Terrender: A Web-Based Multi-Resolution Terrain Rendering Framework

Julian A. Croci
julian.croci@bluewin.ch
University of Zurich
Zurich, Switzerland

Alireza Amiraghdam
amiraghdam@ifi.uzh.ch
University of Zurich
Zurich, Switzerland

Renato Pajarola
pajarola@ifi.uzh.ch
University of Zurich
Zurich, Switzerland

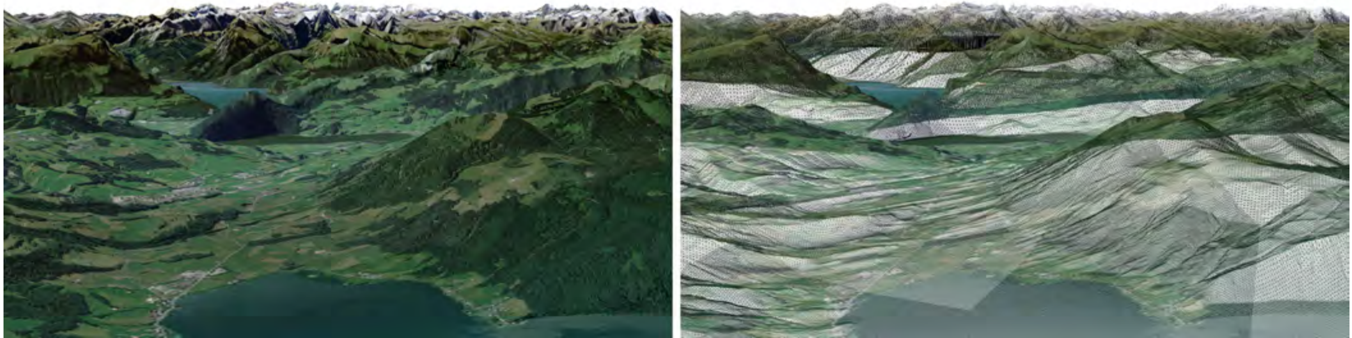

Figure 1: Using Terrender for real-time rendering of Switzerland based on high-resolution height and color data (left) and the corresponding wireframe (right). The triangulation density is adapted to the distance and roughness of the terrain.

ABSTRACT

Terrain rendering is a fundamental requirement when visualizing 3D geographic data in various research, commercial or personal applications such as geographic information systems (GIS), 3D maps, simulators, and games. It entails handling large amounts of data for height and color as well as high-performance algorithms that can benefit from the parallel rendering power of GPUs. The main challenge is (1) to create a detailed renderable mesh using a fraction of the data that is most relevant to a specific camera position and orientation, and (2) to update this mesh in real time as the camera moves while keeping the transition artifacts low. Many algorithms have been proposed for adaptive adjustment of the level of detail (LOD) of large terrains. However, the existing web-based terrain rendering frameworks do not use state-of-the-art algorithms. As a result, these frameworks are prone to classic shortcomings of simpler terrain rendering algorithms such as discontinuities and limited visibility. We introduce a novel open-source web-based framework for rendering high quality terrains with adaptive LOD: *Terrender*. Terrender employs RASTeR, a modern LOD-based terrain rendering algorithm, while running smoothly with a limited bandwidth on all common web browsers, even on mobile devices. Finally, we present a thorough analysis of our system's performance when the camera moves on a predefined trajectory. We also compare its performance and visual quality to another well-known framework.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
Web3D '22, November 2–4, 2022, Evry-Courcouronnes, France
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9914-2/22/11.
https://doi.org/10.1145/3564533.3564567

CCS CONCEPTS

• **Human-centered computing** → **Geographic visualization**; **Visualization systems and tools**; • **Computing methodologies** → *Rasterization*; • **Theory of computation** → Computational geometry.

KEYWORDS

web-based, terrain rendering, open source, level of detail

ACM Reference Format:
Julian A. Croci, Alireza Amiraghdam, and Renato Pajarola. 2022. Terrender: A Web-Based Multi-Resolution Terrain Rendering Framework. In *The 27th International Conference on 3D Web Technology (Web3D '22), November 2–4, 2022, Evry-Courcouronnes, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3564533.3564567

1 INTRODUCTION

Terrain is a fundamental component of visualizing 3D geographic data in geographical information systems (GIS), 3D maps, and games. Virtual terrains can be generated and displayed based on height data. The area extent and the amount of detail affect the size of the data, which can be so large that techniques for managing and visualizing variable levels of detail (LOD) are required. This has been researched extensively and many solutions have been proposed. Nevertheless, implementing an efficient LOD-based terrain renderer in a web-based system remains demanding.

Therefore, frameworks such as CesiumJS [Cesium GS 2021] and ArcGIS [Environmental Systems Research Institute 2022] have been developed to be used by other developers to integrate a LOD terrain renderer quickly into a web-based system with minimal effort. However, despite the advanced features these frameworks provide in

other areas, their terrain rendering techniques are not state-of-the-art and are prone to classic shortcomings such as discontinuities and inefficient distribution of details.

In this paper, we introduce a novel open-source web-based multiresolution terrain renderer called *Terrender*, which can be used as a base for any geographic visualization system that relies on large and detailed data for terrain height and texture. Our contribution is a system offering the full pipeline for preprocessing the raw input data, setting up a server for distributing the preprocessed data, and a web client that generates and renders the terrain mesh in real time using adaptively varying LOD. The displayed terrain eventually consists of a simple mesh which can be rendered using the WebGL shader pipeline. Due to this simplicity, it is suitable for applying a wide variety of rendering extension techniques.

2 RELATED WORK

Focusing on mesh based real-time terrain rendering, one can organize terrain data as Triangulated Irregular Networks (TINs) [Fowler and Little 1979], regular grids or hybrid methods such as e.g. [Pajarola et al. 2002; Paredes et al. 2016]. While TINs and hybrid methods can better optimize mesh sizes for a targeted geometric accuracy, regular or semi-regular grid models offer advantages in terms of tiling and downsampling, as well as compression and rendering performance. Due to the simplicity of working with regular grids and their resemblance with images, we use regular grids as a basis for Terrender and commonly refer to them as height textures. See also the surveys on multiresolution terrain triangulation and rendering [De Floriani et al. 1996; Pajarola and Gobbetti 2007] as well as digital Earth systems [Mahdavi-Amiri et al. 2015]. Furthermore, due to the limited scope of this paper we do not further discuss aspects of data compression, for this see other prior methods such as e.g. [Dick et al. 2009; Gerstner 2003a; Gobbetti et al. 2006].

The LOD of a terrain is adjusted using scene and terrain dependent error metrics. To have a continuous LOD terrain triangulation, no hanging vertices (i.e. cracks or T-junctions) should be present in the generated mesh. The continuity of the terrain can be maintained in different ways [Hill 2002], e.g., cracks can be covered by additional triangles, vertical triangles can be added to the edges of the tiles so that the cracks are not visible, or the background color can be used to make the cracks less visible [Holst 2004]. However, the drawback of these methods is that they either only hide the cracks or require further analysis of the data at run time.

Bintrees can be used for managing the triangles formed by the vertices in regular grids, which enable us to create lower LOD terrains. However, the challenge is that splitting a triangle, e.g. for increasing the LOD, not only affects the direct triangles in the bintree but requires a series of further splits in neighboring triangles that can be far in the tree. Looking up neighboring triangles can be explicitly tracked [Duchaineau et al. 1997] or can be avoided, e.g., by creating the LOD tree bottom up [Pajarola 1998] which causes the algorithm to need all the vertices at run time. Another approach to avoid the explicit look-up of neighbors is to make the LOD error metric inherently guarantee that if one node is split, its neighbor is automatically split as well. The saturation condition term was introduced [Ohlberger and Rumpf 1999] as a property of error metrics that can guarantee this implicitly. An efficient way to ensure the saturation condition for camera-angle-dependent error metrics, such as viewing distance and view culling, is using the bounding shapes that enclose all possible triangles that can be affected by a particular triangle split. While spherical bounding shapes can be used [Cignoni et al. 2003; Lindstrom and Pascucci 2001, 2002], an optimized octagonal alternative was proposed in [Gerstner 2003b]. For Terrender, we use this *octagon metric* that encloses only the volume of interest and can still be evaluated efficiently in real time.

Algorithms for terrain mesh generation and rendering need to be designed considering the GPUs' ability of processing groups of elements in parallel. In addition, GPUs have specific memory and require efficient memory management strategies. Thus instead of focusing on single vertices and triangles, GPU performance can better be optimized by selecting the LOD in terms of triangulated terrain patches [Bösch et al. 2009; Cignoni et al. 2003; Lario et al. 2003; Pomeranz 2000]. The RASTeR [Bösch et al. 2009] algorithm defines the basic LOD terrain management component in Terrender.

With larger terrain datasets where the data cannot be stored locally, the data has to be streamed to the client. A subset of multiresolution techniques are suitable for client-server architectures [Bettio et al. 2007; Gobbetti et al. 2006]. In client-server approaches, data compression is used to reduce the transfer bandwidth and memory footprint. For example, in [Bettio et al. 2007], data compressed by a customized lossy wavelet compression algorithm is sent to the client to be decompressed on the CPU.

CesiumJS [Cesium GS 2021] and ArcGIS [Environmental Systems Research Institute 2022] are two major frameworks for GIS applications in web browsers. ArcGIS is an entirely commercial product, while parts of CesiumJS (especially the frontend) are open source. Some open-source alternatives exist for the commercial parts of CesiumJS [Zwaagstra and rumicuna 2016; Zwaagstra et al. 2018]. Apart from these two frameworks, not many others exist. Three different GIS software products are compared in [Mat et al. 2009] that precompute a file which can then be shown in the browser using a viewer that explicitly needs to be installed. A GIS browser implementation using WebGL was proposed in [Feng et al. 2011]. While WebGL has the advantages that it does not need to be installed, this system does not offer any form of LOD selection. A similar approach without LOD selection, presented in [Larrick et al. 2020], is mostly concerned with the required backend. Recently, a system that uses CesiumJS is proposed to bring real-life satellite image data together with a digital elevation model [Wan et al. 2021]. None of these works offer or use a framework that employs a more sophisticated state-of-the-art algorithm as Terrender does.

3 TERRENDER

Terrender is composed of three units as shown in Fig. 2. (1) The preprocessor takes the raw height and optional color data as input and creates a multiresolution pyramid data structure. Each layer includes a tiled representation of the original data in a specific resolution. The preprocessor can optionally calculate the geometry errors used by the frontend when updating the terrain's mesh. (2) The backend is responsible for serving the frontend with the geometry error and the tiles. (3) The frontend creates and updates a variable-LOD mesh based on the RASTeR algorithm [Bösch et al. 2009] using scene and terrain dependent error metrics such as the

octagon metric [Gerstner 2003b]. It requests the relevant tiles for the current state of the mesh from the backend and sends the mesh and data to the WebGL renderer.

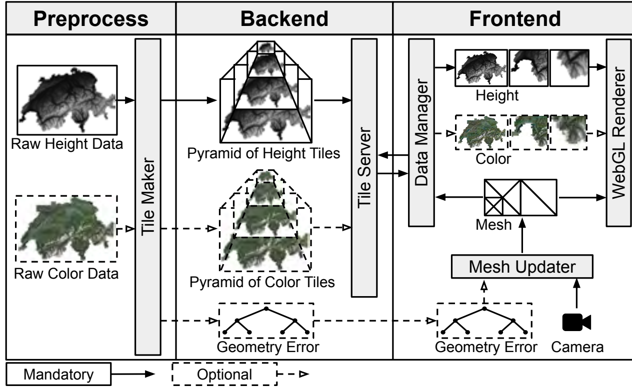

Figure 2: An overview of the full pipeline of Terrender.

3.1 RASTeR

The RASTeR algorithm [Bösch et al. 2009] influences how the data is preprocessed and defines the behavior of the frontend. RASTeR works by subdividing root triangles that span the whole terrain recursively until the desired level of detail is reached in specific areas dependent on the scene. The subdivision process is stored in a bintree. Each node of the bintree is associated with a triangulated triangle called K-patch making the tree significantly smaller as when single vertices were considered. The height and color data are stored in a quadtree. Each node of the quadtree is called a M-block and is associated with one or more nodes of the bintree. For a more detailed explanation see Appendix A and [Bösch et al. 2009].

3.2 Error Metrics

When creating or updating the bintree, Terrender uses three different error metrics to decide if a triangle T needs to be subdivided [Gerstner 2003b]: (1) *distance metric* denoted as $\mu_{dis}(T)$ generates an error value based on a predefined function of the distance from the triangle. (2) *culling metric* denoted as $\mu_{cul}(T)$ generates an error value based on whether the respective node is inside the view frustum. (3) *geometry metric* denoted as $\mu_{geo}(T)$ generates an error value based on the height difference that the triangle can cause, i.e. flat areas cause lower errors thus need less triangles. An example of a triangulation using each of these metrics applied separately to a specific camera location is depicted in Fig. 3(a-c). Finally, a total error value is calculated for a node using the formula

$$\mu(T) = \mu_{dis}(T) \times \mu_{cul}(T) \times \mu_{geo}(T) \quad (1)$$

An example triangulation using the combined error value is shown in Fig. 3(d).

In order to maintain the continuity of the mesh, two criteria must be fulfilled: Adjacent triangles sharing a common hypotenuse must be subdivided simultaneously and both triangles have to exist. The first criterion is achieved by making the error value dependent on the refinement point on the middle of the hypotenuse instead of the triangle itself. As two neighboring triangles share the same refinement point, they will thus both get the same error value.

We enforce the second criterion using the saturation condition [Ohlberger and Rumpf 1999]. For the distance and culling metric, we use the octagon metric [Gerstner 2003b] that is based on bounding shapes. Instead of using the refinement points in the error metric functions, it uses an octagon around the triangle. This octagon encompasses all other triangles affected by the current one. Therefore, no child can be subdivided without its parent being subdivided first. As the center of the octagon is the refinement point also the first criterion holds. The geometry error is calculated in a preprocess stage since the complete height data is not available at runtime. A bintree which stores an error value calculated using min-max bounds represents the geometry error. The resulting bintree is saturated by setting the error value of a node to the maximum error value of its children and its neighbor's children. We extended the usage of binary addresses [Gerstner 2003a] of nodes to find neighbors to also work with multiple side-by-side bintrees to support the use of non square terrains.

4 IMPLEMENTATION

In this section, we explain the implementation of the three units of Terrender shown in Fig. 2: Preprocess, Backend and Frontend.

4.1 Data Preprocessing

The data preprocessing consists of two main steps. First, the M-block quadtrees are created for the color and the height data according to the RASTeR algorithm. The quadtrees are created bottom up. The raw data is first tiled into squares with the desired M-block side length. Then, each four tiles are merged and downsampled to form the parent M-block tile in the next LOD. The tiles that only contain zero values are deleted to save space. At last, the tiles are converted to the desired format e.g. PNGs, TIFs or JPGs.

Lossy compression can not easily be used for height tiles since the values of the same vertices should match across different LODs. The precision of the data format of the tiles is matched with the original data. For instance, if the type of the height values is 32bit floating point, the four bytes of the height value are stored separately in four channels in RGBA format because a raster image format is easier to process in the frontend. It is also possible to keep them as TIFs, in this case other data types such as unsigned integers are supported.

Instead of using the whole height data, the frontend can calculate the geometry error by using the min-max bounds of the height values. Therefore, in the second step of the preprocess, we calculate these values for all nodes of the full bintree. After saturating the tree, we store the resulting tree in a file. The backend sends this file to the frontend when the program is loaded.

For calculating the min-max bounds, the bintree is created bottom up. First, the min-max bounds for all leaf triangles in the lowest level of the bintree are calculated. Then, the next levels are calculated one by one up to the root of the bintree. Depending on the K-patch size, the leaf triangles cover three or more height values. The min bound B_{min} for leaf nodes are calculated as

$$B_{min}(T) = min(H(T) \cup H(N(T))) \quad (2)$$

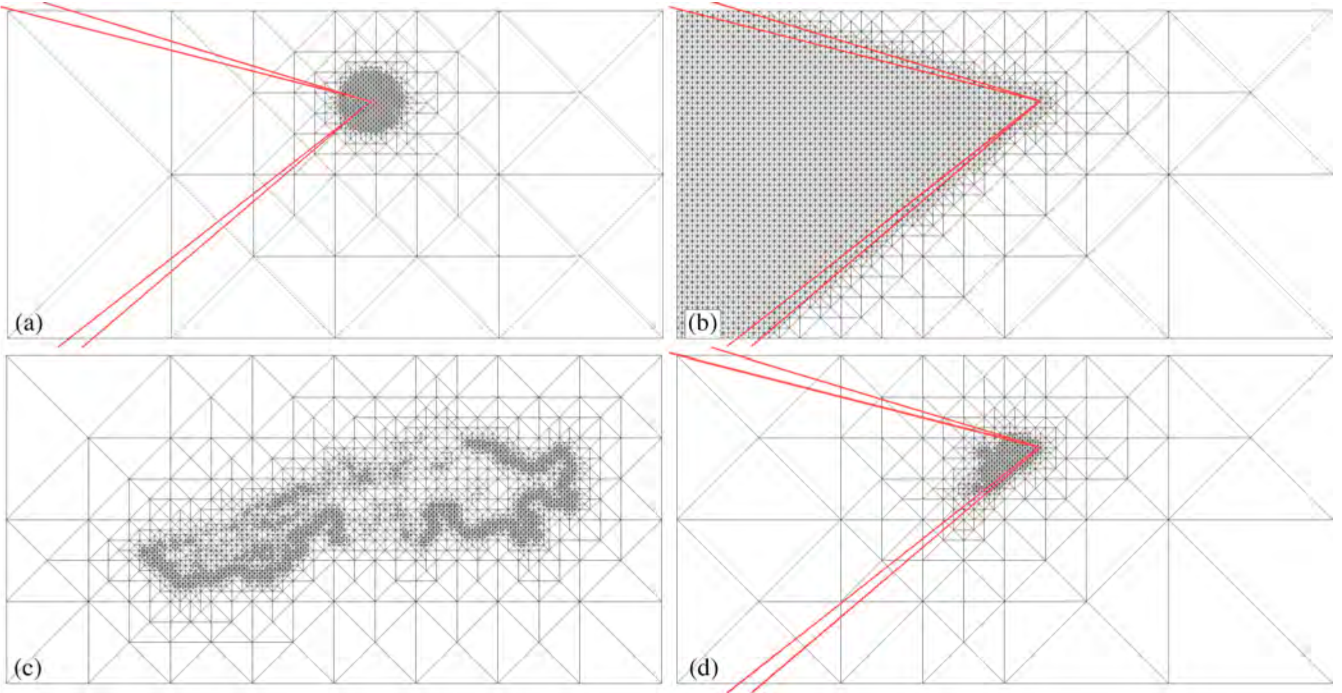

Figure 3: Three different metrics used in Terrender plus their combination: (a) Distance Metric, (b) Culling Metric, (c) Geometry Metric, (d) Combined Metrics. The view frustum is illustrated in red in (a),(b), and (d). (c) is independent of the view frustum.

where $H(T)$ returns the height values covered by triangle T and $N(T)$ returns the neighbor of triangle T. The min bound for the intermediate nodes are calculated as

$$B_{min}(T) = min\{B_{min}(C_l(T)), B_{min}(C_r(T)), B_{min}(C_l(N(T))), B_{min}(C_r(N(T)))\} \quad (3)$$

where $C_l(T)$ returns the left child and $C_r(T)$ returns the right child of triangle T. The neighbor of a triangle is looked up using the algorithm from [Gerstner 2003a]. Likewise, the max bound B_{max} is calculated for the leaf and intermediate nodes. Every two neighboring triangles have the same min-max bounds since $N(N(T)) = T$ and therefore $B_{min}(N(T)) = B_{min}(T)$. Finally, the subtrees with small min-max bounds are removed to reduce the output size at the cost of information loss. The default threshold is 0 to avoid the loss.

4.2 Backend

The backend of Terrender is a web server that provides the frontend with the code and the geometry error at the beginning. Afterwards, it serves the frontend with the height and color tiles that the frontend requests. The implementation of this web server is independent of the frontend. We implemented a simple server in Node.js.

4.3 Frontend

As a web-based application, Terrender is limited to what the browser allows. Therefore, unlike a stand-alone real-time graphics program, it can not run a render loop. Instead, the rendering process, which repeatedly produces the frames, is triggered by the browser. The whole process is shown in Figure 4 and is explained in this section.

4.3.1 Parallel Processing. Terrender's main goal is to adapt the terrain's details to the current camera state as fast as possible. However, this adaptions may require a lot of new data from one frame to another depending on the speed of the camera movement. Considering the limited bandwidth in a network, downloading the data that is required for the optimum details can possibly take longer than the time available for one frame. Instead of waiting for the new data, Terrender continues to display the terrain using the available data and switches to the new data as soon as it is ready. This requires the data management and rendering to be done in parallel. JavaScript, the programming language of Terrender, does not support multi-threadingwith shared memory access but provides other limited options of parallel processing which we exploit to achieve this.

We use three mechanisms for running another part of the program without blocking the current process, shown as dotted control flows in the flowchart in Fig. 4: Asynchronous code execution, browsers' built-in background tasks (e.g. downloading data), and web workers. Asynchronous code execution still runs in the main thread but does not block the current process. Instead, the execution of the code is delayed until no more code is running. In contrast, the browsers' background tasks are optimized to reliably run in parallel to the main thread. When the task is completed, a callback function is called on the main thread. Additionally, they can transform some image data formats, e.g. color JPGs and PNGs, to a format that can be directly loaded to GPU. We transform other formats, such as

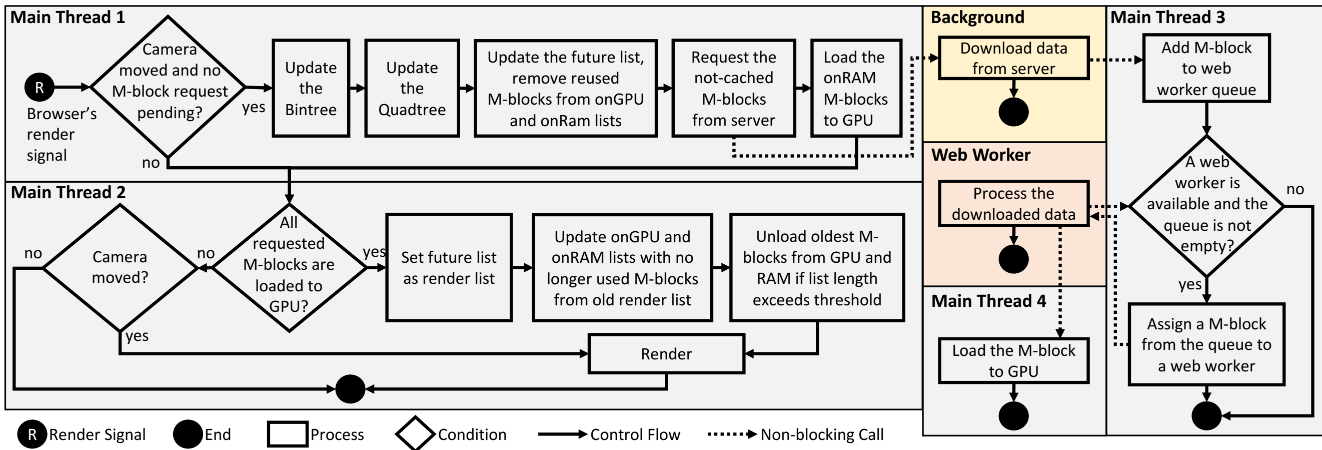

Figure 4: The process of the front end depicted as a flowchart. Note that Main Thread 1 to 4 denote the same thread.

TIFs, in web workers before loading them to the GPU. Web workers offer parallel computation without shared memory access.

The standard way of communication with web workers that run custom code in parallel to the main thread is message passing. To avoid copying large data back and forth, instead of message passing, we use buffers that can be handed over to web workers. After computing the result, the webworker hands the buffer back to the main thread. At any time, either the main thread or the web worker can access the buffer. As shown in the flowchart, when a M-block is required, a background download is started to run in parallel. Meanwhile, the browser can possibly invoke the rendering code several times and the user can navigate the terrain. When the download is done, a function in the main thread is called to assign the downloaded data to a web worker to prepare it as a usable M-block. A fixed number of webworkers defined by the number of cores (if disclosed by the browser) or a constant is used. When a M-block is downloaded and no web worker is available, the downloaded data waits in a queue until a web worker finishes its task. When the web worker is done, it asynchronously calls two functions: one to load the M-block to the GPU which must be done in the main thread and the other to assign waiting downloaded data to a free web worker.

The background downloads run in parallel to the main thread and to each other. When the camera moves to a new location, all the downloads are started at the same time. We keep track of the number of downloading M-blocks. The individual M-blocks that are ready for use can not be used until all requested M-blocks are ready. However, every single M-block that is downloaded and processed by the web worker is loaded to the GPU immediately to avoid loading large amount of data to the GPU at once. When height and color data of all required M-blocks is on the GPU, the newly downloaded M-blocks can be used for rendering.

4.3.2 Data Structures. The frontend of Terrender uses three data structures. The bintree is responsible for adapting the LOD to the current state of the camera by managing the subdivisions of K-patches. The quadtree is responsible for storing the M-blocks. Additionally, the nodes of the quadtree form four doubly linked lists: (1) *render list* that contains the M-blocks that are available on the GPU and are being used for rendering the terrain. (2) *future list* that contains the M-blocks that are needed based on the current state of the camera. (3) *OnGPU list* that contains the M-blocks that are on the GPU but are not in use. (4) *onRAM list* that contains the M-blocks that are available on RAM and can be loaded onto GPU if needed. In Fig. 5, an example situation of the quadtree and its four lists is shown. In this example, the green nodes are being used for rendering the terrain. Based on the camera state, the blue nodes and two of the green nodes are required for the next adaptation of the LOD. Terrender waits until all blue nodes are downloaded, prepared and loaded to the GPU. The orange nodes exist on GPU but are not in use. If the camera moves in a way that any of these nodes are needed, those nodes are available immediately. The red nodes have been previously downloaded. If they are needed again, they can be loaded to GPU and are afterwards available.

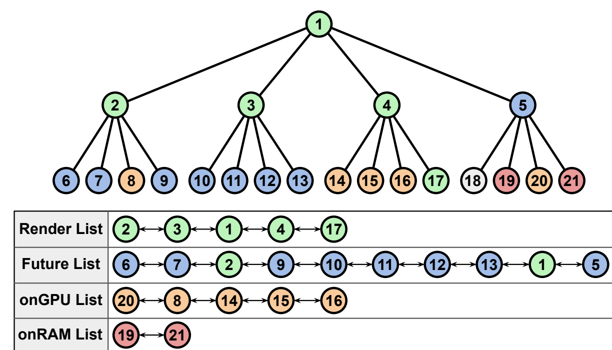

Figure 5: An arbitrary example of the quadtree data structure and the linked lists formed by the quadtree nodes.

4.3.3 Creating the Mesh. Each time the rendering task is invoked by the browser, two actions can occur which are shown in Fig. 4 in Main Thread 1 and Main Thread 2. First, if the camera has moved and there are no M-blocks waiting to be moved from future list to

render list, the bintree is updated. Then, the quadtree is updated based on the new bintree and the required M-blocks are added to the future list. The future M-blocks that are on the onRAM list are loaded to the GPU and the rest of the required M-blocks that are not in the onGPU list nor in the render list are requested to be downloaded in the background in parallel. Second, if all the M-blocks in the future list are loaded to the GPU, the future list becomes the render list and the M-blocks from the render list that are not in the future list are moved to the onGPU list. Finally, the scene is rendered with the new data. If unfinished M-block requests exist and the camera has moved, the new frame is rendered with the unchanged render list using the available data on the GPU.

Every bintree node corresponds to a K-patch, is associated with a M-block, and has a transformation that positions the K-patch inside the M-block. The required M-blocks are identified based on these associations.

During long camera movements, a large amount of new data may be needed. Instead of waiting for all the required data to become ready, we progressively increase the details without compromising any property of Terrender. To achieve this, we decide if a new level of the bintree should be created based on the ratio between the not loaded and loaded quadtree nodes. If the ratio exceeds a certain threshold, the bintree creation is stopped at the current level and the data required starts loading. Once the loading finished the bintree is reevaluated. This progressive increase of details results in higher visual fidelity during long movements.

The nodes in the onGPU and onRAM lists are ordered based on the time they were last used for rendering. Every time a node is added to the onGPU or onRAM lists, we check if the length of theses lists exceeds a certain threshold. In this case, the older nodes are removed from the list. The ones unloaded from the GPU memory go to the onRAM list.

4.3.4 Rendering. While the frontend is triggered every frame by the browser, the terrain is only redrawn if either (a) the camera state has changed, e.g. the camera moved, (b) new data is ready for rendering, (c) the rendering mode changed, e.g. from triangles to wireframe lines, or (d) an external module requests rendering. An external module can be an application built on top of Terrender. If none of these reasons applies, the frontend will not draw anything. This reduces the computational footprint of the application and helps especially on mobile devices to limit the battery usage.

During the effective drawing of the terrain, a draw call is issued for every leaf of the bintree. Each draw call results in a K-patch to be rendered. The vertices of the K-patches do not have the height dimension and can be reused for different nodes. In the vertex shader, first, a 2D transformation provided by the bintree node is applied to the vertices of the K-patch to translate and rotate the K-patch into its correct location within the M-block. Then, the transformed 2D coordinates are used to look up the height value in the height texture of the M-block. The resulting 3D coordinate of the vertex is still relative to the M-block. A model matrix for the current M-block is applied to the relative 3D coordinates to obtain the world coordinate of the vertices. At this point, the vertices of the K-patch can be treated as vertices of a static textured mesh. This enables the developers to apply a broad range of rendering techniques in forward and deferred pipelines.

5 RESULTS

Terrender provides an easy to use terrain renderer with adaptive level of details both for height and color data. The resulting terrain is always a continuous textured surface. The navigation through the terrain is always smooth and the background processes such as downloading new data and adapting the LOD do not affect the user experience adversely. Based on our experiments, it runs on the three major web browsers. Supporting both WebGL 2 and WebGL 1 with a few widely-supported extensions make Terrender compatible with mobile devices as well as desktops.

In addition, Terrender is capable of different rendering modes and displaying statistics which are precious for debugging and exploring the internal behavior of the RASTeR algorithm. In this regard, it can visualize the effect of different metrics by rendering only the K-patch outlines shown in Fig. 6(d). It also can render the wireframe mesh which allows us to better evaluate the actual LOD of the terrain as illustrated in Fig. 6(a, b). Furthermore, it is possible to have M-blocks randomly colored for visualizing the extents of each M-block and understand the connection between K-patches and M-blocks as shown in Fig. 6(c). Additionally, Terrender can generate buffers containing the world coordinates of the terrain per screen pixel useful for further applications with deferred rendering pipeline. The rendering parameters that control the quality of the rendered terrain can be changed at run time allowing to investigate the optimal parameters for a given terrain and usage environment.

We tested Terrender with a dataset of Switzerland. The raw height data saved as GeoTif is 60.6GB in size, and the generated pyramid consisting of GeoTifs compressed using the deflate algorithm is 41.1GB in size. The pyramid using PNG files is 29.2GB. Fig. 6(a,b) show how detailed the height dataset is. The raw color data is 250.1GB in size when saved as uncompressed GeoTif. The generated pyramid is 190.5GB in size using deflated GeoTif. Converting the tiles to JPEG reduces the total size to 32.8GB. In Fig. 6(a), the color can be seen on the mesh wireframe. For all reported benchmarks, the data is preprocessed to 257-blocks and 129-patches are used. A MacBook Pro from 2021 with an M1 Max processor and 64GB of RAM has been used for all benchmarks.

5.1 Terrender's Performance

We first tested Terrender under optimal conditions without a bandwidth limit. This is an ideal stress test for the frontend JavaScript code as the loading of the data is not a limiting factor. In this benchmark, the camera moves on a predefined trajectory for 10 seconds.

We ran this experiment using three different browsers: Chrome, Firefox and Safari. In Fig. 7, the frame rate during the trajectory is visualized. While in all three browsers the application was usable, the differences between the browsers are notable. In Chrome, the application runs at a nearly constant 60FPS. However, in Safari and Firefox, notable frame drops are visible.

In Fig. 8, the terrain updates during the trajectory are marked using a more life-like bandwidth of 40Mbit/s in Chrome browser. The values on the y axis denote the depth of the quadtree that has been created. The terrain is updated fairly often at the beginning because data from the initial render can be reused. Later in the trajectory, the updates are more apart and the depth of the used

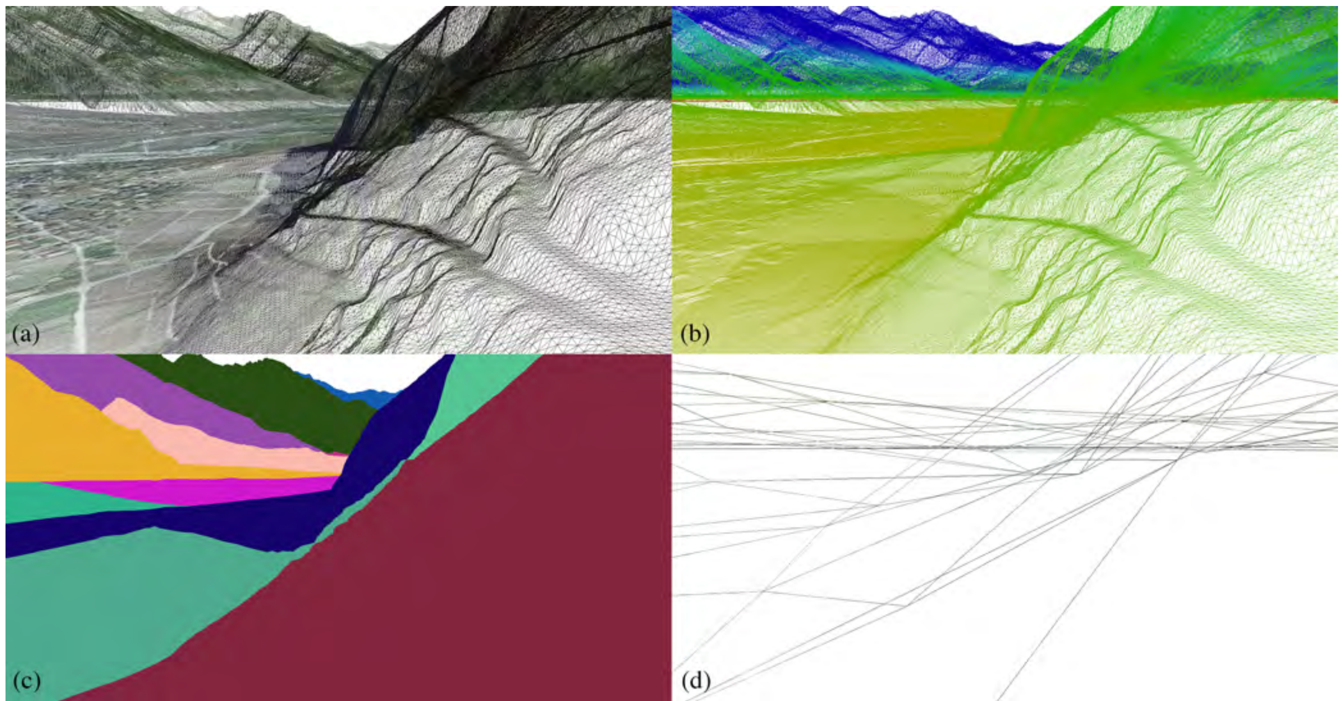

Figure 6: The different view modes of Terrender with the view showing a mountain road. (a) Mesh with normal colors. (b) Mesh with height dependent color gradient. (c) M-blocks rendered in a random color. (d) Outlines of the rendered K-patches

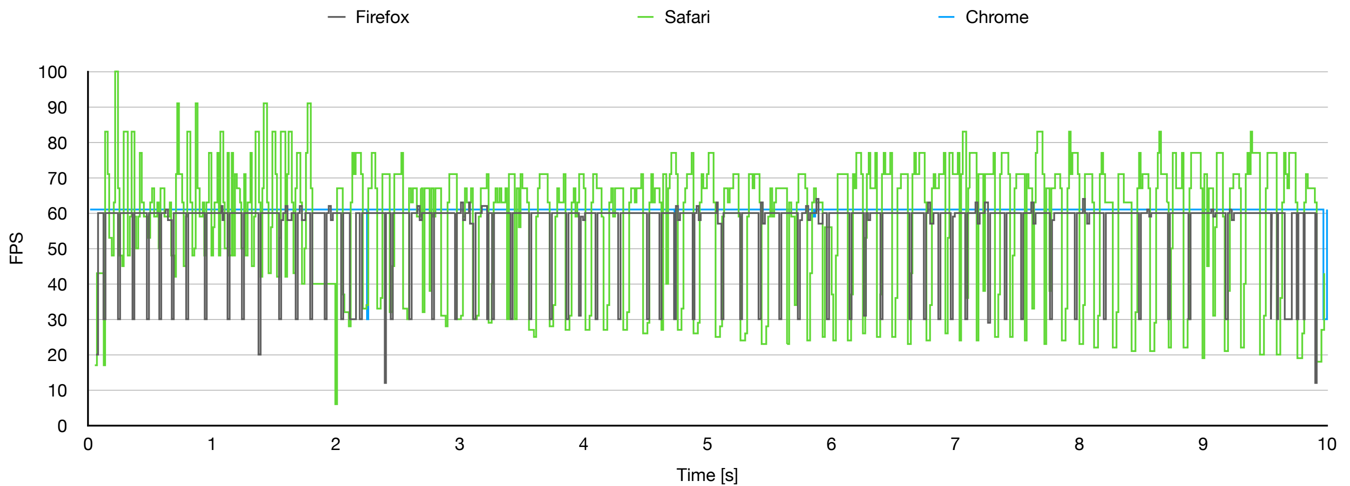

Figure 7: The frames per second of Terrender during the trajectory.

quadtree decreases as more data needs to be loaded. The maximum depth of the quadtree is 9 in this benchmark.

5.2 Example Application

One of the advantages of Terrender is the simplicity of the final rendering data. After retrieving the height data for each pixel in the vertex shader, the rest of the pipeline can be freely designed by the developer. To show this capability, we implemented an example application with a deferred rendering pipeline that allows the user to draw lines on top of the terrain. For finding the click point, we used the world coordinates output written to a G-buffer. For rendering the lines, we implemented the deferred vector rendering technique [Thöny et al. 2018]. In this technique, the lines are provided as vector data stored in a texture. The data structure of the texture allows optimized spatial queries. In the fragment shader, each pixel of the terrain is tested against the lines. If the pixel is covered by a line, the pixel gets the line color. A screenshot of this application drawing lines over the terrain is displayed in Fig. 9.

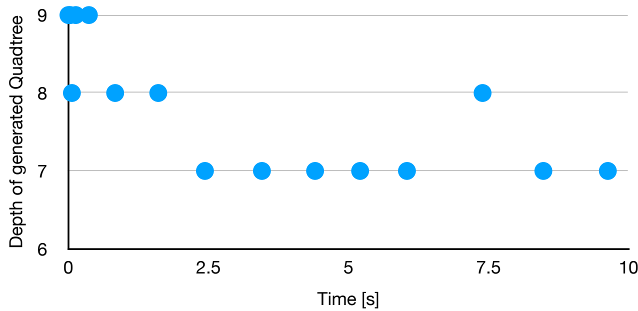

Figure 8: The timestamps at which the terrain was updated in Terrender and the depth of the generated quadtree.

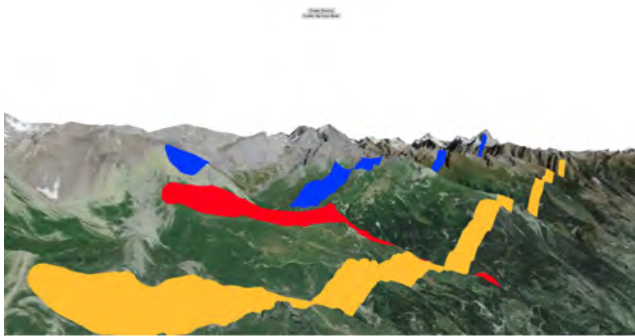

Figure 9: Example Application built on top of Terrender that allows the drawing of lines on the terrain.

5.3 Comparison with CesiumJS

CesiumJS is a well-known open-source framework for creating web-based geographical information systems. It has been developed for a long time and provides a wide variety of features. We focus only on the terrain component of CesiumJS for the comparison.

The main difference between CesiumJS and Terrender is the advantages of using the RASTeR algorithm. Both systems divide the terrain into quadratic tiles but there are two major differences in their usage. Firstly, CesiumJS renders each of them as a whole while Terrender uses the tiles partially based on the current need. Secondly, RASTeR can use tiles from different LODs side by side without causing discontinuities in the terrain's mesh. We compare both systems during a camera movement.

To setup CesiumJS for comparison, we preprocessed our height data using the open source preprocessor built for CesiumJS [Zwaagstra et al. 2018]. An open source server for CesiumJS [Zwaagstra and rumicuna 2016] served the height data. The cameras were programmed to move along the same trajectory in both systems. The bandwidth for both systems was limited to 40Mbit per second. The color data, however, is not the same. For CesiumJS we used the color data provided as default by Cesium GS, Inc which itself is provided by *Bing Maps*. As this color data was not hosted on our local server, we measured the ping of the requests to be in the range of 50-100ms and artificially added a similar delay to our local server when serving color data.

Fig. 10 shows a comparison between the visual quality of the rendered output of CesiumJS and Terrender. In (a) and (b), the camera is not moving. In this situation, both systems create comparable results. Some of the differences like the CesiumJS's fog in the far distance and harder shadows in Terrender are due to the different styles and color textures. In (c) and (d) the screenshot is captured when the camera was moving thus the data was not completely loaded. In this situation, some discontinuities are visible in the output of CesiumJS. The terrain in Terrender is guaranteed to be always continuous. The comparison was conducted using the Chrome browser on which both algorithms perform their best.

Both frameworks are able to maintain a fluid frame rate throughout the trajectory. For the whole trajectory, Terrender downloaded 70MB of color and terrain data and CesiumJS 25MB. In total, e.g. including scripts and geometry error, Terrender downloaded 100MB and CesiumJS 44MB. The lower data usage of CesiumJS is due to using more compressed tiles in comparison to the raster tiles that Terrender uses. On the other hand, the number of single requests is considerably higher for CesiumJS. Terrender sent in total 524 requests while CesiumJS sent 2849. The reason is that CesiumJS tries to load all the tiles up to a certain distance from the same LOD while Terrender can use lower LODs for medium to far regions.

In the second comparison between Terrender and CesiumJS, we wanted to know how much each system is behind their own ideal state when the camera is moving. Therefore, we counted the number of complete terrain updates during the trajectory. By complete terrain update we mean that no more data would be loaded if the camera would not move again. We have disabled the progressive increase of details in this test to make Terrender be comparable to CesiumJS. As CesiumJS continuously renders loaded data, for this comparison, we stop the camera as long as CesiumJS was loading. When the loading completes and CesiumJS is in its ideal state, we moved the camera to the next position where the camera should be at that time stamp. As Terrender only updates the terrain once all data has been loaded no modifications were done.

Fig. 11 depicts the updates that CesiumJS and Terrender performs. Closer updates means that the system is able to catch up with the camera movement better. Both systems are able to generate and update the mesh at a similar frequency. Terrender updates the terrain a bit more frequently especially at the beginning where only close tiles are updated and the required new data is smaller. When camera moves further, the amount of required new data increases and both systems fall behind their ideal states. At this situation, the integrity of the terrain in Terrender is maintained.

Our comparison results show that Terrender performs on the same level as a well-known and well-developed terrain rendering system while offering additional advantages. Our system guarantees high-fidelity mesh for the terrain even when the data it requires for the scene is delayed. This advantage currently comes with a data overhead which can be reduced in a future research, e.g. by optimizing the tile data structure.

6 CONCLUSION

In this paper we introduced Terrender, an open-source, web-based, real-time, variable LOD terrain rendering framework that can handle large-scale terrain height and color texture data. By employing

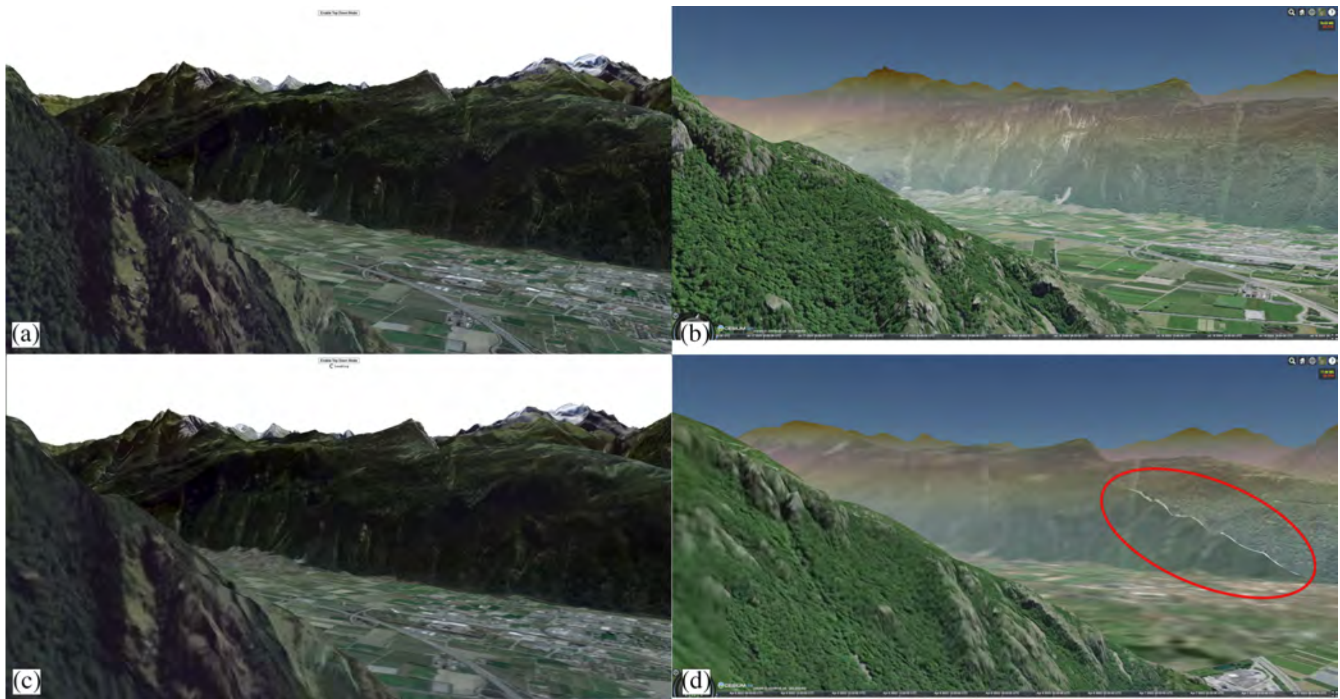

Figure 10: Screenshot of the terrain with for the scene optimal mesh in (a) Terrender and (b) CesiumJS and the same scene during the trajectory in (c) Terrender and (d) Cesium. In (d) the discontinuities in CesiumJS are visible. Color data in CesiumJS courtesy of *Bing Maps*.

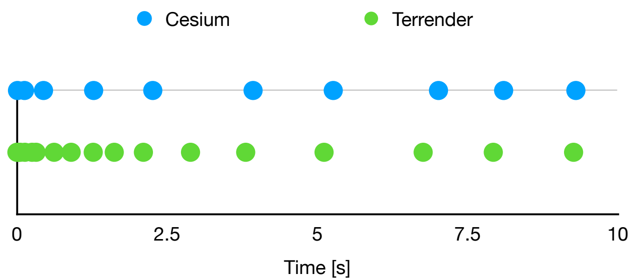

Figure 11: The timestamps at which the terrain was updated for both CesiumJS and Terrender.

RASTeR, a modern multi-resolution terrain rendering algorithm, at its core, Terrender can generate and continuously update a high-fidelity triangulated terrain mesh in real time. It can locally adapt the triangle density to the camera parameters and the geometry error. Our framework is intended for development of web-based rendering systems that need to render a terrain based on real-world or generated height and color data.

Terrender can run in web browsers on a wide range of desktop and mobile devices including laptops, phones and tablets. The behavior of Terrender can flexibly be adjusted by changing the error metric functions and mesh parameters such as the K-patch size. This enables the quality to be adjustable according to the available resources like the network bandwidth. In the future, these adjustments could be automatized. Additionally, Terrender produces simple accessible meshes suitable for a wide range of GPU-based computer graphic techniques.

We compared our system with the terrain rendering from CesiumJS. The results show that our system performs competitively while increasing the fidelity and quality of the mesh, especially when the required data for a scene is not completely available. This can often happen in web applications with a limited bandwidth.

Terrender shows that sophisticated terrain rendering algorithms such as RASTeR are feasible in web applications, despite the limited resources and functionalities. Also, it demonstrates that the limited parallel computing power of JavaScript is sufficient for preparing the data concurrently without affecting the user experience.

Next, we will focus on additional tools to enhance the workflow of developing applications on top of Terrender, while keeping the simplicity and accessibility of the rendering data structure.

ACKNOWLEDGMENTS

The authors want to thank the Swiss Federal Office of Topography Swisstopo for providing the Swiss SwissTLM dataset. This project was partially supported by a Swiss National Science Foundation (SNSF) research grant (project no. 200021_169628).

REFERENCES

Fabio Bettio, Enrico Gobbetti, Fabio Marton, and Giovanni Pintore. 2007. High-quality networked terrain rendering from compressed bitstreams. In *Proceedings ACM Conference on 3D Web Technology*. 37–44.

Jonas Bösch, Prashant Goswami, and Renato Pajarola. 2009. RASTeR: Simple and Efficient Terrain Rendering on the GPU. In *Proceedings Eurographics Areas Papers, Scientific Visulization*. 35–42. https://doi.org/10.2312/ega.20091006

Inc Cesium GS. 2021. Cesium JS. https://cesium.com/platform/cesiumjs/. Accessed: 2021-11-02.
Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. 2003. BDAM - Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization. *Computer Graphics Forum* 22, 3 (2003), 505–514.
Leila De Floriani, Paola Marzano, and Enrico Puppo. 1996. Multiresolution Models for Topographic Surface Description. *The Visual Computer* 12, 7 (August 1996), 317–345.
Christian Dick, Jens Schneider, and Rüdiger Westermann. 2009. Efficient Geometry Compression for GPU-based Decoding in Realtime Terrain Rendering. *Computer Graphics Forum* 28, 1 (March 2009), 67–83.
Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Marc C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. 1997. ROAMing Terrain: Real-time Optimally Adapting Meshes. In *Proceedings IEEE Visualization*. Computer Society Press, 81–88.
Inc. Environmental Systems Research Institute. 2022. ArcGIS. https://developers.arcgis.com/javascript/latest/. Accessed: 2022-04-08.
Lei Feng, Chaoliang Wang, Chuanrong Li, and Ziyang Li. 2011. A Research for 3D WebGIS based on WebGL. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, Vol. 1. 348–351. https://doi.org/10.1109/ICCSNT.2011.6181973
Robert J. Fowler and James J. Little. 1979. Automatic Extraction of Irregular Network Digital Terrain Models. *SIGGRAPH Comput. Graph.* 13, 2 (aug 1979), 199–207. https://doi.org/10.1145/965103.807444
Thomas Gerstner. 2003a. Multiresolution Compression and Visualization of Global Topographic Data. *Geoinformatica* 7, 1 (2003), 7–32.
Thomas Gerstner. 2003b. *Top-Down View-Dependent Terrain Triangulation using the Octagon Metric*. Technical Report. Institute of Applied Mathematics, University of Bonn.
Enrico Gobbetti, Fabio Marton, Paolo Cignoni, Marco Di Benedetto, and Fabio Ganovelli. 2006. C-BDAM – Compressed Batched Dynamic Adaptive Meshes for Terrain Rendering. *Computer Graphics Forum* 25, 3 (September 2006), 333–342. http://www.crs4.it/vic/cgi-bin/bib-page.cgi?id='Gobbetti:2006:CCB'
David Hill. 2002. *An efficient, hardware-accelerated, level-of-detail rendering technique for large terrains*. Master's thesis. University of Toronto, Department of Computer Science.
Hanna Holst. 2004. *Avoiding cracks between terrain segments in a visual terrain database*. Master's thesis. Institutionen för teknik och naturvetenskap.
Roberto Lario, Renato Pajarola, and Francisco Tirado. 2003. HyperBlock-QuadTIN: Hyper-Block Quadtree based Triangulated Irregular Networks. In *Proceedings IASTED International Conference on Visualization, Imaging and Image Processing (VIIP)*. 733–738.
Gregory Larrick, Yun Tian, Uri Rogers, Halim Acosta, and Fangyang Shen. 2020. Interactive Visualization of 3D Terrain Data Stored in the Cloud. In *2020 11th IEEE Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*. 0063–0070. https://doi.org/10.1109/UEMCON51285.2020.9298063
Peter Lindstrom and Valerio Pascucci. 2001. Visualization of Large Terrains Made Easy. In *Proceedings IEEE Visualization*. Computer Society Press, 363–370.
Peter Lindstrom and Valerio Pascucci. 2002. Terrain Simplification Simplified: A General Framework for View-Dependent Out-of-Core Visualization. *IEEE Transactions on Visualization and Computer Graphics* 8, 3 (2002), 239–254.
Ali Mahdavi-Amiri, Troy Alderson, and Faramarz Samavati. 2015. A Survey of Digital Earth. *Computers & Graphics* 53 (December 2015), 95–117. https://doi.org/10.1016/j.cag.2015.08.005
Ruzinoor Che Mat, Abdul Rashid Mohamed Shariff, and Ahmad Rodzi Mahmud. 2009. Online 3D Terrain Visualization: A Comparison of Three Different GIS Software. In *2009 International Conference on Information Management and Engineering*. 483–487. https://doi.org/10.1109/ICIME.2009.57
Mario Ohlberger and Martin Rumpf. 1999. Adaptive Projection Operators in Multiresolution Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics* 5, 1 (January-March 1999), 74–93.
Renato Pajarola. 1998. Large scale Terrain Visualization using the Restricted Quadtree Triangulation. In *Proceedings IEEE Visualization*. 19–26,515. https://doi.org/10.1109/VISUAL.1998.745280
Renato Pajarola, Marc Antonijuan, and Roberto Lario. 2002. QuadTIN: Quadtree based Triangulated Irregular Networks. In *Proceedings IEEE Visualization*. 395–402. https://doi.org/10.1109/VISUAL.2002.1183800
Renato Pajarola and Enrico Gobbetti. 2007. Survey on Semi-Regular Multiresolution Models for Interactive Terrain Rendering. *The Visual Computer* 23, 8 (2007), 583–605. https://doi.org/10.1007/s00371-007-0163-2
Enrique G. Paredes, Margarita Amor, Marcial Bóo, Javier Díaz Bruguera, and Jürgen Döllner. 2016. Hybrid Terrain Rendering based on the External Edge Primitive. *International Journal of Geographical Information Science* 30, 6 (2016), 1095–1116. https://doi.org/10.1080/13658816.2015.1105375
Alex A. Pomeranz. 2000. *ROAM Using Surface Triangle Clusters (RUSTiC)*. Master's thesis. University of California at Davis.
Matthias Thöny, Markus Billeter, and Renato Pajarola. 2018. Large-Scale Pixel-Precise Deferred Vector Maps. *Computer Graphics Forum* 37, 1 (February 2018), 338–349. https://doi.org/10.1111/cgf.13294
Wei Wan, Zhenkun Yang, Xingqiang Du, and Xinwei Zhao. 2021. Space Make the Virtual a Reality: A Web-Based Platform for Visualization and Analysis with Earth Observation Satellite Data. In *2021 IEEE 7th International Conference on Virtual Reality (ICVR)*. 279–285. https://doi.org/10.1109/ICVR51878.2021.9483848
Homme Zwaagstra and rumicuna. 2016. Cesium Terrain Server. https://github.com/geo-data/cesium-terrain-server. Accessed: 2022-04-08.
Homme Zwaagstra, Thomas Weidner, Akira Kurosava, Chris Cooper, Takayuki Mizutani, Jule, and gberaudo. 2018. Cesium Terrain Builder. https://github.com/geo-data/cesium-terrain-builder. Accessed: 2022-04-08.

A RASTER

When displaying a 3D terrain on a screen, the distance from the camera to the terrain is not constant and varies depending on the camera perspective. In order to render the terrain efficiently, the LOD of different locations on the terrain is adapted to their distances from the camera, such that the overall LOD of the mesh projected on screen appears uniform. This is especially important when the height data is large and the entire terrain can not be displayed at the highest LOD. Our terrain renderer follows the RASTeR approach [Bösch et al. 2009] for displaying large-scale terrains with view-dependent LOD without the need to manipulate vertices at run time to preserve the continuity of the terrain.

RASTeR creates the terrain's mesh by starting with a small set of primary isosceles right triangles that cover the whole area, and subdivides each triangle into two new isosceles right triangles when more details are needed at a region of the terrain. The subdivision is done by adding a vertex in the middle of the hypotenuse and connecting it to the opposite vertex. RASTeR avoids hanging nodes or T-junctions by always also subdividing the adjacent triangle sharing the same hypotenuse.

To avoid too many checks and subdivisions at run time, RASTeR uses pre-triangulated triangular patches with a fixed number of vertices per side. These are called *K-patches* where the K stands for the number of vertices on one side as illustrated in Fig. 12. The size K of the K-patch has to satisfy the condition $K = 2^k + 1$ where $k \in \mathbb{N}$.

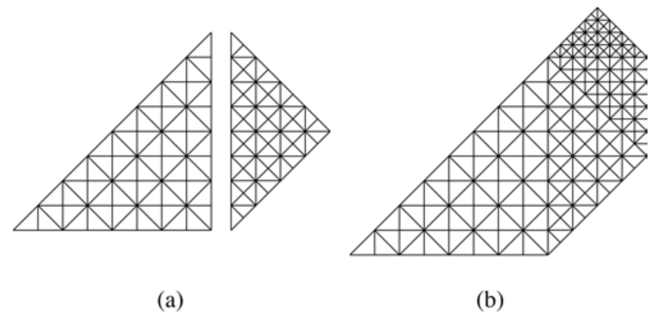

Figure 12: Examples of K-patches. (a) A 9-patch with different orientation. (b) how the 9-patches from different LODs can be attached to each other without creating T-junctions.

The subdivision process of each primary triangle is represented as a binary tree. These bintrees can be created top-down using view and terrain dependent error metrics to evaluate if a bintree node should be subdivided or not. From the generated bintree, which can be unbalanced, the leaf nodes are rendered.

The actual height data is represented by square terrain tiles called *M-blocks* which are organized in a quadtree. The size M of the M-blocks stays the same across all levels while the LOD resolution doubles every level in the quadtree. The M-block tiles must comply to three criteria: (1) the side length of the tiles should be of size $M = m^2 + 1$ where $m \in \mathbb{N}$. (2) Each tile must have one line of pixels in common with all its neighboring tiles to avoid cracks when tiles of the same LOD are side by side. (3) when downsampling the tiles for a lower resolution, exactly every second value from the child tiles is taken to avoid cracks when combining tiles from different LODs. The K-patches are associated with the lowest resolution M-block that provides a unique height value for each vertex. For this to be possible, the size M of the M-block and the size K of the K-patch have to satisfy the following condition:

$$K \leq \frac{(M-1)}{2} + 1 \tag{4}$$

In the end, each K-patch maps into exactly one M-block, however, one M-block can contain multiple different K-patches. The number of K-patches that can map into one M-block is proportional to the quotient between M and K. In Fig. 13, (a) shows a 5-block and (b) shows its four children that together cover the same area. Each child is a 5-block that shares two sides with two other child 5-blocks. The shared sides contain exactly the same values. (c) shows an example set of 14 3-patches. Each triangle is a 3-patch with three points on its sides. Three of the 3-patches use the height values from the red 5-block. Five of them use the yellow 5-block, and six of them use the blue 5-block. There are overlapping points from 3-patches that are bound to different 5-blocks. However, all the 5-blocks have exactly the same value at the overlapping points. For example, in the center, corners from the red, yellow and blue 3-patches overlap. Even though these 3-patches get the heights from three different 5-patches, the values are the same. (d) shows the terrain mesh when the internal lines of 3-patches are drawn.

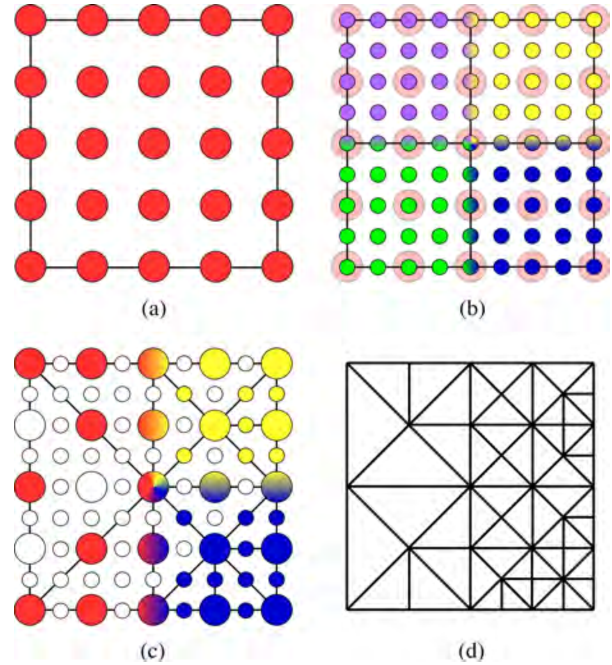

Figure 13: The relationship between K-patches and M-blocks. (a) A 5-block, (b) the same area covered by four 5-blocks from the next higher LOD, (c) an example of 14 3-patches. The color of each point indicates which 5-blocks share the point. Points with gradient colors are shared between multiple 5-blocks. White points are not used by any 3-patch. (d) The corresponding terrain mesh.