order to find an efficient way for reclaiming the lost bit.

REFERENCES

[1] Wade Brainerd, Tim Foley, Manuel Kraemer, Henry Moreton, and Matthias Nießner. 2016. Efficient GPU Rendering of Subdivision Surfaces Using Adaptive Quadtrees. *ACM Trans. Graph.* 35, 4, Article 113 (July 2016), 12 pages.
[2] Jonathan Dupuy, Jean-Claude Iehl, and Pierre Poulin. 2018. Quadtrees on the GPU. *GPU Pro: Advanced Rendering Techniques* 5 (10 2018), 211–222.
[3] Irene Gargantini. 1982. An Effective Way to Represent Quadtrees. *Commun. ACM* 25, 12 (1982), 905–910.
[4] Jad Khoury, Jonathan Dupuy, and Christophe Riccio. 2019. Adaptive GPU Tessellation with Compute Shaders. *GPU Zen: Advanced Rendering Techniques* 2 (2019), 3–17.