An Improved Triangle Encoding Scheme for Cached Tessellation

Bernhard Kerbl[1] , Linus Horváth[1], Daniel Cornel[2] and Michael Wimmer[1]

[1]TU Wien, Vienna, Austria   [2]VRVis Forschungs-GmbH, Vienna, Austria

Abstract

With the recent advances in real-time rendering that were achieved by embracing software rasterization, the interest in alternative solutions for other fixed-function pipeline stages rises. In this paper, we revisit a recently presented software approach for cached tessellation, which compactly encodes and stores triangles in GPU memory. While the proposed technique is both efficient and versatile, we show that the original encoding is suboptimal and provide an alternative scheme that acts as a drop-in replacement. As shown in our evaluation, the proposed modifications can yield performance gains of 40% and more.

CCS Concepts

*• Computing methodologies → Rendering; **Parallel computing methodologies;***

1. Introduction and Related Work

The concept of replacing fixed-function stages of the hardware rasterization pipeline with purpose-built, software-based solutions recently received increased interest in the computer graphics community [KKSS18, KSW21]. Among these built-in stages that are run on the GPU during rendering, hardware tessellation plays a key role in reducing CPU-GPU memory transfer overhead, since it allows to offload the generation of detailed geometry directly to the GPU. However, performance drops at high tessellation factors and the inability to produce more than 64 splits per edge encourage developers to investigate alternatives [LJL13]. A particularly noteworthy approach is the adaptive GPU tessellation method presented by Khoury et al. [KDR19]. The authors' approach aims to exploit caching of triangles on the GPU between frames, as well as enabling a significantly higher tessellation factor per input primitive: their approach enables a theoretical 2^{31} triangles to be generated from each input primitive. To reduce the complexity of cache control, the authors implicitly target applications with slow-changing environments: Instead of generating the full detail anew in every frame, tessellated primitives are produced incrementally by splitting or merging triangles over multiple frames until a desired projected edge length is reached. Splits and merges occur according to longest edge bisection (LEB) of canonic isosceles right triangles in barycentric space [Dup20]. The results are cached in GPU buffers.

In order for this technique to be viable, generated triangles must be compressed in order to economize on-chip memory. The authors' method of storing a tessellated triangle requires only two integers per primitive: one for the original triangle and a tessellation key to describe a subtriangle embedded within. The approach is inspired by the compact representation for linear quadtrees [Gar82, DIP18], which was recently ported to the GPU for real-time rendering [BFK*16]. Specifically, Khoury et al. propose a key lay-
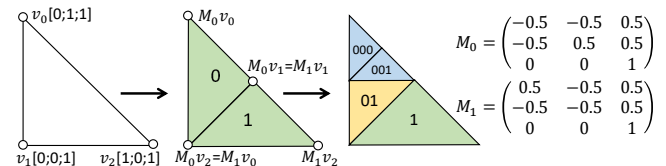

Figure 1: *Original encoding by Khoury et al. [KDR19]. Concatenting the bits for each LEB split into "left" (`0`) and "right" (`1`) yields a triangle's code. Converting to barycentric coordinates requires recursively applying the corresponding transformations (M_0, M_1).*

out wherein each bit represents a transformation of barycentric coordinates. The key is, in turn, decoded by recursively applying these transformations to obtain barycentric coordinates that uniquely identify each tessellated subtriangle (see Figure 1). For a complete description of the original encoding and its application to cached tessellation, please refer to the original article [KDR19]. The authors continue using this encoding in their recent work on concurrent binary trees [Dup20, DYDR21]. However, its recursive nature implies that rendering tessellated subtriangles becomes more compute-intensive as subdivision increases: a triangle that results from 31 subdivisions requires 31 sequential matrix-matrix multiplications to decode it for rasterization; a poor fit for GPU hardware.

2. Our Triangle Encoding Scheme

We propose an alternative triangle encoding method to avoid the recursive decoding procedure described above. Instead of relying on transformation matrices, we map each tessellated subtriangle to a unique location and configuration inside a multi-resolution grid, whose dimensions align with two sides of the triangle. This enables us to produce each triangle from its encoded form in constant time.
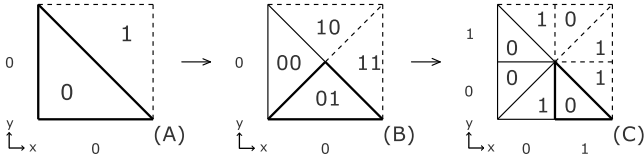
Figure 2: *Transition from an initial, single-cell grid in 2-state (A) to the same quad in 4-state (B). In 4-state, each triangle requires an extra bit to store its index in the cell. Note that this is the singular layout that cells assume in 4-state. In (C), the cell is split into four, each representing a variation of the initial 2-state. Dotted lines indicate invalid triangles, that fall outside of the original primitive.*

2.1. Mapping Tessellation to a Multi-Resolution Grid

In our scheme, each triangle is fully described by its location in a grid of a given resolution, as well as a configuration for its current *state*. In order to support fine-granular recursive tessellation as described in [KDR19], we must define two different states that each triangle can be in: We distinguish *2-state* and *4-state*. This nomenclature is motivated as follows: Consider an input triangle for tessellation. If we perform uniform tessellation down to an arbitrary level L via recursive LEB, the resulting triangles are then enclosed by the lower-left cells of a regular grid with resolution $2^D \times 2^D$, where $D = \lfloor \frac{L}{2} \rfloor$. Depending on L, each cell either contains two or four triangles. Hence, a triangle's state is implicitly defined by the resolution of the grid in which we consider it to be embedded. Given a cell and state, each triangle is uniquely identified by one out of two/four possible indices, respectively (see Figure 2).

2.2. Basic Encoding Layout

We can efficiently store the required information to identify each unique tessellated subtriangle by compressing it into the binary representation of an unsigned integer. For this to be unambiguous and efficient, the ratio of x to y grid resolution must be a known power of two. In the basic case, this ratio is 1:1 (although we will see later that this choice is not ideal). We can then infer both the grid resolution and state for a triangle from the position of the most significant bit (MSB): An even MSB position indicates 2-state, and an odd one indicates 4-state. After consuming the corresponding state's triangle index bits (either 1 or 2), the $\log_2$ of the grid resolution in x and y is equal to half the number of the remaining bits, in which the actual grid cell coordinates (x, y) are stored. Figure 3 outlines the full composition of the subtriangle key codes in either state. As in [KDR19], we omit the MSB in examples unless stated otherwise.

While this encoding is adequately compact and unambiguous, there are two noteworthy implications. First, to perform the conversion from codes to individual numeric variables and back, we need to use helper functions that take care of (un-)packing the codes. Second, note that half of the triangles that this encoding scheme can represent are invalid (see Figure 2). Thus, one bit is wasted with this basic approach. In Section 2.4, we show how we can rectify this by slightly adapting the conversion helper functions.

(a) *2-state triangle code* **(b)** *4-state triangle code*
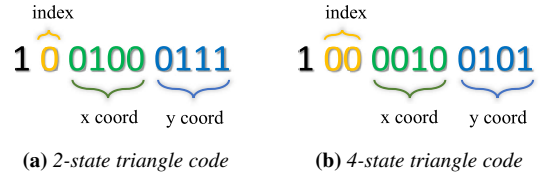
Figure 3: *Components of subtriangle codes in 2-state and 4-state. Grid resolution (16×16) and state are implicitly given by the MSB.*

2.3. Tessellation Rules

We outline the required rules for applying our encoding to tessellation in the context of LEB. In contrast to the scheme in [KDR19], handling our codes requires more effort for splitting and merging a given code. However, decoding a tessellated triangle becomes less expensive. This suggests a strong opportunity for performance improvement since the number of split and merged triangles in each frame only ever represents a subset of the total displayed triangles.

Splitting If a higher-detail level is required, triangles can be split at the longest edge, yielding two new triangles. Figure 2 illustrates how triangle indices and perceived grid resolutions change in our scheme as subtriangles transition between different states. To avoid flipping artifacts due to changing triangulation between tessellation levels, our rules implicitly distinguish 2-state cells with / and \ diagonals. While there are two 2-state variants, there is only one 4-state layout. This setup suffices to ensure that child triangles avoid T-junctions if they differ by at most one tessellation level [KDR19].

When transitioning from a 2-state to a 4-state by splitting, the grid resolution and grid cell coordinates remain the same, but the correct indices of the triangle's children must still be computed: If the sum of grid cell coordinates $x + y$ is odd, the indices for the first|second child triangle are computed from the current index i as $i_{0|1} = 2i + 0|1$. Otherwise, the indices become $i_{0|1} = i + 0|2$. If we instead split a triangle in 4-state, we must compute new child indices as well as new grid cell coordinates. In this case, the two child indices are directly given by the two bits of the current index. The children's x coordinates can then be computed as $x_{0|1} = 2x + i_{1|0}$. For the children's y coordinates $y_{0|1}$, we have:

$$y_0 = \begin{cases} y & \text{if } i = 1 \\ y+1 & \text{otherwise} \end{cases}, \quad y_1 = \begin{cases} y+1 & \text{if } i = 2 \\ y & \text{otherwise} \end{cases}$$

Merging To reduce the tessellation degree, two children can be merged into one triangle. In practice, this requires the ability to compute the parent key from either child key. Furthermore, we need a function that will always yield `true` for only one child to avoid both children assuming the role of the merged parent.

For merging two 4-state triangles to a 2-state triangle, we identify the ones that will take the role of the parent by selecting those with index $i = 3$ or $i = 0$. Note that these two always form part of two separate 2-state triangles on the previous tessellation level. In this transition, the x and y coordinates remain unchanged. If the sum of grid cell coordinates $x + y$ is odd, the parent index is $i_p = \lfloor \frac{i}{2} \rfloor$, otherwise i_p is given by the lower bit of the child's index. For merging from 2-state to 4-state, we must first identify triangles that will

never map to the same parent. To select these, we consider their *y* coordinates: a child assumes the parent's role if its index and *y* coordinate are both even/odd. It is easily confirmed in Figure 2 that no two triangles in 2-state originating from the same parent in the previous tessellation level will fulfill this criterion. To obtain the parent's grid coordinates, we simply divide the children's coordinates by two. The index of the parent triangle is computed as:

$$
i_p = \begin{cases} i & \text{if both } x \text{ and } y \text{ are even} \\ 2+i & \text{if } x \text{ is odd and } y \text{ is even} \\ 2i & \text{if } x \text{ is even and } y \text{ is odd} \\ 2i+1 & \text{if both } x \text{ and } y \text{ are odd} \end{cases}
$$

Decoding and Displaying For displaying the encoded subtriangles and deciding whether splits/merges are desired for maintaining image quality, the codes must first be translated into geometry primitives again. In [KDR19], this is achieved by computing the barycentric coordinates of subtriangles inside the enclosing, untessellated input triangle. The tessellated triangle's vertices are then obtained by simple interpolation of those in the top-level triangle. To produce the barycentric coordinates, the authors must apply the previously mentioned recursive matrix transformations. In contrast, once unpacked, the generation of barycentric coordinates with our encoding is straightforward. We can identify the corners of the grid cell as combinations of $(x, x+1)$ and $(y, y+1)$ and convert those to normalized coordinates by dividing by the grid resolution. In 2-state, each triangle is formed from three of these points. In 4-state, we only need to compute the additional center of the grid cell in barycentric space to generate the coordinates for each triangle.

2.4. Reclaiming the Lost Bit

Although the presented encoding and rules suffice to provide a drop-in replacement of the original encoding used in [KDR19], they come with a drawback, namely, the fact that one bit is effectively wasted. Hence, compared to the original approach, the number of representable triangles using a single integer is halved.

To rectify this problem, we first observe that the area of an isosceles right triangle is half of a square with the same side length *a*. The same area is also contained in a rectangle with side lengths *a* and $\frac{a}{2}$. Hence, we can consider each point in the triangle as a non-continuous mapping from points in a rectangle with the same area. Doing so would be beneficial since grid coordinates can just as easily be applied for exhaustive spatial indexing of rectangular areas. In practice, we can achieve this by introducing special cases that map particular triangles in the grid to a different location during splitting, merging, and decoding. We first stipulate that the grids must now be rectangular, with their vertical resolution being twice that of their horizontal one (except for the single-cell case). Hence, *x* coordinates will require one bit less to store, which immediately grants us the previously wasted bit. The so-adapted method, including special case treatment, is outlined in Figure 4. We start from the initial, untessellated triangle in a 1×1 grid; apart from the MSB, no additional bits for resolution or index are needed. We initiate the splitting method to make the first split into 0 and 1. Key 1 assumes the role of a bottom triangle in a single-cell 4-state. Hence, when another recursive split of 1 is performed, its right half would be the first triangle to require that the horizontal grid resolution is
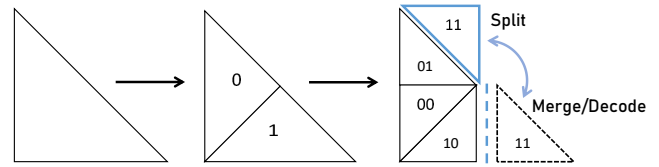
Figure 4: *After two subdivisions, the protruding lower right triangle* 11 *and all its children are treated as the upper right section of a rectangle. The transformation is undone whenever triangles in this region are decoded or the children of triangle* 1 *are merged.*

increased. We explicitly create its children as 10 and 11. Their full keys, including the necessary descriptive MSB, are 110 and 111. With our new scheme based on a *x:y* resolution ratio of 1:2, the MSB in (odd) position 3 indicates that both triangles are in 2-state and the horizontal grid resolution is 1, while the vertical resolution is 2. Both triangles represent index 1 of different 2-state variants at different *y* coordinates, completing the 1×2 rectangular grid.

For merging, the only special treatment required is to always pick the parent of 11 to be 1. Finally, for decoding and displaying, the barycentric coordinate computation must be adapted for tessellated triangles that appear to fall outside of the top-level primitive. This is exactly the case if the sum of the grid cell coordinates and the unpacked index (divided by two in 4-state) exceeds the vertical grid resolution. The actual *x, y* coordinates are obtained by subtracting the unpacked coordinates from the vertical grid resolution, minus 1. The correct index is obtained by flipping the bits of the unpacked index. Once this transformation is complete, decoding and displaying can be performed for these triangles as described above.

3. Evaluation and Discussion

We evaluate our triangle encoding scheme by comparing performance for adaptive tessellation against the work by Khoury et al. [KDR19]. We measure the run times of both methods in the authors' reference framework for heightmap terrain rendering [Dup18]. The changes to run it with our approach are minimal since only the corresponding rule behaviors have to be modified. To provide a hardware-based baseline, we also consider the recursive tessellation approach by Lee et al. [LJL13]. Timings were obtained by measuring the total GPU time in each frame over an animated camera path, recorded at 1080p on an NVIDIA RTX 2070 Super GPU. The different approaches were evaluated in two scenes on fixed camera paths. For the first, we generate the heightfield from a high-resolution texture with an input geometry of only two triangles arranged in a quad (see Figure 5a) and test desired triangle edge lengths of 4, 2, and 1 pixel(s). Figure 5b plots the rendering performance with our encoding scheme and the unaltered version. For the second scene, we generate the heightfield procedurally with an expensive noise function (see supplemental video). To ensure real-time frame rates, we set the desired edge length to 5 pixels.

In both scenes, we find that the hardware-based approach by Lee et al. is 6–7× slower on average than the software-based approaches. This is expected since it does not cache its results between frames. However, even if caching was used, it stands to reason that the storing and loading of massive amounts of explicit tri-
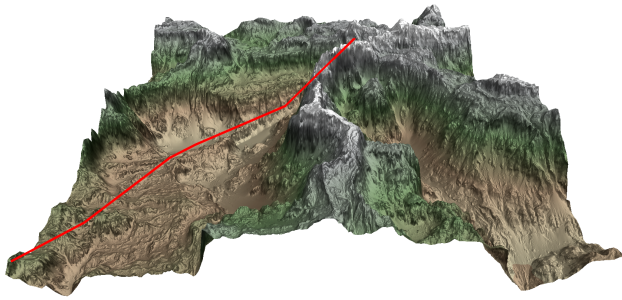
(a) *Rendered test scene with adaptively tessellated terrain*
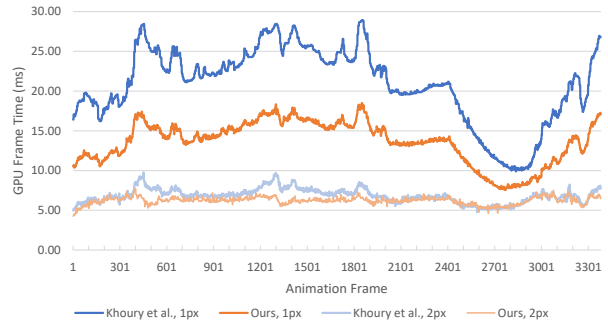
(b) *Recorded frame times with original and modified encoding scheme*

Figure 5: *(a) First test scene for evaluating performance benefits of our encoding scheme. The red line indicates the camera path. Tessellation levels are selected dynamically in each frame. (b) Measured frame time for rendering the terrain in (a) along the camera path with 2 and 1 pixel(s) desired edge length. Compared to Khoury et al., our constant-time triangle encoding can reduce frame times by more than 40%.*

angle data in each frame would limit its performance and applicability. Among the caching-based approaches, our novel encoding scheme can yield clear performance benefits. In the first scene, our method yields a small benefit for a desired projected edge length of 4 pixels (2.01% on average). At an edge length of 2 pixels, our method is 9.93% faster and for 1 pixel 38.73%. For both approaches, memory consumption for caching peaked at 234 MiB, with a desired edge length of 1 pixel. In the second (procedural heightfield) scene, our method performs 19.05% better on average for an edge length of 5 pixels. The ability to improve performance in this scenario with a longer edge length can be explained by the GPU being bottlenecked by arithmetic workload due to the evaluation of the procedural heightfield. For both approaches, memory consumption for caching remained below 16 MiB in this scene.

4. Conclusion and Future Work

In this short paper, we presented a modified, compact triangle encoding scheme for use in cached tessellation. The original work by Khoury et al. achieves fast, fully adaptive triangle tessellation in software. In our work, we have demonstrated how to improve their proposed triangle encoding scheme to move from recursive logic to constant run time. A direct comparison in the original authors' framework has shown that the modified encoding scheme reduces GPU workload and can raise performance by 40% or more when the tessellation rate is high. Thus, our proposed solution serves as a drop-in solution to further increase the effectiveness of their approach. An implementation is available at https://github.com/cg-tuwien/tessellation-encoding.

The context in which the original solution by Khoury et al. was presented is LEB in a 2D space where each triangle forms a unit triangle. While this is both effective and easy to implement, its applicability is limited: It provides no explicit method for resolving T-junctions; triangle subdivision must be governed by simple rules (e.g., distance to the camera) to ensure that adjacent triangles differ by at most one subdivision level. Future work is required to ensure their approach generalizes to different tessellation criteria while retaining its high performance. We did not compare against the follow-up work to [KDR19], concurrent binary trees [Dup20]:

there, the recursive behavior is a necessity of tree traversal, hence using a constant-time decoding would not change its complexity. However, the authors have stated that they are pursuing a sparse hybrid approach, which would most likely benefit from our work.

References

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

[DIP18] DUPUY J., IEHL J.-C., POULIN P.: Quadtrees on the GPU. *GPU Pro: Advanced Rendering Techniques 5* (2018), 211–222. 1

[Dup18] DUPUY J.: Implicit Subdivision on the GPU. https://github.com/jdupuy/opengl-framework/tree/master/demo-isubd-terrain, 2018. [Dec-19-2021]. 3

[Dup20] DUPUY J.: Concurrent Binary Trees (with Application to Longest Edge Bisection). *Proc. ACM Comput. Graph. Interact. Tech. 3*, 2 (2020). 1, 4

[DYDR21] DELIOT T., YAO X., DUPUY J., RIJNEN K.: Experimenting with Concurrent Binary Trees: Large Scale Terrain Rendering. In *ACM SIGGRAPH 2021 Courses, Advances in Real-Time Rendering in Games* (2021), SIGGRAPH '21. 1

[Gar82] GARGANTINI I.: An Effective Way to Represent Quadtrees. *Commun. ACM 25*, 12 (1982), 905—910. 1

[KDR19] KHOURY J., DUPUY J., RICCIO C.: Adaptive GPU Tessellation with Compute Shaders. *GPU Zen: Advanced Rendering Techniques 2* (2019), 3–17. 1, 2, 3, 4

[KKSS18] KENZEL M., KERBL B., SCHMALSTIEG D., STEINBERGER M.: A High-Performance Software Graphics Pipeline Architecture for the GPU. *ACM Trans. Graph. 37*, 4 (2018). 1

[KSW21] KARIS B., STUBBE R., WIHLIDAL G.: A Deep Dive into Nanite Virtualized Geometry. In *ACM SIGGRAPH 2021 Courses, Advances in Real-Time Rendering in Games* (2021), SIGGRAPH '21. 1

[LJL13] LEE H., JEONG Y., LEE S.: Recursive Tessellation. In *SIGGRAPH Asia 2013 Posters* (New York, NY, USA, 2013), SA '13, Association for Computing Machinery. 1, 3