Multi-resolution terrain rendering with GPU tessellation

Article *in* The Visual Computer · April 2014

DOI: 10.1007/s00371-014-0941-6

CITATIONS
18

READS
1,696

4 authors, including:

Hanyoung Jang
NCSOFT
24 PUBLICATIONS **135** CITATIONS
SEE PROFILE

Chang-Sik Cho
Electronics and Telecommunications Research Institute
8 PUBLICATIONS **61** CITATIONS
SEE PROFILE

Noname manuscript No.
(will be inserted by the editor)

Multi-resolution Terrain Rendering with GPU Tessellation

HyeongYeop Kang · Hanyoung Jang · Chang-Sik Cho · JungHyun Han

Received: date / Accepted: date

Abstract GPU tessellation is very efficient and is reshaping the terrain rendering paradigm. We present a novel terrain rendering algorithm based on GPU tessellation. The planar domain of the terrain is partitioned into a set of tiles, and a coarse-grained quadtree is constructed for each tile using a screen-space error metric. Then, each node of the quadtree is input to the GPU pipeline together with its own tessellation factors. The nodes are tessellated and the vertices of the tessellated mesh are displaced by filtering the displacement maps. The multi-resolution scheme is designed to optimize the use of GPU tessellation. Further, it accepts not only height maps but also geometry images, which displace more vertices toward the higher-curvature feature parts of the terrain surface such that the surface detail can be well reconstructed with a small number of vertices. The efficiency of the proposed method is proven through experiments on large terrain models. When the screen-space error threshold is set to a pixel, a terrain surface tessellated into 8.5M triangles is rendered at 110 fps on commodity PCs.

Keywords Terrain rendering · GPU tessellation · Height map · Geometry image

H. Kang · J. Han
Computer Science and Engineering, Korea University, Seoul, Korea
E-mail: jhan@korea.ac.kr
H. Jang
NCsoft, Seongnam, Korea
C. Cho
Electronics and Telecommunications Research Institute, Daejeon, Korea

1 Introduction

Interactive terrain rendering is essential for many applications such as 3D games and flight simulators. The most popular representation for terrain is the *height map*, which is a set of height/elevation data sampled in a regular grid. In order to increase the rendering performance, many *multi-resolution techniques* have been proposed, where the CPU program selects only a subset of the height map samples to produce a coarser mesh, which is then transferred to the GPU for rendering.

Because of the ever increasing computing power of GPUs, using a fine-grained multi-resolution algorithm at the CPU side is no longer recommended. Instead, it is considered more beneficial to allocate coarse-grained control of the terrain data to the CPU, and then, allocate the rest of the processing to the GPU [24].

GPU began supporting *hardware tessellation* with OpenGL 4.0 and Direct3D 11. GPU tessellation is very efficient and is reshaping the terrain rendering paradigm. In a typical implementation [3], for example, a quad input to the GPU is tessellated on the fly into a high-resolution planar mesh, and then, its vertices are vertically displaced by referencing the height-map texture. Using a mid-range GPU, a terrain surface composed of 10M triangles can be rendered in real time. However, such a proof-of-concept implementation adopts a brute-force multi-resolution method and leaves plenty of room for optimization.

In the context of GPU-tessellated terrain rendering, we propose a multi-resolution scheme that balances the use of the CPU and GPU well. We have devised a coarse-grained algorithm at the CPU side, which optimizes the use of GPU tessellation. Further, the multi-resolution scheme accepts not only height maps

but also *geometry images* [12], which can effectively encode the terrain surface features and reconstruct the surface more accurately with a smaller number of vertices.

The remainder of this paper is organized as follows. In Section 2, related studies are reviewed. In Section 3, the general flow of terrain rendering based on GPU tessellation is presented, and in Section 4, we describe our multi-resolution scheme. In Section 5, geometry images for terrain rendering are presented. In Section 6, the experimental results are presented and related issues are discussed. In Section 7, we present our conclusions.

2 Related Work

Terrain rendering approaches have been categorized into two groups. The first group exploits a regular hierarchical structure to represent a multi-resolution terrain. A good example is the hierarchies of right triangles (HRT) [10]. Lindstrom et al. [17] presented a view-dependent dynamic meshing algorithm using HRT. Duchaineau et al. [8] proposed a dual-queue algorithm, which splits and merges triangles while maximizing the use of frame-to-frame coherence. Lindstrom and Pascucci [18] presented a simple HRT method with an efficient view-dependent refinement technique and an indexing scheme for organizing the data in a memory-friendly manner. The same authors [19] also showed smoother view-dependent meshes and pointerless indexing scheme that improves the data locality and paging performance. Pajarola [23] and Samet [27] used a restricted quadtree triangulation for terrain visualization. The main idea shared by the algorithms is to build a regular multi-resolution hierarchy through refinement and simplification. The refinement method starts from an isosceles right triangle and recursively bisects its longest edge to create two smaller right triangles. The steps are reversed for simplification. Pairs of right triangles located in a regular triangulation of a gridded terrain are selectively merged.

On the other hand, the second group is based on triangulated irregular networks (TINs). Puppo [25] utilized multi-triangulation approach to build TINs of variable resolutions. El-Sana and Varshney [9] and Xia et al. [33] presented algorithms for performing a fine-grained level-of-detial (LOD) control of general TIN input meshes by using adaptive merge trees. Hypertriangulation [6], view-dependent progressive mesh [13], and its extension [14] were also based on TINs. Because the terrain surface can be sampled non-uniformly, TINs outperform regular hierarchical structures in terms of triangle and error counts. However, TINs require more complicated multi-resolution data structures as well as more storage for the same number of sample points.

To reap the benefits of both HRT and TIN, some approaches [4,5] presented restricted quadtree triangulation in a single data structure. It is based on the idea of exploiting the partitioning induced by a recursive subdivision of the input domain in a right triangle cluster. By doing so, the per-triangle workload to extract a multi-resolution model is significantly reduced and small patches can be preprocessed and optimized off-line to achieve more efficient rendering.

The advances in GPU computing power and programmability have driven the development of new algorithms for multi-resolution terrain rendering. Wagner [32] used GPU-based geomorphs to render terrain patches of different resolutions. Hwa et al. [15] used GPU programmability to generate and render procedural details for terrains. Dachsbacher and Stamminger [7] presented a geometry image warping method, where the procedural geometry and texture detail can be added using GPU. Schneider and Westermann [28] proposed progressive transmission of the discrete hierarchy of terrain data between the CPU and GPU. Losasso and Hoppe [22] presented a clipmap approach, which renders the terrain as a set of nested regular grids centered about the viewer. The grids, stored at the video memory, represent different levels of detail at power-of-two resolutions and enable efficient GPU utilization. Livny et al. [21] cached a persistent grid in the video memory and projected it onto the height field to reconstruct a view-dependent terrain surface. More recently, Livny et al. [20] proposed to subdivide the terrain into rectangular patches. Each patch is represented by four triangular tiles, which are selected from different resolutions, and four strips that stitch the tiles seamlessly. At run time, the GPU generates the meshes of the patches using the scaled instances of cached tiles and elevates each vertex using the height-map textures.

The evolution of GPU architecture enables the polygons to be generated within the GPU pipeline. Tatarchuk [30] presented a terrain rendering approach that utilizes the hardware-tessellation units available on ATI Radeon 2000 series. Although it was implemented on a specific GPU, it motivated the research of adaptive tessellation using GPU. Ripolles et al. [26] introduced an adaptive tessellation scheme for terrain that works completely on the geometry shader, which is unfortunately incapable of generating a sufficient number of triangles needed for terrain rendering.

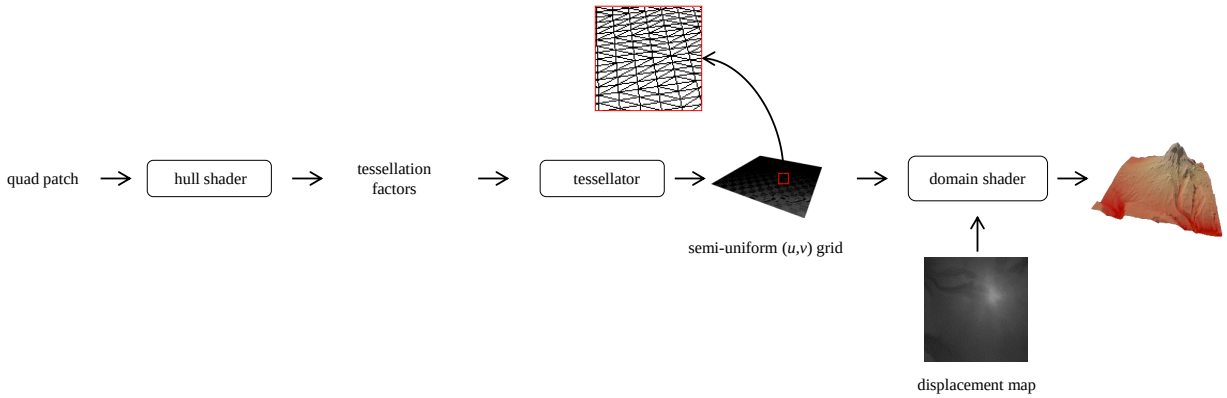


Fig. 1 GPU pipeline for terrain rendering. A quad is input to the GPU and is tessellated on the fly into a high-resolution planar mesh. Then, its vertices are vertically displaced by referencing the displacement map.

With the advent of OpenGL 4.0 and Direct3D 11, GPUs earnestly began supporting hardware tessellation and the vertex transmission problem between the CPU and GPU, which was regarded as a major bottleneck in real-time rendering systems, was solved. Cantlay [3] implemented a technique for terrain rendering using Direct3D 11 tessellation functionalities. However, it does not consider the local adaptivity and consequently produces many unnecessary triangles, thus causing performance degradation. Valdetaro et al. [31] presented a tessellation-based terrain rendering algorithm, where the surface is refined by calculating the second-order derivative of the terrain data and the distance from the terrain to the camera. The proposed algorithm does not produce multi-resolution tiles, leading to limited performance gain. Yusov and Shevtsov [36] also presented a tessellation-based terrain rendering method. At the preprocessing stage, the method constructs a multi-resolution height map representation. Cracks between adjacent blocks of terrain are filled by special meshes at run time. The multi-resolution scheme is based on subdivision into equal-sized blocks and consequently unnecessarily many triangles are often generated.

The algorithms proposed in this paper accept not only height maps but also geometry images [12], where an irregular triangle mesh is converted into a topological disk, and then, the disk is parameterized such that the mesh surface is resampled into a texture. The single-chart geometry image has been extended to multi-chart textures [11]. Moreover, we presents a multi-resolution scheme for GPU-tessellated terrain rendering with no crack.

3 GPU Pipeline for Terrain Rendering

Fig. 1 illustrates a typical flow for terrain rendering based on GPU tessellation. The input to the GPU pipeline is typically represented in a *bivariate parametric patch*; the most suitable patch for terrain rendering is the *quad patch* or *bilinear patch* with four corner points. The programmable stage *hull shader* determines the *tessellation factors*, and then, the hard-wired stage *tessellator* produces a grid of (u, v)s. Another programmable stage *domain shader* evaluates the patch with (u, v)s to generate the mesh vertices, and offsets them using a displacement map, such as a height map.

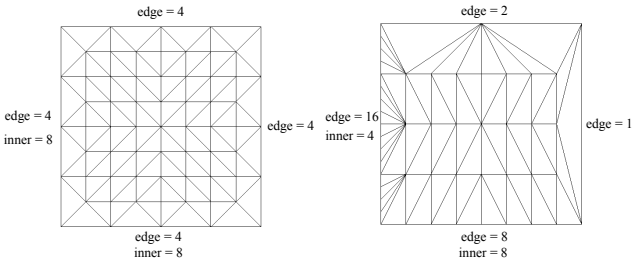


Fig. 2 The inner and edge tessellation factors are denoted by "inner" and "edge," respectively.

In the GPU, a patch is associated with four "edge" tessellation factors and two "inner" tessellation factors, as shown in the examples in Fig. 2. The inner tessellation factors determine the amount of parts into which a patch is broken vertically and horizontally. In contemporary GPUs, the tessellation factors are limited to the range [1,64]. For the sake of simplicity, we call the tessellation factor simply *LOD* (level of detail), and we use the terms, *edge LODs* and *inner LODs*.

4 Multi-resolution Scheme

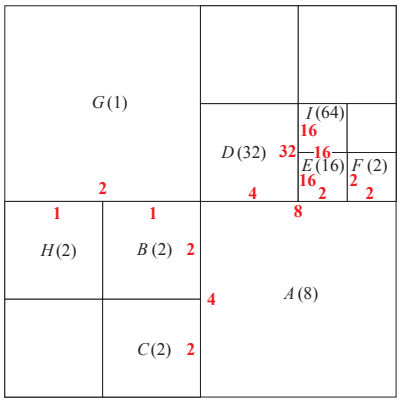


Fig. 3 A quadtree is constructed for each tile. When node A's inner LOD is 8, for example, we denote it by A(8). The edge LODs (in red) are computed using the inner LODs.

The planar domain of a large terrain area is partitioned into a set of *tiles*. In the current implementation, a tile covers an area of 513×513 samples in the height map. (Section 6.5 discusses the texture resolution issue.) The tiles are processed independently, and a *quadtree* is computed for each tile using a *screen-space error metric*. Fig. 3 illustrates a tile decomposed into quadtree nodes. Each node of the quadtree is assigned its own LODs. Then, the quadtree is input to the GPU and each node of the quadtree is taken as a quad patch in the flow depicted in Fig. 1. It should be noted that the LODs are computed by the CPU program, instead of the hull shader, in the proposed scheme.

In our method, the patch LODs are restricted to 2^n, i.e., 1, 2, 4, 8, 16, 32, and 64, in order to prevent cracks occurring between patches. Further, the two inner LODs (vertical and horizontal) in a patch are made identical, because little can be gained by distinguishing between them in terrain rendering.

4.1 Object-space Errors

The screen-space errors are determined using the object-space errors, which are computed at the pre-processing stage. For computing the object-space errors, we construct a discrete hierarchy of meshes. The finest mesh in the hierarchy is made from the original height map of a tile and has 513×513 vertices. The mesh resolution is then repeatedly decreased to a quarter size, i.e., to 257×257, 129×129, 65×65, etc. A mesh composed of $(2^n+1)\times(2^n+1)$ vertices is named "level 2^n." The coarsest mesh at level 1 has 2×2 vertices; there are ten levels in total, as shown in Fig. 4(a).

For each level in the hierarchy, the object-space errors are computed. The tile domain is divided into 8×8 blocks, and for each block, the largest *vertical error* is computed with respect to the finest mesh at level 512.

The meshes shown in Fig. 4(a) are used at the pre-processing stage for computing the object-space errors and then discarded. At run time, we use the 8×8 object-space errors computed for each level.

4.2 Quadtree and Inner LODs

At run time, a quadtree is computed for a tile, and each node in the quadtree is assigned a power-of-two inner LOD up to 64. Then, four edge LODs of a node are determined using the inner LODs of the node and its neighbors. In this subsection, we present the quadtree and inner LODs, and in the next subsection we present the edge LODs.

For a tile, the object-space errors are retrieved from the coarsest level (level 1). Each of 8×8 object-space errors is placed at its block's boundary closest to the viewpoint and is projected into the screen space. The maximum among them is taken as the screen-space error of the tile. If the screen-space error is smaller than the predefined threshold, the inner LOD of the tile is set to the current level, e.g., 1 for the coarsest level. Otherwise, the next levels are iteratively tested in the same manner.

If a tile is assigned inner LOD 64 or below, the entire tile is taken as a single patch and input to the GPU pipeline. Otherwise, LOD 128 should be considered; however, this is beyond the capability of GPU tessellation. To resolve this problem, the tile is split into four quadrants, and each quadrant is tested as to whether its screen-space error with LOD 64 is acceptable. If it is, the smallest possible LOD (64 or below) is computed for the quadrant; otherwise, the quadrant is split again. This recursive split produces a quadtree.

The quadtree depth is limited to four. As a 513×513-resolution height map is associated with a tile, a node at depth four occupies an area of 65×65-resolution, which is equivalent to the maximum tessellation factor 64 of the GPU. At this point, the geometric error becomes zero, and therefore, there is no reason to split the node. The quadtree in Fig. 3 has the maximum depth, four, and the four smallest nodes of the tree will not be split further.

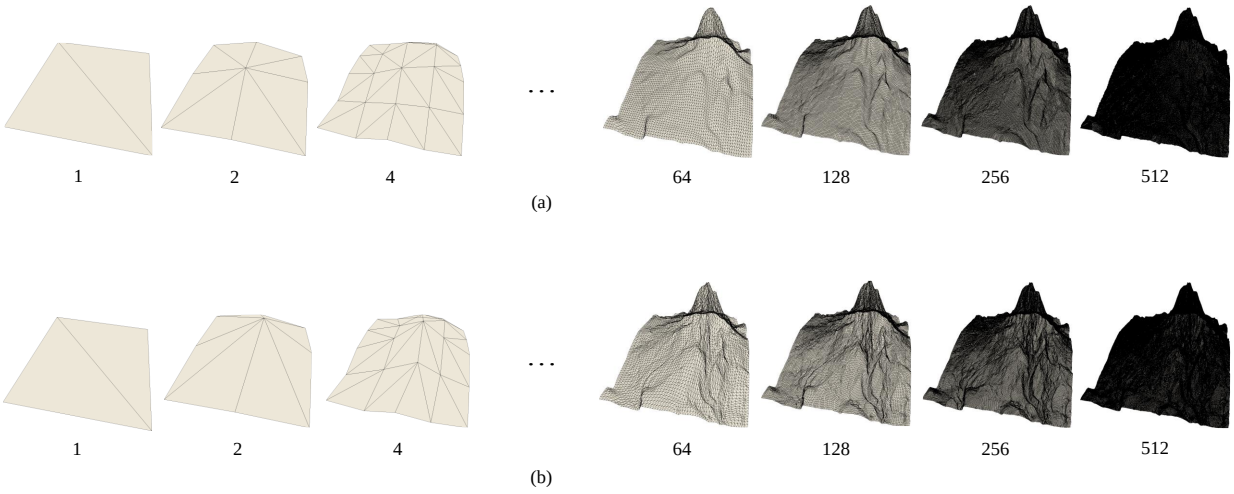


Fig. 4 Mesh hierarchies used for computing the object-space errors. The mesh with $(2^n + 1) \times (2^n + 1)$ vertices is named "level 2^n," $n = [0..9]$. (a) Height-mapped terrain. (b) FGI-mapped terrain. Observe that, in the low-curvature parts, the vertex density is lower than that shown in (a). In contrast, the vertex density is higher in the high-curvature parts. Consequently, the surface details are better described.

4.3 Edge LODs

Once the inner LODs are determined for all leaf nodes of the quadtree, the edge LODs can be computed. Our goal is to compute the *minimum* edge LOD such that the number of triangles in the tessellated mesh is minimized. For this purpose, a breadth-first traversal is performed. Suppose that, in Fig. 3, node A is visited, whose inner LOD is 8. Its left edge is shared by two "smaller" nodes, B and C. We then select the highest LOD from the "smaller" nodes and then scale it up according to the ratio of the edge lengths of the "larger" node, A, and the selected "smaller" node. In the example, B and C have the same inner LOD, 2, and 2 is scaled up twice, i.e., to 4, because B's (and C's) edge is half the size of A's. Let us take 4 as the *vote* from B and C. On the other hand, A's inner LOD is 8 and it is taken as the vote from A. In the range bounded by the votes, i.e., in [4..8], we select the *minimum* that satisfies two conditions: (1) it should be a power of two, and (2) it is distributable to the edges according to the ratio of the edge lengths. In the range [4..8], we select 4. Then, the LOD of A's left edge is set to 4. Simultaneously, 4 is distributed to B and C according to the ratio of their edge lengths; B and C will have the same edge LOD, 2.

It should be noted that, when the inner LOD of B is computed using the method presented in Section 4.2, not only its interior but also its boundary edges are considered. This is also the case in C. The right edges of B and C are shared by A, and therefore, the LOD of A's left edge can be safely set to 4 even though its inner LOD is 8. This implies that A's interior has higher curvature than does its left boundary. In fact, the first example in Fig. 2 shows this case.

Let us see more examples. Consider A's top edge shared by three "smaller" nodes, D, E, and F. The vote from A is 8 whereas that from D, E, and F is 64. In the range [8..64], 8 is selected as the minimum that satisfies the above conditions. Then, the LOD of A's top edge is set to 8. Simultaneously, 8 is distributed to D, E, and F according to the ratio of their edge lengths. Then, the edge LODs of D, E, and F will be 4, 2, and 2, respectively.

Consider G's bottom edge shared by two "smaller" nodes, B and H. The vote from G is 1 whereas that from B and H is 4. In the range [1..4], 1 does not satisfy the second condition; if 1 were distributed to B and H, their edge LODs would be 0.5, which unfortunately cannot be handled by the GPU. In contrast, 2 in [1..4] satisfies the condition. Consequently, the LOD of G's bottom edge is set to 2, and B and H have the same edge LOD, 1.

Let us consider nodes E and I that are "of the same size." The smaller of their inner LODs is 16. It is taken as the LOD of the edge shared by E and I.

The procedure presented in this subsection is applied also to an edge shared by two "tiles." For this, quadtrees and the inner LODs need to be determined

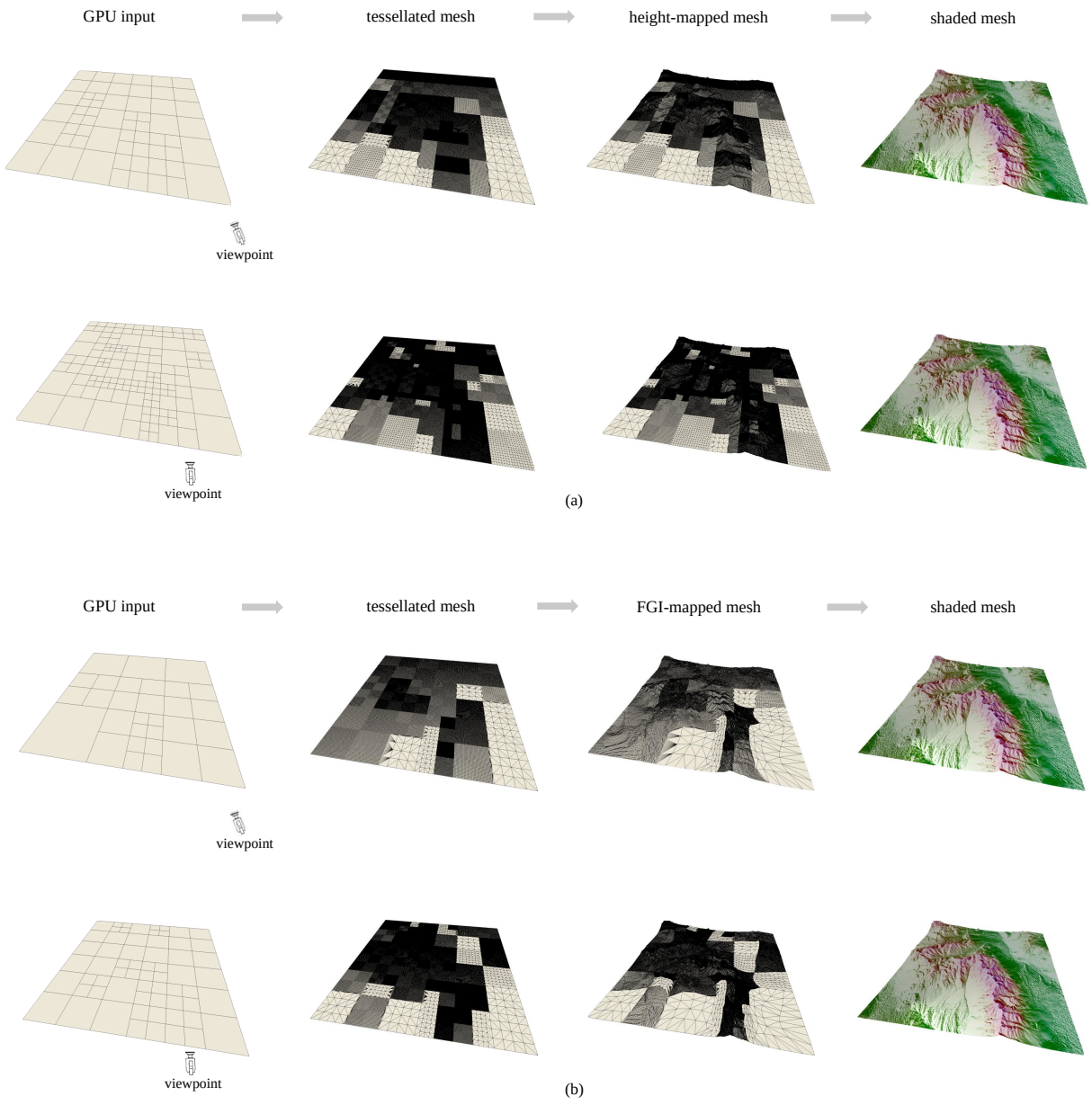


Fig. 5 The GPU flow illustrated using an example of 3×3 tiles. The quadtrees are refined as the viewpoint approaches the terrain surface. The tiles are accordingly tessellated, and the vertices of the tessellated mesh are displaced using the per-tile displacement maps. (a) Height-mapped terrain. (b) FGI-mapped terrain.

for all tiles overlapped with the view frustum, which we call *active* tiles, before the edge LODs are computed.

4.4 Quadtree Update

The quadtree of a newly activated tile is "constructed from scratch," but the quadtree of a tile that was active also in the previous frame is "updated." Temporal coherency is exploited between frames. For

Algorithm 1: Node merging

```
N = bottom_up_search_for_parent(quadtree)
while N do
    if N's children are all leaf nodes then
        if LOD 64 is sufficient for N then
            delete N's children
        end
    end
    N = bottom_up_search_for_parent(quadtree)
end
```

Algorithm 2: Node splitting

```
N = depth_first_search(quadtree)
while N do
    if N is a leaf node then
        if LOD 64 is insufficient for N then
            split N into four children
        end
    end
    N = depth_first_search(quadtree)
end
```

this purpose, Algorithm 1 is invoked for merging the quadtree nodes in a bottom-up and breadth-first fashion. For every "parent" of four leaf nodes, the screen-space error is tested using LOD 64. If the error is acceptable, i.e., if it is less than the threshold, the parent need not be split, and therefore, its children (four leaf nodes) are all deleted.

After the node merging procedure, Algorithm 2 is invoked for splitting the quadtree nodes. In a depth-first fashion, the quadtree is searched for a leaf node that makes an unacceptable screen-space error when rendered with LOD 64. Such a node is split into four children, and the depth-first search resumes with the first child.

When the quadtree structure is updated by Algorithms 1 and 2, the only information available about each leaf node is that it is error-free with LOD 64. There is a chance that a coarser LOD is also error-free for the leaf. If a leaf is spawned by Algorithms 1 or 2, the coarser LODs are tested one by one, starting from 32, and the coarsest error-free LOD is assigned to the leaf. On the other hand, if a leaf existed in the previous frame, the coarsest error-free LOD is searched up and down from its current LOD. Fig. 5(a) shows an example of 3×3 active tiles. As the viewpoint moves, the quadtree and its LODs are dynamically updated for each tile.

Let us briefly discuss the noteworthy features of our multi-resolution scheme.

- It is coarse-grained. The maximum depth of our quadtree is four, and the CPU procedure for quadtree handling is executed at a small cost, as will be presented in Section 6.5.
- Although the quadtree nodes (quad patches) are processed independently by the GPU pipeline, the reconstructed mesh for the entire terrain surface is geometrically continuous, i.e., crack-free, because the edge shared between two patches has the same LOD.
- Because the quadtree structure is regular, it is sufficient to input "an index to the quadtree node" to the GPU pipeline instead of "four corner points of the quad patch."

5 Geometry Images for Terrain Rendering

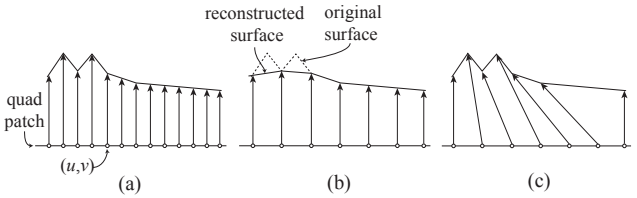


Fig. 6 Height map and feature-preserving geometry image (FGI). (a) Height map displaces the vertices of the tessellated mesh along the surface normals. (b) If the tessellation factors are decreased, the high-curvature feature parts may not be reconstructed accurately. (c) FGI displaces more vertices toward the higher-curvature feature parts, which are then reconstructed with sufficient details.

In the GPU pipeline, the tessellation factors are determined *per patch*, and (u, v)s are *semi-uniformly* distributed within the patch, as illustrated in Fig. 1. Consequently, flat parts of the terrain surface reconstructed from a patch often have unnecessarily many vertices, as shown in Fig. 6(a). On the other hand, if the tessellation factors are decreased, high-curvature feature parts may not have enough vertices, leading to loss of detail (see Fig. 6(b)).

Our multi-resolution scheme presented in Section 4 accepts not only height maps but also *geometry images* [12]. In the context of GPU tessellation, Jang and Han [16] proposed *feature-preserving geometry image* (FGI), which densely samples the high-curvature feature parts of a surface and consequently displaces more vertices of the tessellated mesh toward the feature parts. Fig. 6(c) illustrates the concept of FGI. Fig. 6 implies that, if FGIs are used, the tessellation factors can be kept small, leading to decreased triangle counts and increased frame rates. In Sections 6.2 and 6.3, we use real terrain data and compare the performances (triangle counts and frame rates) of height mapping and FGI mapping.

An additional benefit obtained by replacing the height maps with geometry images is that *overhangs* can then be reconstructed, which is not possible when height maps are used (see Fig. 7). Section 5.1 shows that no additional effort is needed to reconstruct such overhangs.

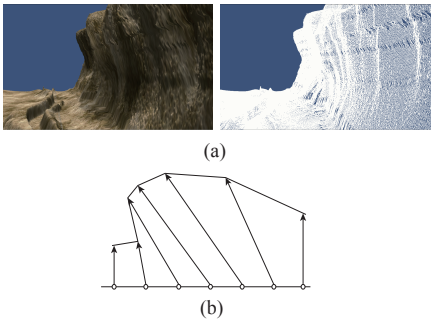


Fig. 7 Overhanging geometry. (a) An overhang example. (b) Overhang reconstructed by FGI.

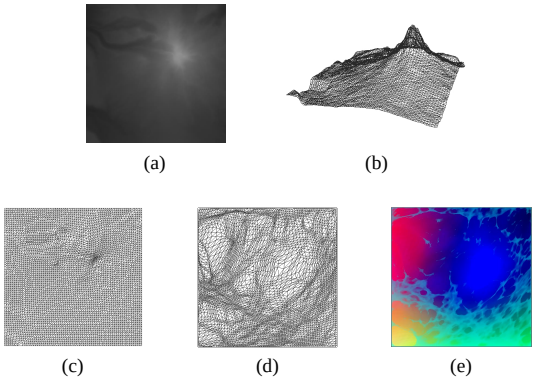


Fig. 8 FGI generation process. (a) Original height map. (b) Regular grid mesh. (c) Initial parameterization. (d) Iteration result. (e) FGI.

5.1 Feature-sensitive Parameterization

An FGI is generated from a height map at the preprocessing stage. We slightly modified the FGI generation method proposed by Jang and Han [16]. Fig. 8(a) shows a height map visualized as a gray-scale image. It is converted into a regular grid mesh, as shown in Fig. 8(b). For each vertex v of the mesh, the *feature weight* (w_v) is estimated in terms of the scalar mean curvature (C_v) [29], the area sum of the triangles sharing v (A_v), and the scale factor (S_v):

$$\begin{aligned} w_v &= \alpha \frac{C_v}{\sum C} + (1-\alpha)\frac{S_v A_v}{\sum SA} \\ C_v &= \textstyle\sum_{e \ni v} 2\|e\| \sin\frac{\theta_e}{2} \\ S_v &= \begin{cases} 2 \; if \quad v \in ridges \\ 1 \; otherwise \end{cases} \\ A_v &= \textstyle\sum_{t \ni v} A_t \end{aligned} \tag{1}$$

where α is a user-defined parameter and θ_e is the exterior dihedral angle of edge e connected to v. *Ridges* are generally considered to be global features that provide silhouettes of the terrain surface, and S_v is used for increasing the weights of the vertices belonging to the ridges. We use MapCalc [2] for ridge detection. $\sum C$ denotes the sum of C_vs of all vertices in the terrain mesh. Similarly, $\sum SA$ denotes the sum of $S_v A_v$s of all vertices. $\sum C$ and $\sum SA$ are used for normalization purpose.

The mesh in Fig. 8(b) is flattened into the 2D space through a traditional parameterization method. The result is shown in Fig. 8(c). Then, for each vertex v, the area of its Voronoi region is made equal to w_v so that if w_v is heavier, more samples are captured in the vicinity of v. For this purpose, the per-vertex parameterization error is defined as

$$\varepsilon_v = 1 - \frac{a_v}{w_v} \tag{2}$$

where a_v denotes the area of v's Voronoi region in the parameter space. The iterative parameterization method [35] gradually updates the parameterized mesh in order to minimize the overall error. The iteration converges quickly and ensures that no flipped triangle is generated. Fig. 8(d) shows the iteration result. Observe that the high-curvature feature parts of the terrain surface take large areas in the parameter space. Finally, the terrain surface is sampled and the sample points are stored in a texture. This is the FGI. As the FGI stores 3D points, it is visualized as a color image in Fig. 8(e).

5.2 Terrain Rendering with FGIs

Let us recall that a tile is associated with a 513×513-resolution height map. The FGI generated from the height map has the same resolution. Then, the flow presented in Section 4 is applied to the FGI. First of all, ten levels of FGI-mapped surfaces are created "through GPU tessellation" by repeatedly decreasing the mesh resolution to a quarter size, i.e., to 257×257, 129×129, 65×65, etc. Fig. 4(b) shows the result. It should be noted that the lower-curvature parts have fewer vertices than the height-mapped surfaces of Fig. 4(a), whereas the higher-curvature parts have more vertices, and consequently, the surface details are better described.

For each level of the mesh hierarchy in Fig. 4(b), 8×8 object-space errors are computed with respect to the mesh at level 512. Then, a quadtree is computed using the screen-space error metric, and the quadtree is input to the GPU pipeline. Fig. 5(b) shows the GPU flow. It can be seen that the quadtrees in Fig. 5(b) are largely coarser than those in Fig. 5(a). Nonetheless, the FGI-mapped terrain maintains the same surface quality as the height-mapped terrain when they use the same screen-space threshold.

5.3 Zippering

At the top of Fig. 9(a), a terrain surface composed of two tiles is shown. Their height maps are *independently* processed to generate their own FGIs. It should be understood that the parameterized meshes may not have an identical boundary, as illustrated in the closeup view of Fig. 9(a).

In Fig. 9(b), the black dot represents a texel located at the boundary shared by two FGIs. Suppose that the texel is referenced when a mesh is reconstructed from the left FGI. It produces a vertex on the left mesh. The texel produces another vertex on the right mesh as well. Unfortunately, the positions of the two vertices may not be identical. As a result, *cracks* may occur at the boundary between the reconstructed meshes, as illustrated in the lower-left image of Fig. 9(b).

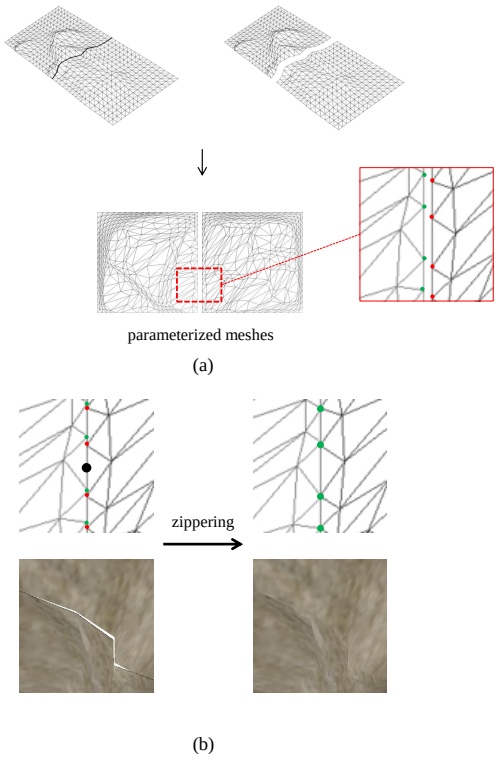


Fig. 9 Zippering. (a) The parameterized meshes often have different boundaries. (b) The zippering process makes the boundaries identical.

To avoid such cracks, we make a two-step effort for *zippering* [12] when the FGIs are generated. When an edge of a mesh is parameterized into the boundary of the parameter space, it is assigned the same length as the parameterized "mate edge" belonging to the adjacent mesh. The boundary edge is sampled into the boundary texels of the FGI, and then, the texels are copied into the boundary of the adjacent FGI. The lower-right image of Fig. 9(b) shows the disappearance of the crack due to zippering.

6 Experimental Results and Discussions

The algorithms presented in this paper are implemented in Direct3D 11 on a 3.4GHz Intel Core i7-3770 CPU with an NVIDIA GeForce GTX 560Ti GPU. Sixteen-bit textures are used for height maps, and the FGIs are of the format DXGI_FORMAT_R32G32B32A32_FLOAT. The screen resolution is 1920×1080, and by default the threshold for the screen-space error is set to a pixel.

The input to our terrain renderer is an array of tiles, where a 513×513-resolution displacement map (height map or FGI) is associated with each tile. Fig. 10 shows the terrain data used in our experiments: Puget Sound, Grand Canyon, and Canada. (See the attached video.)

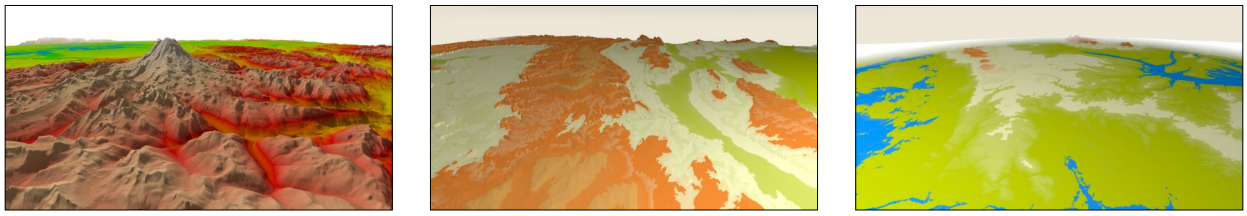
6.1 Out-of-Core Rendering and View Frustum Culling

Our terrain data cover large areas. For efficient out-of-core rendering [34], view frustum culling and resource management are tightly coupled. The resource manager loads the textures and the object-space error tables for newly activated tiles, and simultaneously releases the data for inactivated tiles. When the viewpoint moves rapidly, the stage of loading the texture data often becomes a major bottleneck. In order to avoid such a bottleneck, a pre-fetching technique is adopted with an "enlarged" view frustum. The resource manager also calculates the time elapsed for loading the extra data caused by the "enlarged" view frustum, and immediately stops loading the extra data if the elapsed time exceeds a threshold.

On the other hand, the vertex shader implements its own view frustum culling within the GPU. For each quadtree node, its 3D bounding box is tested for intersection with the view frustum. Only the visible nodes are further processed.

6.2 Height Mapping vs. FGI Mapping

Let us use the meshes in Fig. 4 and compare the object-space errors of height mapping and FGI mapping. The 513×513-resolution height map associated with a tile produces the level-512 mesh in Fig. 4(a). We call it the *original surface*. For each of the other height-mapped



Puget Sound
16K×16K resolution (32×32 tiles)

Grand Canyon
16K×16K resolution (32×32 tiles)

Canada
32K×32K resolution (64×64 tiles)

Fig. 10 Terrain data used for experiments. A tile is associated with a 513×513-resolution displacement map.

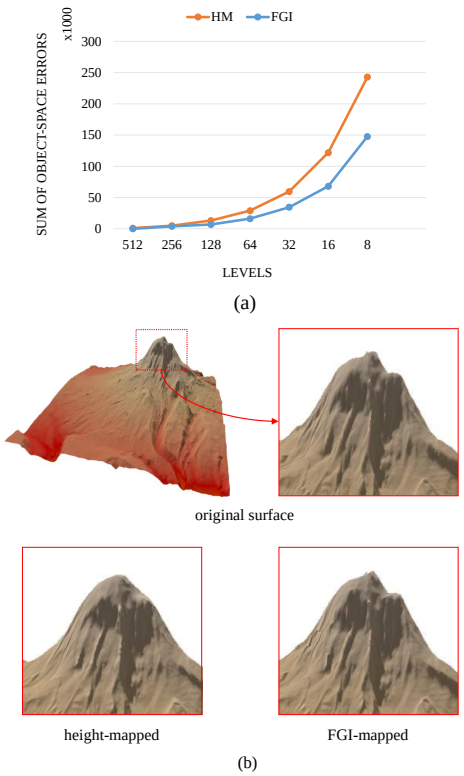


Fig. 11 Comparison of height mapping (denoted by HM) and FGI mapping (denoted by FGI) with a single tile. (a) The object-space errors. (b) Reconstructed meshes.

and FGI-mapped meshes in Fig. 4, the object-space error is computed with respect to the original surface; each mesh surface is sampled at 513×513 grid points and the vertical errors with respect to the original surface are summed. The errors are plotted in Fig. 11(a). The FGI-mapped mesh at level 512 virtually has no error. For both height mapping and FGI mapping, the lower-level mesh incurs the larger error.

Fig. 11(a) clearly shows that height mapping makes larger error than FGI mapping. The original surface is shown at the top of Fig. 11(b); the height-mapped and FGI-mapped meshes shown at the bottom are of level 128. It is not surprising that the high-frequency surface features are reconstructed better by the FGI. It is straightforward to infer from Fig. 11 that, given the same screen-space error threshold, the FGI-mapped mesh has fewer triangles than the height-mapped, and consequently, FGI mapping excels height mapping in terms of the frame rates, as will be shown in the next subsection.

However, there exists a tradeoff between frame rate and memory cost. Let us recall that the FGI stores 3D points, whereas the height map stores 1D scalars. Thus, FGIs require more memory than height maps. Also note that FGIs cannot easily be compressed. It should also be noted that FGI mapping needs additional preprocessing; the FGI itself is generated from the height map, as presented in Section 5.1, and the FGIs are zippered, as presented in Section 5.3.

6.3 Performance Comparison

In the field of GPU-tessellated terrain rendering, a state-of-the-art study was reported on NVIDIA.com [3]. The study was based on height mapping and adopted a brute-force screen-space error metric that determines the tessellation factors using the distance between a patch and viewpoint. On the other hand, Livny et al. [21] proposed a state-of-the-art terrain rendering algorithm, which is *not* based on GPU tessellation. We call the algorithm simply *seamless patches*. We implemented the NVIDIA and seamless patches algorithms in order to compare their rendering performances with those of our algorithms.

Fig. 12(a) and (b) compare the triangle counts and frame rates, respectively, measured by running the

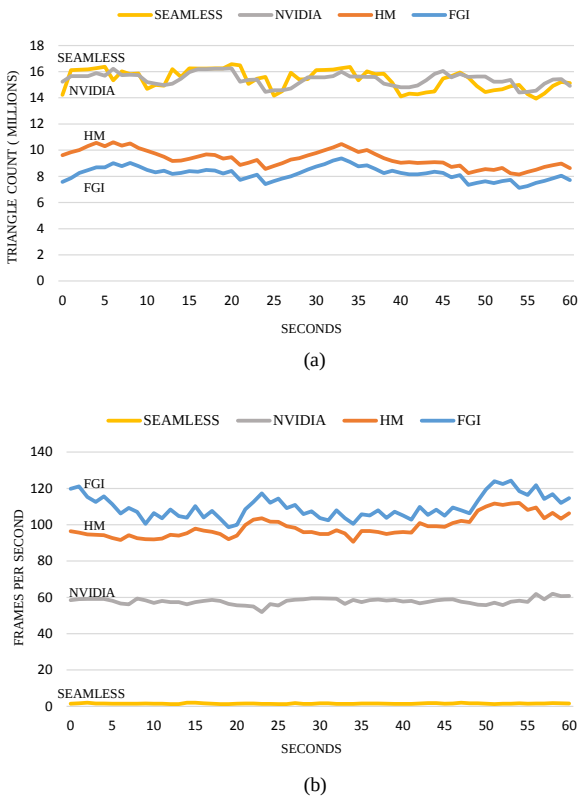


Fig. 12 Performance comparison. "SEAMLESS" and "NVIDIA" denote the seamless patches and NVIDIA algorithms, respectively, whereas our methods based on height maps and FGIs are denoted by "HM" and "FGI," respectively. (a) Triangle counts. (b) Frame rates.

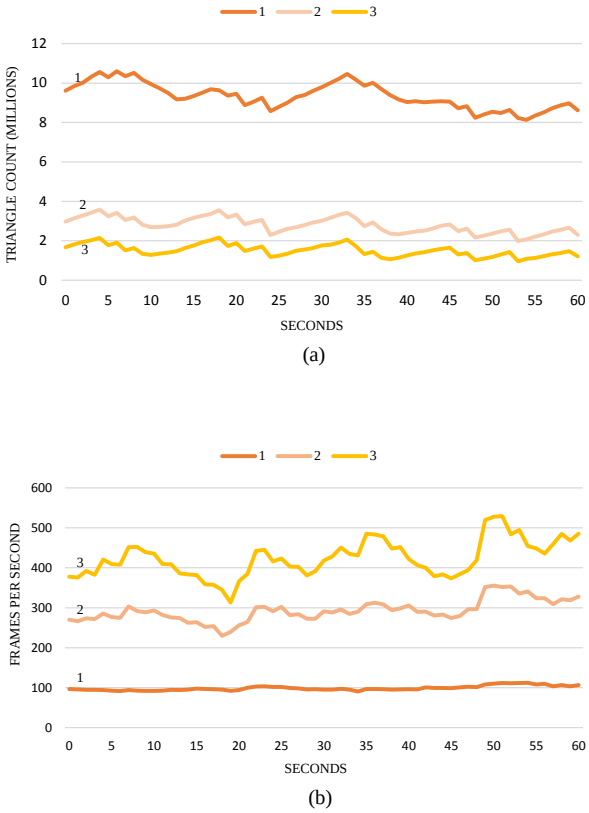


Fig. 13 Performances tested with different screen-space error thresholds. (a) Triangle counts. (b) Frame rates.

seamless patches algorithm, the NVIDIA algorithm, and our algorithms with the Puget Sound data for a 60-second fly through. For all algorithms, the screen-space error threshold is set to one pixel, and a wide and deep view frustum is used, which encompasses millions of triangles. The average frame rates are 1.5, 58, 98, and 110 for the seamless patches algorithm, the NVIDIA algorithm, our height mapping algorithm, and our FGI mapping algorithm, respectively.

6.4 Screen-space Error Threshold

While increasing the threshold for the screen-space error from 1 to 2 and 3, we ran our height mapping algorithm and measured the triangle counts and frame rates. Fig. 13 shows the results. The average frame rates are 98, 292, and 423 for the thresholds 1, 2, and 3, respectively. (For the FGI mapping algorithm, the average frame rates are 110, 312, and 463 for the thresholds 1, 2, and 3, respectively.) Fig. 14 shows the screenshots. Visual artifacts such as popping are not perceived when thresholds 1 and 2 are applied, but are slightly visible at threshold 3.

6.5 Texture Resolution

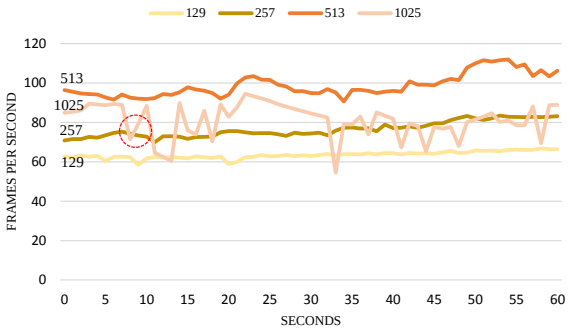


Fig. 15 Performance comparison with varying tile sizes in our height mapping algorithm.

The rendering performance depends on the height-map area covered by a tile. We tested our height mapping algorithm using four tile sizes: 129×129,

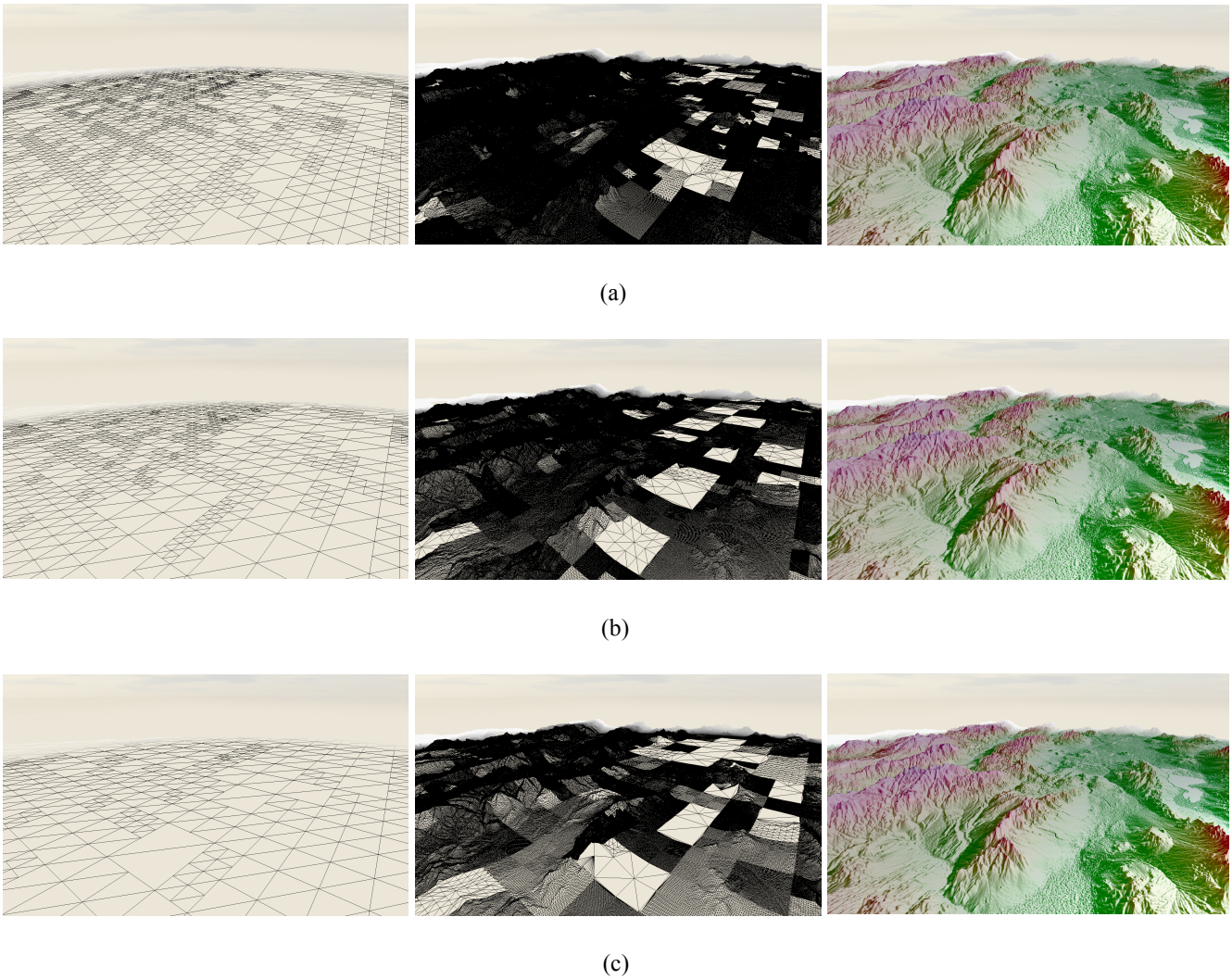


Fig. 14 Screenshots with increasing screen-space error threshold. Left to the right: GPU input (quadtree nodes), height-mapped mesh, and shaded mesh. (a) Threshold = 1. (b) Threshold = 2. (c) Threshold = 3.

257×257, 513×513, and 1025×1025. Fig. 15 compares the frame rates. This subsection discusses why the resolution of 513×513 shows the best performance.

In Section 4.2, we presented why the quadtree depth is limited to four when a 513×513 resolution is used. For the same reason, the maximum depths for 129×129, 257×257, and 1025×1025 resolutions are two, three, and five, respectively. The larger the tile size, the deeper the quadtree. On the other hand, as the tile size becomes larger, the number of active tiles is reduced, and therefore, fewer tiles are processed. Table 1 shows the average numbers of processed tiles for the 60-second fly through.

We measured the execution time consumed only by the CPU procedure for handling the quadtrees. Fig. 16(a) shows the result. It takes 6.1ms, 5.5ms, 3.5ms, and 3.0ms for 129×129, 257×257, 513×513, and 1025×1025 resolutions, respectively. It is found that the number of tiles has more impact on the performance than does the quadtree depth; the larger the tile size, the better the CPU performance.

In the current implementation, the maximum number of textures (including the height map, image texture, and normal map) handled by a single *rendercall* is 128 because of the *texture slot* constraint of the GPU. Consequently, as the tile size is reduced, more rendercalls are needed for rendering the large terrain area (see the last row of Table 1). It is well-known that the rendercall incurs the overhead of context switching and is a major bottleneck in real-time rendering. Fig. 16(b) shows the GPU execution time triggered by the rendercalls. As expected, the more rendercalls, the more degraded performance. An exception is found between 513×513 and 1025×1025 resolutions. It can

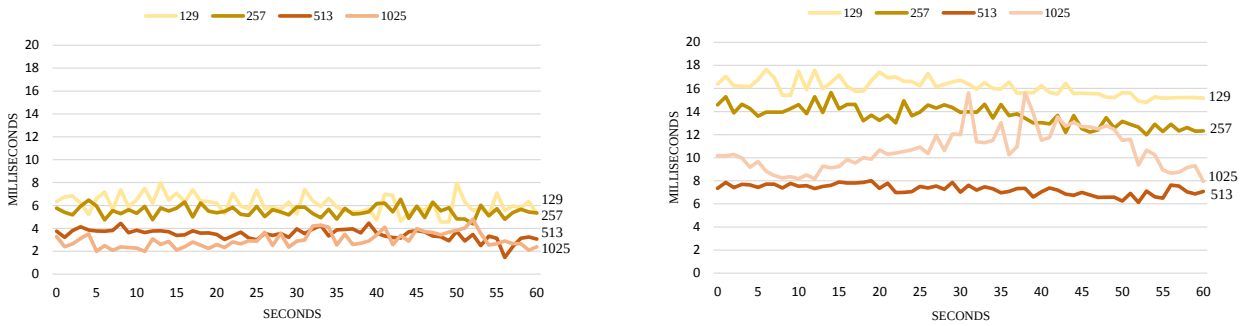


Fig. 16 Performance comparison with varying tile sizes. (a) Time consumed only by the CPU procedure for quadtree handling. (b) GPU execution time.

Table 1 Average numbers of tiles and rendercalls for varying tile sizes

	129×129	257×257	513×513	1025×1025
number of tiles	1345	361	102	31
number of rendercalls	33	9	3	1

be explained in many aspects. First of all, transferring the huge textures of 1025×1025 resolution to the GPU memory significantly decreases the performance due to the narrow bandwidth between the CPU and GPU. Even when a small part of a tile is visible, the entire textures of the tile should be transferred. What is worse, the texture cache performance is also decreased when the textures are too large [1]. When the 513×513 resolution is chosen, the gain shown in Fig. 16(b) surpasses the loss shown in Fig. 16(a). This explains why the 513×513 resolution shows the best performance in Fig. 15.

In Fig. 15, an abrupt drop in the frame rate is marked by a small circle. This kind of drop is encountered when the textures are loaded from the hard disk to the CPU memory. Obviously, the larger is the tile size, the greater is the frame rate drop.

7 Conclusion

This paper presented a multi-resolution scheme for GPU-tessellated terrain rendering. It constructs a coarse-grained quadtree that optimizes the use of GPU tessellation. A strength of the proposed scheme is that not only height maps but also geometry images can be plugged into the scheme. Considering the tradeoff between frame rate and memory/preprocessing cost, users can choose between height maps and geometry images.

The experimental results prove that the proposed method is efficient and robust. However, it also has some drawbacks. For example, the proposed scheme does not support "zoom in Earth from space." It entails a considerably large number of active tiles, which is not afforded by the current implementation because a texture hierarchy is not constructed. On the other hand, the processing time for the FGIs needs to be reduced so that the use of geometry images becomes more attractive for terrain rendering. Further investigation will be done to remedy these drawbacks.

References

1. Gpu programming guide geforce 8 and 9 series. `http://developer.download.nvidia.com/GPU_Programming_Guide/GPU_Programming_Guide_G80.pdf` (2008)
2. Berry, J.: Unlock dems to identify upland ridges. GEOWorld **22**(5), 14–15 (2009)
3. Cantlay, I.: Directx 11 terrain tessellation. `https://developer.nvidia.com/sites/default/files/akamai/gamedev/files/sdk/11/TerrainTessellation_WhitePaper.pdf` (2011)
4. Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., Scopigno, R.: Bdam batched dynamic adaptive meshes for high performance terrain visualization. Computer Graphics Forum **22**(3), 505–514 (2003). DOI 10.1111/1467-8659.00698. URL `http://dx.doi.org/10.1111/1467-8659.00698`
5. Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., Scopigno, R.: Planet-sized batched dynamic adaptive meshes (p-bdam). In: Proceedings of the 14th IEEE Visualization 2003 (VIS'03), pp. 147–154. IEEE Computer Society, Washington, DC, USA (2003). DOI 10.1109/VISUAL.2003.1250366. URL `http://dx.doi.org/10.1109/VISUAL.2003.1250366`

6. Cignoni, P., Puppo, E., Scopigno, R.: Representation and visualization of terrain surfaces at variable resolution. The Visual Computer **13**(5), 199–217 (1997). DOI 10.1007/s003710050099. URL http://dx.doi.org/10.1007/s003710050099
7. Dachsbacher, C., Stamminger, M.: Rendering procedural terrain by geometry image warping. In: Proceedings of the Fifteenth Eurographics conference on Rendering Techniques, pp. 103–110. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2004). DOI 10.2312/EGWR/EGSR04/103-110. URL http://dx.doi.org/10.2312/EGWR/EGSR04/103-110
8. Duchaineau, M., Wolinsky, M., Sigeti, D.E., Miller, M.C., Aldrich, C., Mineev-Weinstein, M.B.: Roaming terrain: real-time optimally adapting meshes. In: Proceedings of the 8th conference on Visualization '97, pp. 81–88. IEEE Computer Society Press, Los Alamitos, CA, USA (1997). URL http://dl.acm.org/citation.cfm?id=266989.267028
9. El-Sana, J., Varshney, A.: Generalized view-dependent simplification. Computer Graphics Forum **18**(3), 83–94 (1999). DOI 10.1111/1467-8659.00330. URL http://dx.doi.org/10.1111/1467-8659.00330
10. Evans, W., Kirkpatrick, D., Townsend, G.: Right-triangulated irregular networks. Algorithmica **30**(2), 264–286 (2001). DOI 10.1007/s00453-001-0006-x. URL http://dx.doi.org/10.1007/s00453-001-0006-x
11. Feng, W.W., Kim, B.U., Yu, Y., Peng, L., Hart, J.: Feature-preserving triangular geometry images for level-of-detail representation of static and skinned meshes. ACM Trans. Graph. **29**(2), 11:1–11:13 (2010). DOI 10.1145/1731047.1731049. URL http://doi.acm.org/10.1145/1731047.1731049
12. Gu, X., Gortler, S.J., Hoppe, H.: Geometry images. ACM Trans. Graph. **21**(3), 355–361 (2002). DOI 10.1145/566654.566589. URL http://doi.acm.org/10.1145/566654.566589
13. Hoppe, H.: View-dependent refinement of progressive meshes. In: Proceedings of the 24th annual conference on Computer graphics and interactive techniques, pp. 189–198. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1997). DOI 10.1145/258734.258843. URL http://dx.doi.org/10.1145/258734.258843
14. Hoppe, H.: Smooth view-dependent level-of-detail control and its application to terrain rendering. In: Proceedings of the conference on Visualization '98, pp. 35–42. IEEE Computer Society Press, Los Alamitos, CA, USA (1998). URL http://dl.acm.org/citation.cfm?id=288216.288221
15. Hwa, L.M., Duchaineau, M.A., Joy, K.I.: Adaptive 4-8 texture hierarchies. In: Proceedings of the conference on Visualization '04, pp. 219–226. IEEE Computer Society, Washington, DC, USA (2004). DOI 10.1109/VISUAL.2004.4. URL http://dx.doi.org/10.1109/VISUAL.2004.4
16. Jang, H., Han, J.: Feature-preserving displacement mapping with graphics processing unit (gpu) tessellation. Comp. Graph. Forum **31**(6), 1880–1894 (2012). DOI 10.1111/j.1467-8659.2012.03068.x. URL http://dx.doi.org/10.1111/j.1467-8659.2012.03068.x
17. Lindstrom, P., Koller, D., Ribarsky, W., Hodges, L.F., Faust, N., Turner, G.A.: Real-time, continuous level of detail rendering of height fields. In: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, pp. 109–118. ACM, New York, NY, USA (1996). DOI 10.1145/237170.237217. URL http://doi.acm.org/10.1145/237170.237217
18. Lindstrom, P., Pascucci, V.: Visualization of large terrains made easy. In: Proceedings of the conference on Visualization '01, pp. 363–371. IEEE Computer Society, Washington, DC, USA (2001). URL http://dl.acm.org/citation.cfm?id=601671.601729
19. Lindstrom, P., Pascucci, V.: Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. IEEE Transactions on Visualization and Computer Graphics **8**(3), 239–254 (2002). DOI 10.1109/TVCG.2002.1021577. URL http://dx.doi.org/10.1109/TVCG.2002.1021577
20. Livny, Y., Kogan, Z., El-Sana, J.: Seamless patches for gpu-based terrain rendering. Vis. Comput. **25**(3), 197–208 (2009). DOI 10.1007/s00371-008-0214-3. URL http://dx.doi.org/10.1007/s00371-008-0214-3
21. Livny, Y., Sokolovsky, N., Grinshpoun, T., El-Sana, J.: A gpu persistent grid mapping for terrain rendering. Vis. Comput. **24**(2), 139–153 (2008). DOI 10.1007/s00371-007-0180-1. URL http://dx.doi.org/10.1007/s00371-007-0180-1
22. Losasso, F., Hoppe, H.: Geometry clipmaps: terrain rendering using nested regular grids. ACM Trans. Graph. **23**(3), 769–776 (2004). DOI 10.1145/1015706.1015799. URL http://doi.acm.org/10.1145/1015706.1015799
23. Pajarola, R.: Large scale terrain visualization using the restricted quadtree triangulation. In: Proceedings of the conference on Visualization '98, pp. 19–26. IEEE Computer Society Press, Los Alamitos, CA, USA (1998). URL http://dl.acm.org/citation.cfm?id=288216.288219
24. Pajarola, R., Gobbetti, E.: Survey of semi-regular multiresolution models for interactive terrain rendering. Vis. Comput. **23**(8), 583–605 (2007). DOI 10.1007/s00371-007-0163-2. URL http://dx.doi.org/10.1007/s00371-007-0163-2
25. Puppo, E.: Variable resolution terrain surfaces. In: Proceedings of the 8th Canadian Conference on Computational Geometry, pp. 202–210. Carleton University Press (1996). URL http://dl.acm.org/citation.cfm?id=648249.751895
26. Ripolles, O., Ramos, F., Puig-Centelles, A., Chover, M.: Real-time tessellation of terrain on graphics hardware. Comput. Geosci. **41**, 147–155 (2012). DOI 10.1016/j.cageo.2011.08.025. URL http://dx.doi.org/10.1016/j.cageo.2011.08.025
27. Samet, H.: Applications of spatial data structures: Computer graphics, image processing, and GIS. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1990)
28. Schneider, J., Westermann, R.: Gpu-friendly high-quality terrain rendering. WSCG **14**, 49–56 (2006)
29. Sullivan, J.M.: Curvature measures for discrete surfaces. In: ACM SIGGRAPH 2005 Courses. ACM, New York, NY, USA (2005). DOI 10.1145/1198555.1198662. URL http://doi.acm.org/10.1145/1198555.1198662
30. Tatarchuk, N.: Dynamic terrain rendering on gpus using real-time tessellation. In: W. Engel (ed.) ShaderX 7: Advanced Rendering Techniques. Charles River Media (2009)
31. Valdetaro, A., Nunes, G., Raposo, A., Feijo, B., Toledo, R.d.: Lod terrain rendering by local parallel processing on gpu. In: Proceedings of the 2010 Brazilian Symposium on Games and Digital Entertainment, pp. 182–188. IEEE Computer Society, Washington, DC, USA (2010). DOI 10.1109/SBGAMES.2010.30. URL http://dx.doi.org/10.1109/SBGAMES.2010.30

32. Wagner, D.: Terrain geomorphing in the vertex shader. In: ShaderX2: Shader Programming Tips and Tricks with DirectX 9.0, pp. 18–32. Wordware Publishing, Inc. (2003)
33. Xia, J., El-Sana, J., Varshney, A.: Adaptive real-time level-of-detail based rendering for polygonal models. Visualization and Computer Graphics, IEEE Transactions on **3**(2), 171–183 (1997). DOI 10.1109/2945.597799
34. Yoon, S.E., Gobbetti, E., Kasik, D., Manocha, D.: Real-Time Massive Model Rendering. Morgan and Claypool Publishers (2008)
35. Yoshizawa, S., Belyaev, A., Seidel, H.P.: A fast and simple stretch-minimizing mesh parameterization. In: Proceedings of the Shape Modeling International 2004, pp. 200–208. IEEE Computer Society, Washington, DC, USA (2004). DOI 10.1109/SMI.2004.2. URL `http://dx.doi.org/10.1109/SMI.2004.2`
36. Yusov, E., Shevtsov, M.: High-performance terrain rendering using hardware tessellation. WSCG **19**(3), 85–92 (2011)