Watertight Incremental Heightfield Tessellation

Daniel Cornel, Silvana Zechmeister, Eduard Gröller, and Jürgen Waser

Abstract—In this paper, we propose a method for the interactive visualization of medium-scale dynamic heightfields without visual artifacts. Our data fall into a category too large to be rendered directly at full resolution, but small enough to fit into GPU memory without pre-filtering and data streaming. We present the real-world use case of unfiltered flood simulation data of such medium scale that need to be visualized in real time for scientific purposes. Our solution facilitates compute shaders to maintain a guaranteed watertight triangulation in GPU memory that approximates the interpolated heightfields with view-dependent, continuous levels of detail. In each frame, the triangulation is updated incrementally by iteratively refining the cached result of the previous frame to minimize the computational effort. In particular, we minimize the number of heightfield sampling operations to make adaptive and higher-order interpolations viable options. We impose no restriction on the number of subdivisions and the achievable level of detail to allow for extreme zoom ranges required in geospatial visualization. Our method provides a stable runtime performance and can be executed with a limited time budget. We present a comparison of our method to three state-of-the-art methods, in which our method is competitive to previous non-watertight methods in terms of runtime, while outperforming them in terms of accuracy.

Index Terms—Visualization techniques and methodologies, heightfield rendering, terrain rendering, level of detail, tessellation

1 Introduction

Heightfield rendering is essential in many applications using geospatial visualization, particularly of 3D geoinformation systems for visualizing digital elevation models and environmental simulation data. Heightfields are discrete scalar fields, usually defined on regular or unstructured grids, and have no visual representation of their own. For visualization, a continuous surface must be reconstructed from the values by interpolation. The reconstructed surface then has to be approximated by triangulated geometry that GPUs can handle, which is called tessellation.

Tessellation of heightfields is challenging, as application requirements often collide with hardware limitations. As current GPUs can only render a few million triangles in real time, it is not feasible to uniformly triangulate each cell of a heightfield with several hundred million cells or more. To overcome this problem, tessellation algorithms with view-dependent level of detail (LoD) have been proposed that decouple the complexity of the heightfield from its triangulation. However, each of these algorithms comes with its own set of limitations and trade-offs between the achievable LoD and interactivity. A common problem here is the inability to produce a watertight triangulation. Instead, many algorithms produce cracks caused by T-junctions between adjacent triangles leading to visual artifacts. Other limitations include lack of support for dynamic updates of the heightfield data, no efficient support for higher-order interpolation, a limited number of subdivisions per triangle, and an oftentimes lower than real-time performance. In the context of decision support for flood management as illustrated in Fig. 1, which provides a real-world use case for our proposed method, these limitations are too severe. For this use cases, we require a tessellation algorithm that provides watertight triangulations of multiple large heightfields defined on regular or adaptive grids with higher-order interpolation enabling a wide zoom range with continuous LoD at steady real-time performance.

In this paper, we propose a novel GPU-based heightfield tessellation with adaptive LoD that fulfills all of these requirements. We tackle the most challenging aspect of guaranteed watertightness during parallel processing by maintaining triangle adjacency information to operate on the direct neighborhood of each triangle during subdivision and merging. To avoid corrupting changes of shared information of neighboring triangles during parallel processing, we employ task scheduling by graph coloring. Our solution also caches a computed triangulation as starting point for the next computation. As a result, only incremental changes are required instead of a complete recomputation, which greatly reduces the computational effort and thus significantly improves the runtime. This particularly benefits applications in geoinformation systems, where complex interpolation methods mean that sampling vertex positions accounts for a large part of the tessellation effort.

Our solution fills a gap in heightfield visualization, which is the artifact-free visualization of heightfields that fit into GPU memory completely, but are too large to be rendered without view-dependent LoD, with at least 60 frames per second. We focus on the concrete use case of visualizing a country-sized static terrain heightfield defined on an adaptive grid overlaid by a dynamic simulation heightfield in a focus area with all data provided as unfiltered scalar fields. Yet, our solution is largely independent of the underlying heightfield data, their sampling strategy, and the employed LoD metric such that it can be integrated into many existing systems with minimal effort. In summary, we contribute a novel approach for the adaptive tessellation of large heightfields that

- is guaranteed to be watertight

- *Daniel Cornel is with VRVis Forschungs-GmbH, Vienna, Austria. E-mail: cornel@vrvis.at*
- *Silvana Zechmeister is with VRVis Forschungs-GmbH, Vienna, Austria.*
- *Eduard Gröller is with TU Wien, Vienna, Austria.*
- *Jürgen Waser is with VRVis Forschungs-GmbH, Vienna, Austria.*

Manuscript received May 5th, 2022.

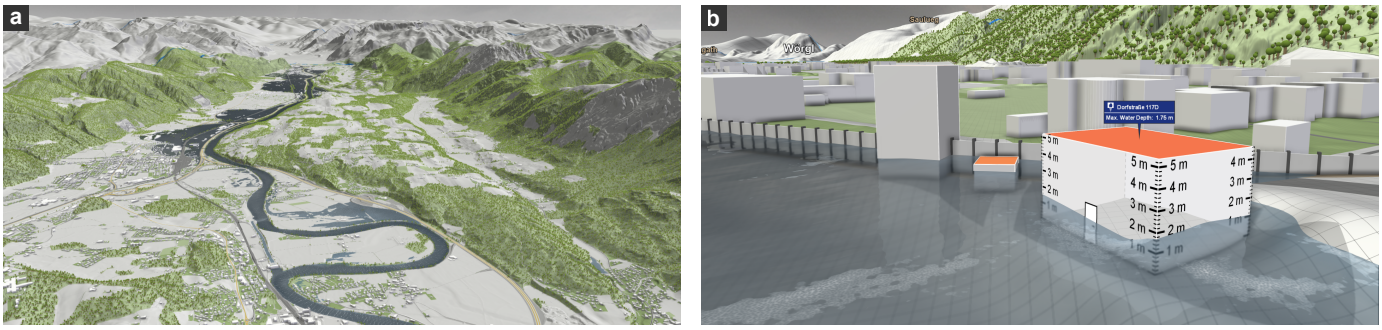

Fig. 1: Real-time visualization result of our method for decision support in flood management. (a) Overview perspective with geospatial data of Austria. Three nested terrain heightfields defined on an area of $572\,\mathrm{km} \times 293\,\mathrm{km}$ are combined with a dynamic water heightfield in a focus region of $22\,\mathrm{km} \times 15\,\mathrm{km}$. (b) Close-up of flood simulation results at 1 m resolution.

- minimizes computational effort by incrementally updating previous solutions
- provides stable and controllable real-time performance for interactive tasks
- allows for an unlimited number of subdivisions and levels of detail
- supports multiple and nested heightfields.

An implementation of our solution is provided in the supplemental material of this paper.

2 Related Work

Heightfield rendering in real time is a challenging task due to the combination of the required surface reconstruction from complex heightfield data with the efficient generation of a view-dependent visual representation. The most straightforward combination is achieved through ray casting [1], [2], [3], where the reconstructed surface is directly rendered through the intersection of a view ray with the polynomial surface of the heightfield's interpolant. The downside of ray casting is that the interpolant has to be evaluated for each pixel of the screen, which means that rendering performance depends on both the complexity of the used surface reconstruction and the screen size. In practice, this limits surface reconstruction to simple interpolants such as bilinear interpolation, for which a ray-surface intersection can be calculated efficiently, and excludes more accurate methods such as kriging [4], local refinable splines [5], and adaptive bicubic interpolation [6]. In previous work, combining higher-order surface reconstruction with real-time ray casting has not been successful [7].

This is why most approaches rely on the generation of a view-dependent triangulation as an approximation of the reconstructed surface. One option is to move the triangulation with the camera and resample the vertices on the content they currently cover, as demonstrated with the projected grid [8], [9], the persistent grid [10], [11], and the projected mesh [12]. These approaches provide implicit levels of detail with a consistent performance, but may introduce undersampling and hide important features. Another option is to maintain a triangulation fixed in space, but changing its LoD locally. There have been countless approaches on how to generate this triangulation, of which the most noteworthy ones have been surveyed by Pajarola and Gobbetti [13]. The geometry clipmap [14], [15] is one of the most popular and robust triangulation approaches for real-time applications. However, it is limited by only providing a few discrete regions of decreasing resolution instead of continuous levels of detail, and only considers the 2D distance of triangles to the camera instead of the actual heightfield to determine the LoD.

Several approaches have been proposed to facilitate the evolving parallel processing capabilities of modern GPUs. Early hybrid approaches maintain a hierarchical representation of the adaptive subdivisions on the CPU, which is then sent to the GPU for meshing [16], which requires expensive CPU-GPU communication. For GPUs that cannot generate new geometry on the fly, patch-based mesh refinement [17], [18] can be used, which replaces a patch of a mesh with a precomputed refinement pattern. With support for the dynamic generation of geometry, tessellation can become much more flexible, as proposed for geometry shaders [19], [20], tessellation shaders facilitating the hardware tessellation unit [21], [22], [23], compute shaders [24], [25], and mesh shaders [26], [27]. Most approaches based on tessellation shaders are limited in the achievable LoD by a maximum edge tessellation factor of 64 for triangles. To achieve unlimited tessellation, several approaches [28], [29] subdivide the initial triangulation prior to hardware tessellation on the CPU. Lee et al. [30] avoid this CPU overhead by applying hardware tessellation recursively. However, while the hardware tessellation unit in GPUs is designed specifically for adaptively generating vast amounts of geometry, e.g. for subdivision surfaces, it cannot be used to revert the process to simplify finely triangulated geometry. This means that after camera changes, the entire triangulation has to be generated from scratch.

The approach by Khoury et al. [24] addresses this problem by caching triangles of a previous computation that are then subdivided and merged as needed. Extending this approach, Kerbl et al. [25] propose a different encoding for the cached triangles to accelerate the calculation of vertex coordinates, making this possibly the most efficient and flexible heightfield rendering approach to date. However, neither of the two approaches produces a watertight triangulation, and both of them are limited in the number of subdivisions by the data type used for triangle encoding. If the maximum number of subdivisions is exceeded at runtime, the initial triangulation has to be refined, forcing a recalculation of the entire view-dependent triangulation.

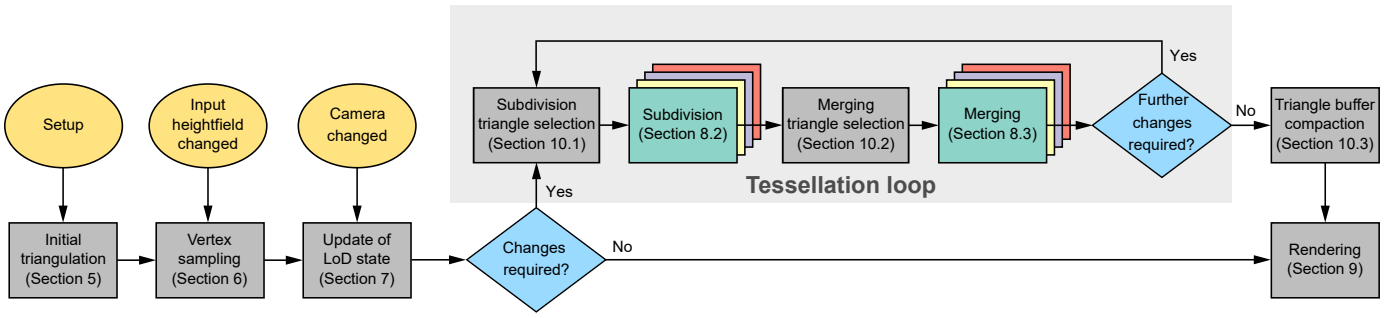

Fig. 2: Overview of the algorithm. Starting with an initial triangulation, an adaptive triangulation is computed from the previous result each frame. After evaluating the current level of detail of each triangle, necessary changes are applied iteratively in the tessellation loop, where the affected triangles are separated by their assigned graph color for subdivision and merging and are processed in four separate passes. The resulting triangle buffer is optionally compacted and rendered.

Another extension to the approach of Khoury et al. using concurrent binary trees for triangle encoding [31] eliminates T-junctions, but requires a predefined maximum number of subdivisions, while the required pre-allocated memory grows exponentially with this number.

3 Overview of the Algorithm

Our algorithm works on a set of triangles with adjacency and hierarchy information stored for each triangle. This set is maintained in GPU memory and is modified with compute shaders to achieve a sufficient triangle density in screen space at minimum cost. As illustrated in Fig. 2, it consists of three main steps, which are the determination of the triangle LoD states, the tessellation loop, and finally a rendering pass, with additional steps inbetween to optimize the pipeline. After a trivial coarse view-independent initial triangulation into right triangles, the rest of the pipeline is executed conditionally if an update of the result from the previous frame is required. Whenever the heightfield changes due to progression of a simulation or by reading another time step from the file system, all vertices of existing triangles are resampled on the heightfield. Succeedingly or every time the camera perspective or user-defined target LoD changes, the LoD state for all triangles is updated, which can be *subdivide*, *merge*, or *keep unchanged*. If all triangles are kept, they are rendered immediately.

If changes are required through subdivision or merging, we enter the tessellation loop, which iteratively refines the triangles as needed. This is the core of the algorithm which requires special data structures and careful synchronization for concurrent processing, which is explained in detail in Section 8. Subdividing a triangle or merging triangles back together changes the adjacency information of neighboring triangles along the edges. To avoid overly complicated low-level synchronization between triangles, we use task scheduling based on graph coloring [32], for which we interpret the mesh as a graph where triangles are the nodes. The graph colors group the triangles into four separate groups that can be processed concurrently. Based on the LoD state, we perform the subdivision separately for the triangles of the four different graph color groups. During subdivision, the shared hypothenuse of two triangles is split, a new vertex is created and sampled on the heightfield. Four new triangles are created, added as child triangles to their parents, and adjacency information of neighboring triangles is updated. For each of the new triangles, the LoD state is calculated and assigned. Then, the triangles are separated again by graph color for merging. During merging, four neighboring child triangles and their common vertex are removed and the adjacency of neighboring triangles is reset to reference their two parent triangles. The sequence of subdivision and merging is repeated until a termination criterion is reached: Either fewer than a minimum number of triangles have been subdivided or merged, or a predefined processing time budget has been used up.

The final step in the pipeline is efficient triangle rendering from an index buffer without any heightfield sampling, since all vertices have already been sampled on the heightfield during subdivision. In subsequent frames, if the LoD does not need to be updated, the tessellation loop is skipped entirely and the cached triangles are rendered.

4 Data Structure

Our algorithm operates on a set of triangles in a memory pool with a fixed size of $8 \cdot 2^{20}$ elements that we refer to as *triangle buffer*. The data we store for each triangle are listed as a struct in Fig. 3. In the GPU implementation, we use separate arrays for the individual struct members for better cache coherence. However, for the sake of a simple explanation of our algorithm, we treat the triangle buffer as an array of structs in the remainder of the paper.

Each triangle contains *semantic flags* in a bitfield concatenated with a *graph color*. The first semantic flag is a *deleted flag*, which indicates that the element of the triangle buffer is empty or can be overwritten. All elements in the triangle buffer are initialized as deleted triangles. The *LoD state* of the triangle expresses the operation necessary to approach the target LoD locally—*subdivide*, *merge*, or *keep*—and is encoded as a 2-bit value. Finally, each triangle is assigned a conceptual *graph color* in the form of a 3-bit value $\in [0, 7]$.

We include a *heightfield index* so that each triangle can be matched to a heightfield for sampling and rendering if multiple heightfields are being processed. This way, triangulations of all heightfields can simply be stored in a shared triangle buffer instead of separate ones. This value can be omitted in applications with a single heightfield.

```
1: struct Triangle
2:     byte deleted | LoD state | graph color
3:     byte heightfield index
4:     int parent triangle index
5:     int first child triangle index
6:     int3 adjacent triangle indices
7:     int3 vertex indices
```

Fig. 3: Overview of the data stored for each triangle.

When merging triangles as explained in Subsection 8.3, a set of four triangles is replaced by their two corresponding parent triangles. For this case, we store the *parent triangle index* for each triangle. Triangles of the initial tessellation are root triangles that cannot be merged any further, which is indicated with a parent triangle index of -1.

Likewise, we store the *first child triangle index* for each parent triangle. During subdivision, we take care that the child indices of the two new triangles are always consecutive, so it suffices to store the first one. A child index of -1 indicates that a triangle has no children and is therefore a leaf triangle in the final tessellation.

As both subdivision and merging interact with neighboring triangles, we require neighborhood information of the triangles, which we store as a triple of *adjacent triangle indices*. The order of indices is chosen such that the order of adjacent triangles is counter-clockwise in 2D and the adjacent triangle along the hypothenuse is the second index, i.e., at component index 1. This simplifies later checking whether two adjacent triangles share a common hypothenuse to a comparison of the adjacent triangle indices at this component index. A missing adjacent triangle at an edge is indicated by the triangle index -1, which is the case along the heightfield boundaries.

Finally, 3D vertex positions are stored together with a heightfield index in a separate *vertex buffer*, which each triangle refers to with a triple of *vertex indices*. Analogous to adjacent triangles, vertex indices are ordered such that the triangle is stored counter-clockwise in 2D. Start and end vertices of the triangle's hypothenuse correspond to component indices 1 and 2, i.e., the second edge of the triangle. Using such a predefined order of indices simplifies the triangle splitting at runtime.

For memory efficiency, we keep track of the free space of the triangle and vertex buffers, which includes the unused portion at the end as well as deleted elements. The unused area in the buffers is marked implicitly by the highest index of all used triangles or vertices, respectively. The deleted elements are tracked in additional *free index buffers* to which free triangle and vertex indices are pushed during merging with the help of a counter. If a free index in one of the buffers is required for new elements during subdivision, a lookup is first performed in the free index buffer and the counter is decreased. Only if the free index buffer is empty, a new index from the unused space at the end of the triangle or vertex buffer is acquired. In total, our data structures occupy 376 MB of GPU memory.

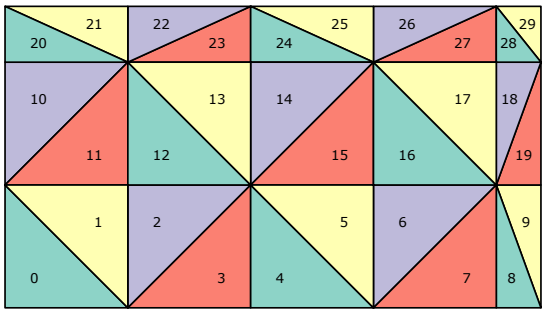

Fig. 4: Initial tessellation example. Cells are alternately split along one of their diagonals. Colors and numbering indicate type and order of the resulting triangles.

5 Initial Triangulation

Tessellation commonly works by iteratively refining a coarse initial triangulation until the level of detail is sufficient. In this section, we explain how to create an initial triangulation of a regular *tessellation grid* that covers the rectangular area of a heightfield. We start with a heightfield given as scalar field defined on a regular or adaptive 2D grid with a known minimum cell size. We create a regular tessellation grid with coarse cells, where the cell size is a power-of-two multiple k of this minimum cell size. This integer factor is determined through an iterative process such that the number of cells of the tessellation grid just barely exceeds 2000. This is an empirical value that does not significantly influence the algorithm. However, defining a roughly equal number of cells across different heightfields has the effect that the initial tessellation is decoupled from the complexity of the heightfield and guarantees similar runtime performance across different use cases.

We now create a triangulation for the tessellation grid by constructing two triangles per grid cell, as shown in Fig. 4. To fit the original heightfield extents, the triangulation grid starts at the bottom left corner of the heightfield, and the vertices of the rightmost column and upmost row are shifted towards the heightfield boundaries. The resulting triangulation consists of $2n$ triangles and $(w+1)(h+1)$ vertices, where $n = wh$ is the number of cells and w, h are the dimensions of the tessellation grid. The square cell (x, y) in the tessellation grid can be split into two right triangles along one of the two diagonals. We use both options, starting at the bottom left cell with a split along the diagonal from top left to bottom right, and then alternate depending on whether the parity of x and y differs. As a result, our initial triangulation consists of four different types of triangles indicated with different colors in Fig. 4. These four types trivially correspond to four colors of a valid graph coloring of the triangle mesh. As we create an entry for each triangle in the triangle buffer, we assign a number $\in [0, 3]$ to the triangle type that we store as graph color. The purpose of these graph colors is task scheduling for parallel processing, which is explained in Section 8. It is already obvious from Fig. 4 that no triangles of the same color share an edge or the same adjacent triangles, so they can be updated concurrently without need for synchronization.

We store the two triangles of each cell (x, y) in the triangle buffer at indices $t_0 = 2(yw+x)$ and $t_1 = t_0+1$. The

vertex indices and adjacent triangle indices of each triangle can be obtained with simple index arithmetics. We assume that all 2D vertex positions of the initial triangulation are inserted row-wise starting from bottom left to top right. Then, the vertex indices of the two triangles are

$$\begin{pmatrix} y(w+1)+x \\ y(w+1)+x+1 \\ (y+1)(w+1)+x \end{pmatrix}, \begin{pmatrix} (y+1)(w+1)+x+1 \\ (y+1)(w+1)+x \\ y(w+1)+x+1 \end{pmatrix}$$

if x and y have the same parity, otherwise

$$\begin{pmatrix} (y+1)(w+1)+x \\ y(w+1)+x \\ (y+1)(w+1)+x+1 \end{pmatrix}, \begin{pmatrix} y(w+1)+x+1 \\ (y+1)(w+1)+x+1 \\ y(w+1)+x \end{pmatrix}.$$

The order of indices fulfills the requirements formulated in Section 4 that the triangles are counter-clockwise in 2D and that their hypothenuse is the segment from the second to the third vertex. Likewise, the adjacency information for each triangle is encoded as a triple of triangle indices refering to the neighboring triangles in counter-clockwise order such that the second adjacent triangle is the neighbor along the hypothenuse. The adjacent triangle indices are

$$\begin{pmatrix} 2w(y-1)+2x \\ t_1 \\ t_0-1 \end{pmatrix}, \begin{pmatrix} 2w(y+1)+2x+1 \\ t_0 \\ t_1+1 \end{pmatrix}$$

if x and y have the same parity, otherwise

$$\begin{pmatrix} t_0-1 \\ t_1 \\ 2w(y+1)+2x \end{pmatrix}, \begin{pmatrix} t_1+1 \\ t_0 \\ 2w(y-1)+2x+1 \end{pmatrix}.$$

For triangles along the borders of the grid, i.e., in the first and last rows and columns, some adjacent triangle indices might be out of bounds, as the triangles do not have neighbors along the borders. In these cases, we assign a triangle index -1 to indicate a missing neighbor for later processing. Likewise, we set the parent and child triangle indices of each initial triangle to -1 to indicate that it is a root triangle that has no child triangles assigned yet. Once all triangles have been stored, the triangle buffer contains a coarse discretization of the heightfield that could already be rendered. It only needs to be recreated when the extents of the heightfield change. Changes of the heightfield values do not invalidate the triangulation, but only require resampling of the existing vertex positions as discussed in Section 6.

6 Vertex Sampling

While the x and y coordinates in the 2D plane of all created triangle vertices follow from specified rules, the height component has to be acquired from the given heightfields through sampling. In our algorithm, this needs to happen in three different stages: (1) After the initial triangulation, (2) for each newly created vertex during subdivision, and (3) after the heightfield data or their sampling strategy change. For the sake of flexibility, we separate the tessellation from the heightfield data and their sampling strategy with the definition of a sampler interface that returns a single absolute height value for a given 2D world-space position. Through this interface, many existing sampling strategies can be used, ranging from simple bilinear interpolation within a regular grid to the approximation of scattered data with thin-plate splines. With the use of a per-vertex *heightfield index* introduced in Section 4, each vertex can be assigned a different sampler. This is particularly useful when using multiple heightfields, as in our use case illustrated in Fig. 1.

However, our sampler interface and therefore the proposed tessellation method are limited by the assumption that a heightfield value for a given 2D position is unique and constant until the heightfield data change, which the caching of vertex positions relies on. This excludes more advanced sampling strategies that integrate prefiltered heightfield data in a multi-resolution data structure such as mipmaps for filtered sampling, which is a useful approach to reduce popping effects and aliasing during subdivision. Such techniques provide data values at different levels of detail for the same position, of which the most suitable ones are selected based on factors such as the current camera perspective and data availability when rendering out of core, and are then blended for smooth transitions between levels of detail. Height values of all vertices might then change continuously and require resampling of all vertices, which is exactly what caching should prevent.

In our application, we mostly rely on C^1-continuous adaptive heightfield interpolation [6] suitable for heightfields defined on quadtrees. We assume that all heightfield data needed in the current frame are already and completely available in GPU memory. We display dynamic water heightfields generated by a GPU-based shallow water simulation on top of static digital elevation models. If the simulation time step changes by playback or manual navigation through time, usually all water heightfield values change at once, which is why it is the most efficient in our application to just loop over and resample all vertices whenever the corresponding heightfield changes. This means that in the worst case of a continuously changing heightfield, the benefit of vertex caching is completely lost, while triangle caching still remains effective.

7 Update of LoD State

In general, the purpose of view-dependent tessellation is to maintain a certain level of detail (LoD) with respect to the visible geometry at any time. As processing and rendering time of the heightfield increase with an increasing triangle count, the goal is to keep the number of triangles as low as possible and only subdivide where necessary to achieve a certain LoD. This LoD is subject to a defined metric. There are a variety of established LoD metrics, many of them considering the error made by approximating the continuous heightfield surface with piecewise linear triangles. An overview of suitable object-space and image-space error metrics is provided by Pajarola and Gobbetti [13], for example. The choice of the error metric is independent from the functioning of our tessellation pipeline, but it has to be suitable for the concrete application, as each metric requires a tradeoff between performance and visual quality. In our application, we want to minimize expensive heightfield sampling to maintain real-time performance. This is why we restrict ourselves to a simple LoD metric similar to that of Cantlay [33] that is solely based on the triangle density in screen space, independent of the underlying heightfield.

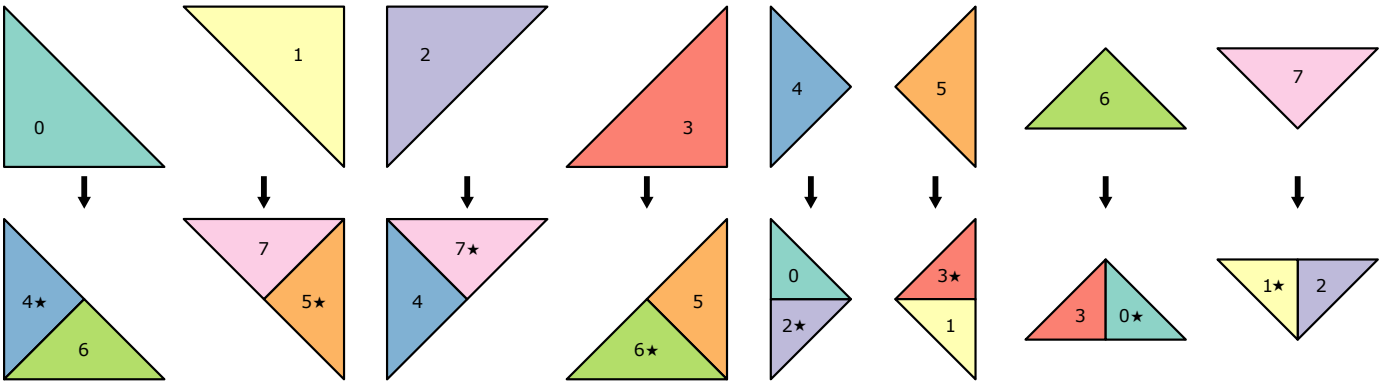

Fig. 5: Subdivision rules. Eight different triangle types can occur in a triangulation (top row). Each type can be subdivided into two triangles of other types (bottom row). Of these two triangles, the one we consider the first child triangle is marked with a star. Any recursive application of these rules to the initial triangulation leads to a valid 8-coloring of the plane.

Since triangles completely outside the camera's view frustum do not contribute to the scene, the first step is to exclude all these triangles from subdivision with a view frustum intersection test. For each remaining triangle, we calculate the screen-space lengths of its edges. A triangle needs to be subdivided if its longest edge in screen space is longer than a predefined maximum edge length in screen space. This maximum edge length, calculated from a user-defined value specified in pixels for better usability, defaults to 10 px. If, however, the longest edge in screen space is shorter than half the predefined maximum edge length, the triangle's parent triangle would better match the desired LoD, so the current triangle needs to be merged. In any other case, the triangle should be kept unchanged.

We apply these rules to each triangle in a compute shader and store the determined action—*subdivide*, *merge*, or *keep*—as *LoD state* with each triangle. This simple interface also allows for the use of any other LoD metric, provided it can assign one of the three mentioned actions to each triangle based on its current state. With the *heightfield index* stored for each triangle, it is also easily possible to use different LoD metrics or parameters for different heightfields, e.g., to reduce the LoD of a decoration heightfield.

8 Tessellation Loop

Once the LoD state of all triangles has been updated, the tessellation loop can be invoked to perform the required triangulation changes. The tessellation loop is the core of our algorithm that iteratively refines all triangles' LoD locally until either the target LoD or a termination criterion is reached. Within this loop, the triangles in the triangle buffer are modified with a sequence of compute shaders. In this section, we will explain the subdivision and merging shaders in detail, which will be extended with simple optimizations in Section 10. Triangles that require subdivision are first subdivided concurrently within each graph color group, while the individual graph color groups are processed sequentially. After subdivision, merging is applied similarly to all triangles that require it within each graph color group. If any triangles have changed, we now repeat this process unless a termination criterion is triggered.

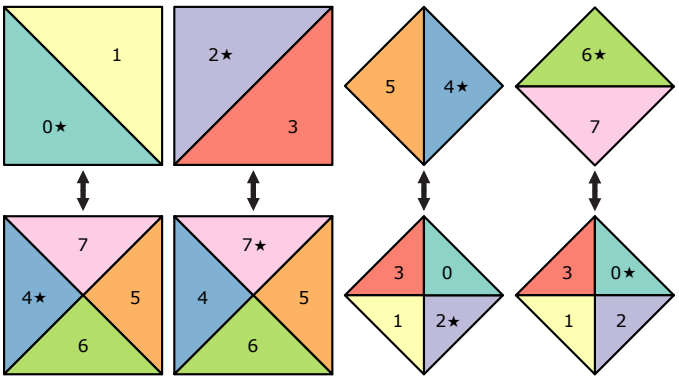

Fig. 6: Valid combinations of triangle types for subdivision and merging (top to bottom and vice versa). The star marks the triangle selected for subdivision or merging.

8.1 Subdivision Rules

In order to subdivide the triangles in a deterministic and thread-safe way, we need to define suitable subdivision rules. Given the very simple and regular initial triangulation with only four different types of triangles illustrated in Fig. 4, we only need to define equally simple rules to break up these triangles into smaller versions of the same type. Our rules illustrated in Fig. 5 achieve this in two steps. The initially given triangle types 0 to 3 corresponding to graph colors can be split along the median to the hypothenuse, i.e., the segment from the first vertex to the hypothenuse's centroid. This is commonly known as *longest-edge bisection*. As Fig. 5 shows, four new triangle types are obtained by this split, which we number from 4 to 7. Subsequently, these four types can again be split into two triangles each of the original types. In this illustration, a star marks the triangle that is its parent's first child triangle.

As mentioned before, we only allow subdivision of triangles that share a hypothenuse with triangles that are also supposed to be split. While the eight triangle types can be arranged in mutliple different ways, there is only one arrangement for each triangle type where this condition is met. This simplifies dealing with eight different triangle types to dealing with only four valid arrangements for subdivision, which are illustrated in Fig. 6. Even more,

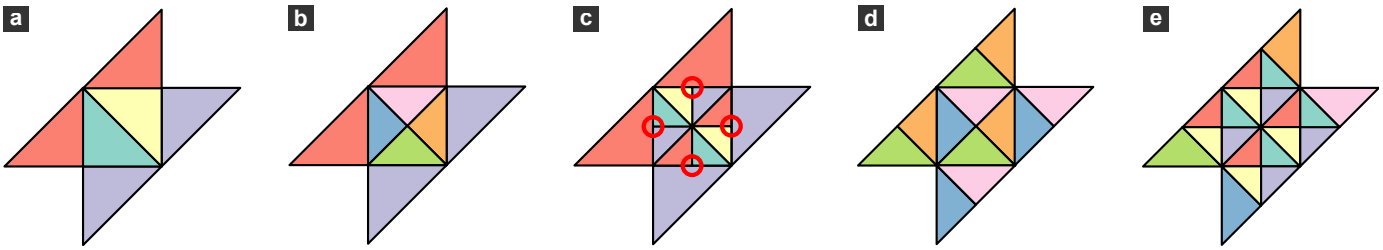

Fig. 7: Prevention of T-junctions. An arbitrary valid combination of triangle types (a) can be subdivided without involving the adjacent triangles (b). T-junctions would occur when subdividing the resulting triangles again (c), which is not possible because none of the resulting triangle arrangements of (b) is a valid combination for subdivision. Only after splitting the adjacent triangles (d), the inner triangles can be split again and introduce no T-junctions (e).

they pair each two triangle types with successive graph color values. This allows us to only consider triangles of graph colors 0, 2, 4, and 6 for subdivision, knowing that their neighbor along the hypothenuse either has a graph color value one higher than their own, or no subdivision is possible at all. These triangles are marked with a star in the top row in Fig. 6.

Adhering to this rule automatically guarantees a watertight triangulation by making *T-junctions* impossible. This is illustrated in Fig. 7 with the example of two triangles of types 0 and 1 and their adjacent triangles along the catheti (a). Both triangles share the hypothenuse, which means that splitting them does not involve the adjacent triangles. After subdivision, the hypothenuses of all four new triangles are the edges that have been the catheti before, which they share with the adjacent triangles (b). T-junctions (marked with red circles) could only occur if these triangles were split again without any regard of the adjacent triangles (c). However, this is not possible as none of the arrangements in (b) is considered a valid combination of triangle types for subdivision. Subdivision in this state can only take place in the adjacent triangles if necessary and possible (d). Only if that is the case the inner triangles can be subdivided again (e). This also implies that the subdivision levels of adjacent triangles can only differ by at most one.

8.2 Subdivision

At this point, every triangle that requires subdivision according to its LoD state should be subdivided. However, this has to happen separated by graph color and according to the subdivision rules. This is why we loop over all triangles four times for the four different graph colors 0, 2, 4, and 6, each time only considering triangles of the respective graph color. For each triangle t_0 of that graph color and its neighbor along the hypothenuse t_1, we then perform the following checks to determine whether to continue with subdivision:

- The deleted flags are not set
- The graph color of t_1 is one higher than that of t_0
- The first child triangle indices are -1
- The second adjacent triangle of t_1 is t_0
- The LoD state of either triangle or any of their adjacent triangles is *subdivide*

The actual subdivision of t_0 and t_1 has four steps: Creating a new vertex at the center of the shared hypothenuse, creating four new child triangles, updating the information

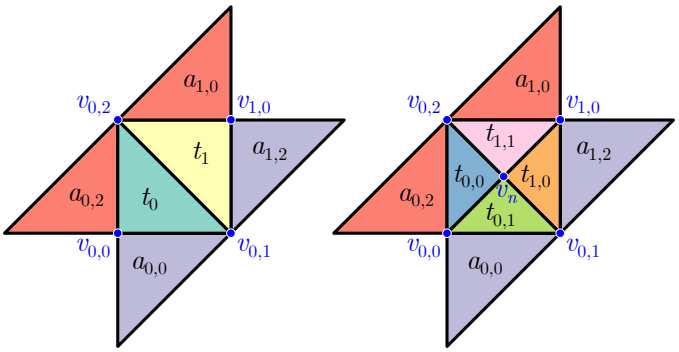

Fig. 8: Overview of adjacent triangles (black) and vertices (blue) of two parent triangles t_0 and t_1 (left) and their respective child triangles $t_{0,0}, t_{0,1}$ and $t_{1,0}, t_{1,1}$ (right).

of the two old triangles, and updating the adjacency information of the four adjacent triangles.

Creating the new vertex that all four child triangles share is conceptually simple, but can be quite expensive. While the x and y coordinates are simply calculated as the center of the segment between the second and third vertex of t_0, the z coordinate requires sampling the heightfield data with some interpolation method. Bilinear interpolation of data defined on a regular grid can be performed efficiently with the help of hardware texture filtering, but higher-order interpolation becomes more costly. For example, we use a third-order interpolation that handles both regular and adaptive grids [6], which requires a significant number of memory accesses. A benefit of our method is that vertex positions will be cached such that created vertices only need to be resampled if the heightfield data change. This legitimates the use of more expensive interpolation. The new vertex is added to the vertex buffer at the next free index, which is obtained as described in Section 4.

The four new triangles $t_{0,0}, t_{0,1}, t_{1,0}, t_{1,1}$—two children of t_0 and t_1 each—are assigned the parent index of the respective parent triangle and inherit the value of the parent's heightfield index. The graph colors are assigned according to the rules illustrated in Fig. 5. Since the new triangles have a different size, a new LoD state is assigned to each triangle according to the LoD function introduced in Section 7. Due to the consistent ordering, the assignment of the new adjacent triangle indices and vertex indices is straightforward. An example in Fig. 8 illustrates

these assignments. Let v_n be the index of the newly created vertex, and $(v_{0,0}, v_{0,1}, v_{0,2})$, $(v_{1,0}, v_{0,2}, v_{0,1})$ the vertex indices of t_0 and t_1, respectively. Then, the vertex indices of the new triangles are $(v_n, v_{0,2}, v_{0,0})$, $(v_n, v_{0,0}, v_{0,1})$, $(v_n, v_{0,1}, v_{1,0})$, and $(v_n, v_{1,0}, v_{0,2})$. Analogously, for the adjacent triangles $(a_{0,0}, t_1, a_{0,2})$ and $(a_{1,0}, t_0, a_{1,2})$ of t_0 and t_1, respectively, the adjacent triangles of the new triangles are $(t_{1,1}, a_{0,2}, t_{0,1})$, $(t_{0,0}, a_{0,0}, t_{1,0})$, $(t_{0,1}, a_{1,2}, t_{1,1})$, and $(t_{1,0}, a_{1,0}, t_{0,0})$.

After the new triangles are stored, we assign the indices of triangles $t_{0,0}$ and $t_{1,0}$ as first child triangle indices to triangles t_0 and t_1, respectively. Finally, we update the adjacency information of the adjacent triangles to reference the newly created ones instead of their parents. Each of the adjacent triangles $a_{0,0}$, $a_{0,2}$, $a_{1,0}$, and $a_{1,2}$ contains adjacency information as a triple of triangle indices, of which one value equals the index of the parent triangle t_0 or t_1. Since these triangles can share any of their edges with the parent triangle, the component index of the parent triangle's index in the triple is only known at runtime, so we cannot define static index assignments as for the other values. Instead, we determine the component index $i \in [0, 2]$ dynamically by comparing the given values to the indices of t_0 and t_1, respectively. Let $a^i_{0,0}$ be the ith component of the adjacency information of adjacent triangle $a_{0,0}$. If $a^i_{0,0}$ equals t_0, we assign $t_{0,1}$ to it, and $t_{0,0}$ to $a^i_{0,2}$, respectively. Analogously, we assign $t_{1,1}$ to $a^i_{1,0}$ and $t_{1,0}$ to $a^i_{1,2}$ if they equal t_1.

From this last step, it becomes obvious why we use task scheduling with graph colors to prevent concurrent processing of neighboring triangles. If neighboring triangles were subdivided at the same time, new triangles in a thread could receive obsolete adjacency information from their parents, referring to triangles that were also subdivided instead of to their children. This would corrupt the entire triangulation. Our approach avoids this by running the subdivision on independent groups of triangles in four separate compute shader dispatches with explicit synchronization inbetween to wait for completion of the previous operations. Another approach would be to subdivide triangles concurrently, but use a separate triangle buffer to store partial subdivision results, which are then merged together to a new triangle buffer in a second pass. This, however, comes at the cost of doubling the GPU memory consumption. A third approach would be to handle low-level synchronization in the subdivision shader itself to prevent reading of obsolete adjacency data of concurrently processed triangles with inter-thread communication. While this would most likely increase the runtime performance, it comes at the cost of a more complex and error-prone implementation that requires careful consideration in future work.

8.3 Merging

Analogous to subdivision, merging of triangles requires a thread-safe way of updating the adjacency information of a triangle's adjacent triangles, meaning that we again need to group merging threads by the triangles' graph colors. The main task of merging is to remove four child triangles created by a previous subdivision and reinstate their two parent triangles. This means that the operation only has to run once for every four triangles. It does not matter which of the four triangles we select for this, as long as the selection is consistent. We select the triangle that is the first child of the parent triangle with the lower graph color. These triangles are marked with a star in the bottom row in Fig. 6. From this follows that merging needs to be performed for triangles with the graph colors 0, 2, 4, and 7. We again loop over all triangles four times for the four different graph colors, each time only considering triangles of the respective graph color, and check whether merging should be performed. For every triangle t of the current graph color, we identify the parent triangle t_0, the parent's second adjacent triangle t_1, both parents' children $t_{0,0}$, $t_{0,1}$, $t_{1,0}$, and $t_{1,1}$, and then sequentially check the following:

- The deleted flags are not set
- The parent triangle index of t is not -1
- The first child triangle of parent t_0 is t
- The second adjacent triangle of t_1 is t_0
- The graph color of t_0 is lower than that of t_1
- The first child triangle index of all children is -1
- The LoD state of all children is *merge*
- The LoD state of neither t_0 nor t_1 is *subdivide*

Merging itself consists of four steps: Removing the four involved child indices from their parents, updating the parents' adjacency information, updating the adjacency information of the parents' neighbors, and removing the child triangles and their common vertex.

The reference of the child triangles is removed from their parents t_0 and t_1 by simply setting their first child triangle indices to -1. Updating their adjacency information is more involved, as it requires to assign the respective neighbors of the child triangles. Again, we refer to Fig. 8 for an annotated example. Let $(t_{1,1}, a_{0,2}, t_{0,1})$, $(t_{0,0}, a_{0,0}, t_{1,0})$, $(t_{0,1}, a_{1,2}, t_{1,1})$, and $(t_{1,0}, a_{1,0}, t_{0,0})$ denote the adjacent triangle indices of child triangles $t_{0,0}$, $t_{0,1}$, $t_{1,0}$, and $t_{1,1}$. Then, $(a_{0,0}, t_1, a_{0,2})$ and $(a_{1,0}, t_0, a_{1,2})$ are the new adjacent triangle indices assigned to parents t_0 and t_1, respectively, which reverses the assignment during subdivision (see Subsection 8.2).

In the next step, we also have to update the adjacency information of the adjacent triangles $a_{0,0}$, $a_{0,2}$, $a_{1,0}$, and $a_{1,2}$, which are the child triangles' respective neighbors along the hypothenuse. However, these adjacent triangles can again have an arbitrary triangle type and orientation, such that we need to dynamically find the component index $i \in [0, 2]$ of the value to replace. Let $a^i_{0,0}$ be the ith component of the adjacency information of adjacent triangle $a_{0,0}$. We assign t_0 to $a^i_{0,0}$ if it equals $t_{0,1}$, as well as to $a^i_{0,2}$ if that equals $t_{0,0}$. Analogously, we assign t_1 to $a^i_{1,0}$ if it equals $t_{1,1}$, as well as to $a^i_{1,2}$ if that equals $t_{1,0}$.

After the parent triangles have been reinstated in the triangulation, we can remove the obsolete child triangles and their common vertex from the respective buffers. We do this by adding the vertex index and the four triangle indices to the respective *free index buffers* described in Section 4. Finally, we set the *deleted flag* of the four triangles to ignore them in subsequent triangle selection passes.

The compute shader dispatches for the four graph colors are synchronized explicitly. After the last dispatch is completed, the triangulation is either one step closer to the target LoD or it has already been reached everywhere. If any triangle has been subdivided or merged, more changes to

the triangulation are potentially needed and that we should start another iteration. In this case, we first check for two criteria that lead to the termination of the tessellation loop: The number of triangles selected and processed in the current iteration was less than a user-defined minimum number of triangles, which we set to 100. Or a user-defined time budget for the entire tessellation loop, which we set to $5\,\mathrm{ms}$, has been exceeded. Both termination criteria weaken the strict LoD criterion to find a compromise between visual quality and processing time. Any unprocessed triangles will then be refined in the next frame.

9 Rendering

As all vertices have already been sampled on the heightfield during the initial tessellation or subdivision, no vertex sampling is required during rendering. However, since triangles frequently change during camera movements through subdivisions and merging, so do the surface normals, which leads to noticeable flickering of the illumination. A common technique to avoid this effect when using adaptive tessellation is to reconstruct the heightfield normals from the central differences of sampled height values per fragment. This results in a temporally stable illumination. Finally, for animated water heightfields, we use vertex displacement for animated waves and water surface shading based on a velocity field [6]. An example result of the visualization in our system is shown in Fig. 1.

10 Optimizations

In this section, we provide crucial runtime optimizations to the naïve implementation outlined in the previous sections. In particular, we address the SIMD architecture of current GPUs that performs best without dynamic branching in shader code, and relies heavily on caching of fetched data. For implementation details, we refer to the implementation of our solution in the supplemental material of this paper.

10.1 Subdivision Triangle Selection

The triangle subdivision discussed in Subsection 8.2 is the most expensive step of our algorithm, because we need to dispatch a compute shader for each of the graph colors 0, 2, 4, and 6 and wait for their completion. Therefore we want to make this step as efficient as possible. For the responsible compute shader, this means to avoid all unnecessary rule checks and branching. We accomplish this by moving all checks to a preceding compute shader that selects all triangles that must and can be subdivided according to our rules. In this selection compute shader invoked for each existing triangle, we check whether all triangles involved in its potential subdivision exist, have no children, match one of the valid combinations depicted in Fig. 6, and have to be subdivided according to their own LoD state or that of their neighboring triangles. If a triangle passes all of these checks, we write its index into a buffer for its corresponding graph color with the help of an atomic counter. If any triangles are selected, subdivision is invoked with one thread for each of these triangles. Otherwise, we directly proceed to merging.

10.2 Merging Triangle Selection

Similar to subdivision, the merging compute shader has to be executed for four different graph colors in sequence. Again, waiting times between executions can be reduced by moving all checks for the triangle selection to a preceding compute shader. In this compute shader invoked for each existing triangle, we check whether the four triangles involved in a merge operation exist, are not root triangles, do not have children, match one of the valid combinations depicted in Fig. 6, and are flagged to be merged. Furthermore, the selected triangle has to conform to our rule of being the first child of the parent triangle with the lower graph color, and none of the two parents of the involved triangles must be flagged to be subdivided. If a triangle passes all of these checks, we write its index into a buffer for its corresponding graph color with the help of an atomic counter. Only for these triangles, the merging compute shader is then invoked.

10.3 Triangle Buffer Compaction

After tessellation, all triangles are scattered in the triangle buffer, including deleted triangles and parent triangles that should not be displayed. While it would be possible to directly render all triangles from this buffer and discard those flagged as deleted and those with child triangles, it is faster to filter the triangles in a separate step. Whenever the triangulation changes, we iterate over the buffer in a compaction compute shader and write each leaf triangle's triple of vertex indices to a compact vertex index buffer with the help of an atomic counter. With the result, we can render the triangulation as common indexed geometry in a highly efficient manner. If we are dealing with multiple heightfields that require different shading effects, we also evaluate the *heightfield index* of each triangle and maintain a different vertex index buffer for each heightfield, which we then render separately.

11 Evaluation

We evaluate the practicability of our solution concerning runtime performance and accuracy in comparison to previous work. While several CPU-based tessellation methods have similar features to our method such as watertightness, incremental updates, and unlimited levels of detail, performance typically lags behind purely GPU-based methods due to unavoidable CPU-GPU communication and less parallelization. This is why we only consider GPU-based approaches, which are recursive tessellation by Lee et al. [30], adaptive GPU tessellation with compute shaders by Khoury et al. [24], and cached adaptive tessellation with improved triangle encoding by Kerbl et al. [25]. For a fair comparison of the individual methods, we implemented them in a common framework using OpenGL that is available in the supplemental material of this paper. The methods of Khoury et al. and Kerbl et al. have been proposed with just one iteration per frame, such that the LoD of the visualization only converges towards the target LoD over several frames. We have extended both methods with a simple loop that repeats the tessellation process until the target LoD is reached within a single frame. For the performance benchmark, we

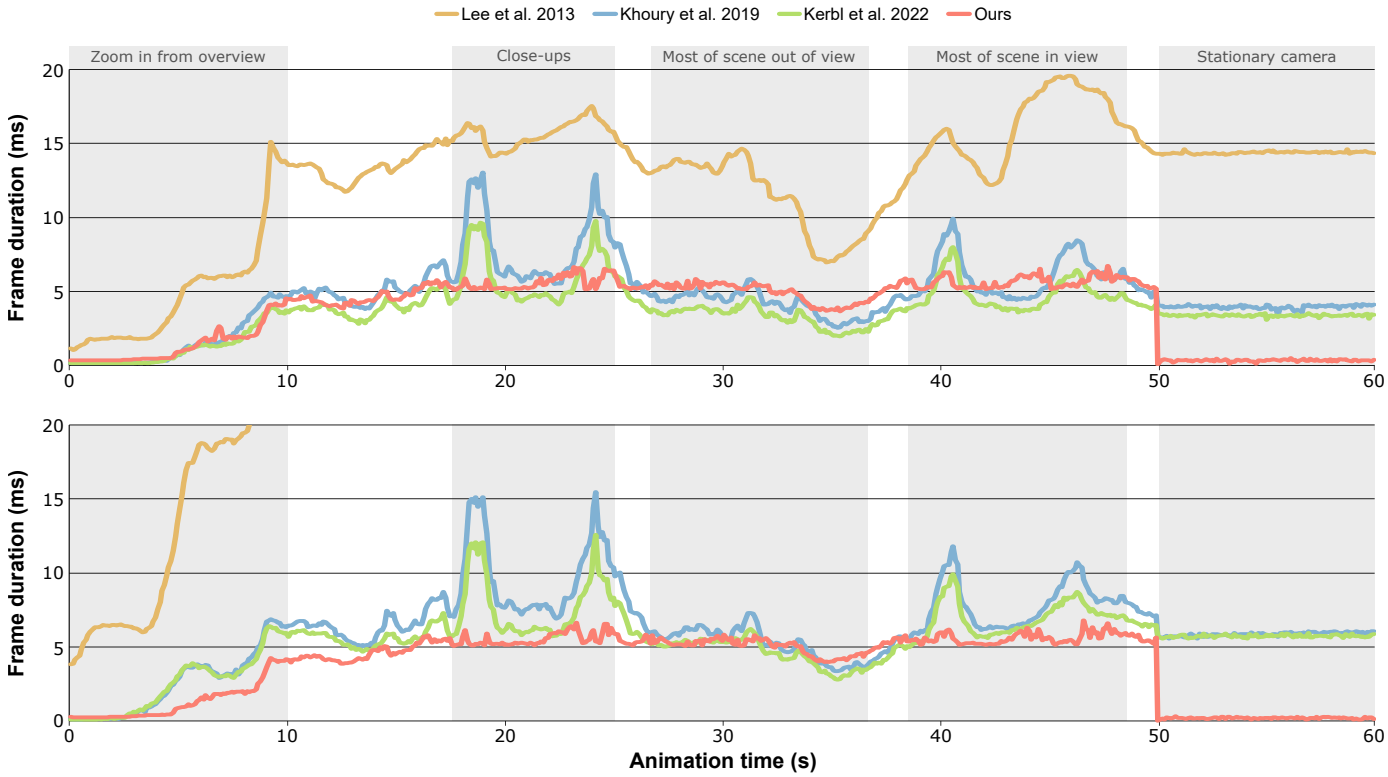

Fig. 9: Average frame durations of GPU-based tessellation methods with bilinear (top) and biquintic (bottom) interpolation during camera animation. Our method (red) achieves overall higher and more stable performance than the state of the art in terms of runtime (green), especially under heavy load with a substantial amount of vertex sampling.

have not replaced the LoD metric of both methods with our own, as we found that our LoD metric leads to slightly higher triangle counts than the proposed one, which would skew the performance comparison.

As a test scene, we use a procedural heightfield that is reconstructed using both fast bilinear and costly, but more accurate (as evaluated by Kidner [34]) biquintic interpolation. The heightfield is defined on a domain of $10\,\mathrm{km} \times 10\,\mathrm{km}$ with a cell size of $1\,\mathrm{m} \times 1\,\mathrm{m}$. For all methods, we set the minimum triangle edge length in world space to $0.1\,\mathrm{m}$ and the desired triangle size in screen space to $5\,\mathrm{px}$.

11.1 Runtime Performance

We measured the runtime performance on a system using an Nvidia Titan V GPU and an Intel Core i7-9700K CPU. The image resolution was $1920\,\mathrm{px} \times 1080\,\mathrm{px}$. For the measurements, we have prepared a camera animation along a fixed path that takes exactly $60\,\mathrm{s}$. We subdivided this time range into 600 intervals of $100\,\mathrm{ms}$ each, calculated the average frame duration within each of these intervals, and averaged these results over five animation runs each. We omitted the first interval to exclude irrelevant calculations for the initialization of the methods. The resulting average frame durations are provided in Fig. 9 for bilinear (top) and biquintic (bottom) interpolation. Our implementation considered here contains all optimizations described in Section 10 and achieves a frame duration of $3.7\,\mathrm{ms}$ (bilinear) and $3.8\,\mathrm{ms}$ (biquintic) averaged over all frames. Without the optimizations, frame durations increase to an average of $14.0\,\mathrm{ms}$ and $14.4\,\mathrm{ms}$, respectively.

The method by Lee et al. (orange), which is the only tested method from previous work that provides watertight tessellation, performs worst, because it has to generate the entire triangulation from scratch every frame. Due to space constraints, we cropped the frame duration plot of this method using biquintic interpolation (Fig. 9, bottom), which peaked at $65\,\mathrm{ms}$. Among the other three methods, our method (red) has the most stable frame duration without exhibiting any significant spikes under heavy load, in contrast to the methods of Khoury et al. (blue) and Kerbl et al. (green). This is in part due to the pre-defined time budget for the tessellation loop set to $5\,\mathrm{ms}$, and in part due to vertex caching, which makes our method largely independent of the employed vertex sampling strategy. This becomes especially clear in the comparison of the two plots, in which the frame durations of our method are almost identical despite the considerably higher cost of biquintic interpolation. In contrast, the frame durations of the other incremental methods peak at more than double the frame duration of our method, as the memory-efficient triangle encoding of these two methods comes at the cost of resampling all vertices of the cached triangulation each frame. A further benefit of our method is the fast processing with a stationary camera, as tested in the last $10\,\mathrm{s}$ of the animation. Then, the entire tessellation loop and triangle buffer compaction are skipped, leaving only the rendering of the compact triangle buffer. This results in a very high frame rate for interactive tasks such as sketching or object manipulation in 3D.

TABLE 1: Evaluation of Triangulation Quality

	Triangle size	# Triangles in mio.		RMSE in m	
		Max.	Avg.	Max.	Avg.
Lee et al.	10 px	3.15	1.35	9.70	3.44
	5 px	10.75	4.99	9.73	2.07
	3 px	25.55	10.58	5.72	1.17
Kerbl et al.	10 px	1.24	0.23	9.20	3.25
	5 px	4.79	0.79	9.18	2.15
	3 px	13.11	1.56	4.09	1.00
Ours	10 px	0.34	0.12	9.20	2.93
	5 px	1.37	0.47	8.98	1.77
	3 px	3.82	1.15	3.46	0.89

11.2 Accuracy of Surface Approximation

To evaluate the quality of the triangulation produced by our tessellation method in comparison to the others, we recorded the number of generated triangles and the deviation of the triangulation from the ground truth each frame during the camera animation. The deviation is estimated from the root mean square error (RMSE), where the error is the per-fragment difference between the height of the rasterized, visible portion of the triangulation and the heightfield surface obtained from the continuous noise function. We aggregate both the number of generated triangles and the RMSE with maximum and average operators over the entire camera animation to obtain average and peak values, which are provided in Table 1. For a fair comparison, we have integrated our LoD metric into the method by Kerbl et al. We omit the method by Khoury et al. here, as the triangulations of the two methods are equivalent.

It can be seen that our tessellation consistently generates the lowest number of triangles while also having the lowest deviation from the ground truth, independent of the desired triangle size. As we used the same LoD metric for all three tessellation methods, we attribute the varying accuracy to the different employed subdivision rules. Hardware tessellation as used in the method by Lee et al. operates on patches subdivided according to inner patch and edge tessellation factors, which do not allow for fine-grained LoD control of individual triangles within the patch. In order to guarantee a maximum triangle size in screen space, many triangles in the patch are over-tessellated as a result, which is reflected by the high number of generated triangles and, in succession, the poor runtime performance. The method by Kerbl et al. produces a triangulation of much higher quality, allowing for a more accurate approximation of the heightfield surface with significantly less triangles. However, the maximum number of triangles shows a five- to eightfold increase over the average during some sequences in the animation, which we identified as the extreme close-ups. At the same time, the high number of triangles does not reduce the approximation error. We suspect that this behavior is caused by disregarding triangle visibility during subdivision and merging, which prevents merging of invisible triangles against the LoD metric. In our method, over-tessellation is significantly reduced, as our subdivision rules consider triangle visibility and allow for the subdivision of almost any pair of triangles to an arbitrary LoD. This allows us to refine triangles precisely where needed.

In summary, our method exhibits stable real-time performance competitive with the fastest non-watertight methods from previous work, while at the same time providing a highly accurate approximation of the heightfield surface without visual artifacts caused by T-junctions. The more complex the used surface reconstruction is, the more vertex caching reduces the cost of tessellation and rendering, which allows our method to outperform the state of the art.

12 Conclusions and Future Work

We presented a novel method for the watertight tessellation of heightfields that offers steady real-time performance and high reconstruction accuracy. Our solution addresses the special requirements of geoinformation systems regarding complex heightfield reconstruction and interactivity and can be integrated into many existing systems as drop-in replacement. A current limitation of our method is the assumption that heightfield values of vertex positions are unique and depend only on changes in heightfield data, not on the triangle LoD or camera perspective. This is essential for vertex caching, but precludes the immediate use for prefiltered heightfields and many out-of-core approaches that stream and blend between data of different LoDs at runtime, which we will address in future work.

By relying on explicit triangle adjacency information, we have solved the problem of independent triangle splitting that causes T-junctions, but have introduced a scheduling problem to avoid read-write conflicts between triangles trying to update the same adjacency information concurrently. Our solution of task scheduling by graph coloring is well-established for distributed systems, but not necessarily optimal for GPUs. It requires expensive synchronization between the compute shader dispatches for the individual graph colors. A single dispatch with fine-grained synchronization between threads could further improve the runtime performance, which we will also investigate in future work.

Acknowledgments

We thank Jonathan Dupuy as well as Linus Horváth and Bernhard Kerbl for making the source code of their methods available to us. VRVis is funded by BMK, BMDW, Styria, SFG, Tyrol and Vienna Business Agency in the scope of COMET - Competence Centers for Excellent Technologies (879730) which is managed by FFG.

References

[1] A. Tevs, I. Ihrke, and H.-P. Seidel, "Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering," in *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, ser. I3D '08. ACM, 2008, pp. 183–190.

[2] C. Dick, J. Krüger, and R. Westermann, "GPU ray-casting for scalable terrain rendering," in *Proceedings of Eurographics 2009 - Areas Papers*. Eurographics Association, 2009, pp. 43–50.

[3] E.-S. Lee, J.-H. Lee, and B.-S. Shin, "A bimodal empty space skipping of ray casting for terrain data," *The Journal of Supercomputing*, vol. 72, no. 7, pp. 2579–2593, 2016.

[4] T. Cheng, "Accelerating universal kriging interpolation algorithm using CUDA-enabled GPU," *Computers & Geosciences*, vol. 54, pp. 178–183, 2013.

[5] X. Li, F. Chen, H. Kang, and J. Deng, "A survey on the local refinable splines," *Science China Mathematics*, vol. 59, no. 4, pp. 617–644, 2016.
[6] D. Cornel, A. Buttinger-Kreuzhuber, A. Konev, Z. Horváth, M. Wimmer, R. Heidrich, and J. Waser, "Interactive visualization of flood and heavy rain simulations," *Computer Graphics Forum*, vol. 38, no. 3, pp. 25–39, 2019.
[7] D. Cornel, Z. Horváth, and J. Waser, "An attempt of adaptive heightfield rendering with complex interpolants using ray casting," *arXiv e-prints*, vol. abs/2201.10887, pp. 1–9, 2022.
[8] C. Johanson, "Real-time water rendering: Introducing the projected grid concept," Master's thesis, Department of Computer Science, Lund University, 2004.
[9] F. Löffler, S. Rybacki, , and H. Schumann, "Error-bounded GPU-supported terrain visualisation," in *WSCG Communication Papers Proceedings*. UNION Agency, 2009, pp. 47–54.
[10] Y. Livny, N. Sokolovsky, T. Grinshpoun, and J. El-Sana, "A GPU persistent grid mapping for terrain rendering," *The Visual Computer*, vol. 24, no. 2, pp. 139–153, 2008.
[11] P. Houska, "Improved persistent grid mapping," Master's thesis, TU Wien, 2020.
[12] J. Luo, G. Ni, and K. Luo, "Projected displaced-mesh: A fast terrain visualization algorithm," in *International Conference on Image Analysis and Signal Processing*. IEEE, 2009, pp. 226–232.
[13] R. Pajarola and E. Gobbetti, "Survey of semi-regular multiresolution models for interactive terrain rendering," *The Visual Computer*, vol. 23, no. 8, pp. 583–605, 2007.
[14] F. Losasso and H. Hoppe, "Geometry clipmaps: Terrain rendering using nested regular grids," *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 769–776, 2004.
[15] A. Asirvatham and H. Hoppe, "Terrain rendering using GPU-based geometry clipmaps," in *GPU Gems 2*, M. Pharr and R. Fernando, Eds. Addison-Wesley, 2004, ch. 2, pp. 27–45.
[16] J. Bösch, P. Goswami, and R. Pajarola, "RASTeR: Simple and Efficient Terrain Rendering on the GPU," in *Eurographics 2009 – Areas Papers*, D. Ebert and J. Krüger, Eds., 2009, pp. 1–8.
[17] T. Boubekeur and C. Schlick, "A flexible kernel for adaptive mesh refinement on GPU," *Computer Graphics Forum*, vol. 27, no. 1, pp. 102–113, 2008.
[18] C. Dyken, M. Reimers, and J. Seland, "Semi-uniform adaptive patch tessellation," *Computer Graphics Forum*, vol. 28, no. 8, pp. 2255–2263, 2009.
[19] L. Hu, P. V. Sander, and H. Hoppe, "Parallel view-dependent level-of-detail control," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 5, pp. 718–728, 2010.
[20] O. Ripolles, F. Ramos, A. Puig-Centelles, and M. Chover, "Real-time tessellation of terrain on graphics hardware," *Computers & Geosciences*, vol. 41, pp. 147–155, 2012.
[21] N. Tatarchuk, J. Barczak, and B. Bilodeau, "Programming for real-time tessellation on GPU," AMD, Inc., Tech. Rep., 2009.
[22] X. Bonaventura, "Terrain and ocean rendering with hardware tessellation," in *GPU Pro 2*, W. Engel, Ed. A K Peters, 2011, pp. 3–14.
[23] H. Kang, H. Jang, C.-S. Cho, and J. Han, "Multi-resolution terrain rendering with GPU tessellation," *The Visual Computer*, vol. 31, no. 4, pp. 455–469, 2015.
[24] J. Khoury, J. Dupuy, and C. Riccio, "Adaptive GPU tessellation with compute shaders," in *GPU Zen 2: Advanced Rendering Techniques*, W. Engel, Ed. Black Cat Publishing, 2019, ch. 1, pp. 3–16.
[25] B. Kerbl, L. Horváth, D. Cornel, and M. Wimmer, "An Improved Triangle Encoding Scheme for Cached Tessellation," in *Eurographics 2022 – Short Papers*, 2022, pp. 53–56.
[26] M. Englert, "Using mesh shaders for continuous level-of-detail terrain rendering," in *SIGGRAPH Talks*. ACM, 2020, pp. 44:1–2.
[27] B. Santerre, M. Abe, and T. Watanabe, "Improving GPU real-time wide terrain tessellation using the new mesh shader pipeline," in *Nicograph International*, vol. 1, 2020, pp. 86–89.
[28] H. Jang and J. Han, "GPU-optimized indirect scalar displacement mapping," *Computer-Aided Design*, vol. 45, no. 2, pp. 517–522, 2013.
[29] H. Kang, Y. Sim, and J. Han, "Terrain rendering with unlimited detail and resolution," *Graphical Models*, vol. 97, pp. 64–79, 2018.
[30] H. Lee, Y. Jeong, and S. Lee, "Recursive tessellation," in *SIGGRAPH Asia Posters*. ACM, 2013, p. 16:1.
[31] J. Dupuy, "Concurrent binary trees (with application to longest edge bisection)," *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 3, no. 2, 2020.
[32] D. J. A. Welsh and M. B. Powell, "An upper bound for the chromatic number of a graph and its application to timetabling problems," *The Computer Journal*, vol. 10, no. 1, pp. 85–86, 1967.
[33] I. Cantlay, "DirectX 11 terrain tessellation," NVIDIA Corporation, White Paper, 2011.
[34] D. B. Kidner, "Higher-order interpolation of regular grid digital elevation models," *International Journal of Remote Sensing*, vol. 24, no. 14, pp. 2981–2987, 2003.

Daniel Cornel works as a postdoctoral researcher in the Integrated Simulations Group at VRVis that he joined in 2014. He is the lead rendering engine developer for Visdom, a decision support system for flood management. He completed his PhD on the topic of "Interactive Visualization of Simulation Data for Geospatial Decision Support" in 2020.

Silvana Zechmeister is a doctoral student and researcher in the Integrated Simulations Group at VRVis since 2020. As part of the rendering team for Visdom, she is interested in interactive visualization techniques for complex 3D scenes, including the dynamically changing visualization of vector data and of embedded labels.

Eduard Gröller is Professor at the Institute of Visual Computing & Human-Centered Technology (VC&HCT), TU Wien, where he is heading the Research Unit of Computer Graphics. He is a key researcher of the VRVis, which performs applied research in visualization, rendering, and visual analysis. Dr. Gröller is Adjunct Professor of Computer Science at the University of Bergen, Norway. His research interests include computer graphics, visualization, and visual computing.

Jürgen Waser is the head of the Integrated Simulations Group at VRVis. Since completion of his master studies in Technical Physics, he has been working in the area of flood simulation and geospatial visualization. His PhD thesis demonstrates Visdom as a decision support system for flood management. He is a core developer and co-founder of the Visdom framework.