Pacific Graphics 2011
Jan Kautz, Tong-Yee Lee, and Ming C. Lin
(Guest Editors)

Volume 30 (2011), Number 7

River Networks for Instant Procedural Planets

E. Derzapf[1], B. Ganster[2], M. Guthe[1] and R. Klein[2]

[1]Philipps-Universität Marburg, Germany
[2]Universität Bonn, Germany

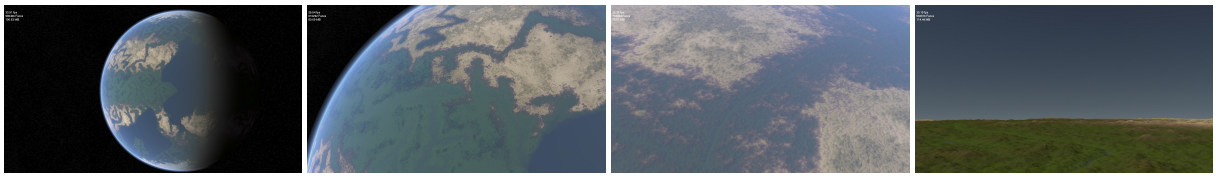

Figure 1: *As the user zooms in, terrain geometry is created to adaptively refine the planet.*

Abstract

Realistic terrain models are required in many applications, especially in computer games. Commonly, procedural models are applied to generate the corresponding models and let users experience a wide variety of new environments. Existing algorithms generate landscapes immediately with view-dependent resolution and without preprocessing. Unfortunately, landscapes generated by such algorithms lack river networks and therefore appear unnatural. Algorithms that integrate realistic river networks are computationally expensive and cannot be used to generate a locally adaptive high resolution landscape during a fly-through. In this paper, we propose a novel algorithm to generate realistic river networks. Our procedural algorithm creates complete planets and landscapes with realistic river networks within seconds. It starts with a coarse base geometry of a planet without further preprocessing and user intervention. By exploiting current graphics hardware, the proposed algorithm is able to generate adaptively refined landscape geometry during fly-throughs.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Fractals, I.3.5 [Computer Graphics]: Physically based modeling, I.3.5 [Computer Graphics]: Object hierarchies

1. Introduction

Movies, simulations, and computer games allow to explore a wide variety of realistic, fictional terrains. In some cases, using real terrain would break the illusion of exploring unknown planets. The computer games Spore and Civilization employ procedural models to generate planets from a set of rules automatically. Procedural models have the advantage that no geometry needs to be stored or streamed, instead they are created when needed. While procedural models may use numerous parameters, default values and help texts allow the user to tweak the planets as desired with minimal effort. In recent years, these algorithms were improved to interactively adapt the geometry to a moving camera, in order to support view-dependent level of detail.

While terrain can be generated quickly using previous procedural models, these terrains lack realistic rivers. Rivers are vital for life and can be important for navigation, as rivers often lead to communities or to the sea. Erosion simulations model the natural processes that form rivers. Unfortunately, these algorithms are computationally expensive and can therefore not be used to generate a locally adaptive, high resolution landscape during a fly-through. Instead of attempting to recreate the physical processes of erosion, we aim for river networks that obey the following observations:

- While endorheic basins exist, most continental areas transport precipitation to the sea over the river networks.
- River networks are surrounded by valleys between mountains and hills.
- Rivers do not cross, they are mostly above ground, and they follow the steepest decline until they reach the coast.

We call such river networks realistic. Our new procedural algorithm creates complete planets and landscapes with realistic river networks without performing an erosion simulation, rendering it suitable for on-the-fly generation of terrain. The algorithm starts by creating a coarse representation of the terrain. Additional geometry has to be produced in order to locally adapt the geometry to the camera position and perspective as the user traverses the terrain. This must be done in a manner that is consistent with the constraints named above. The algorithm uses massively parallel graphics hardware to reach sufficient performance.

In summary, the main contribution of this paper is a novel algorithm that combines the following properties:

- **A**daptive level of detail: We store terrain metadata in the edges and vertices of a mesh and describe rules to refine terrain based on that information. As a result, the algorithm generates river networks at adaptive level of detail.
- **R**ealistic river networks without an erosion simulation: Water levels are computed directly and rivers carve into the landscape without an iterative simulation of water movement. Plausibility of the river networks is maintained while adapting the level of detail.
- **F**ast terrain synthesis: The data structures support massively parallel operations, allowing us to generate a base mesh in less than a second and to process refinements in real time.
- **R**eproducibility: The results of the algorithm are reproducible. This is necessary to ensure that the same terrain is generated when the player returns. It would also guarantee that all players see the same terrain if the algorithm would be integrated into a networked game.

The remainder of this paper is structured as follows: Section 2 analyzes why previous systems cannot generate terrain with river systems spontaneously. Section 3 gives an overview of the operations, reproducibility and computing water levels. Section 4 describes generating the base mesh in detail. In Section 5, we outline the adaption algorithm for real-time rendering. Finally, we evaluate our approach in Section 6, which leads to the conclusion (Section 7).

2. Related Work

Geomorphology studies the processes that shape the relief of Earth. Among these are crust movements, vulcanism and erosion. Erosion can be caused by temperature changes (thermal erosion), water (fluvial erosion), glaciers (glacial erosion), wind (eolian erosion), and other effects.

2.1. Fractal and Procedural Approaches

Mandelbrot [Man83] has analyzed the fractal nature of many types of objects we are dealing with in this paper, including mountains, river networks, and coast lines. Fournier et al. [FFC82] demonstrated how to produce terrain or entire planets using midpoint displacement. Their algorithm starts with a polygon or sphere and recursively inserts new vertices which split polygons into several new polygons. A new vertex's altitude is the average altitude of the surrounding vertices, plus or minus a random value depending on the length of the edges. Midpoint displacement can be used to increase the resolution of terrain. For terrain created using midpoint displacement, it is possible to select an altitude that represents sea level to obtain mountain ranges near the sea, but river networks have to be generated differently. Bokeloh and Wand [BW06] propose a GPU-based implementation of the midpoint displacement algorithm.

Kelley et al. [KMN88] proposed an algorithm for generating procedural terrain with river networks. Their algorithm uses the observation that water flows along edges that are lower than the surrounding landscape. They first create a river network, calculate river vertex altitudes, and finally assign higher altitudes to the surrounding mountains. All tributaries in the terrain lead into a single, main river.

Hnaidi et al. have proposed a method that defines terrain from user-defined control curves that include ridge lines, river beds, hills, cracks and cliffs [HGA*10]. Two-dimensional piecewise Bezier cubic splines are used to define the control curves. The terrain is computed from a set of partial differential equations which are solved on the GPU.

Several algorithms that generate new terrain by transplanting detail from existing terrain have been proposed [BSS06, EF01, ZSTR07]. A number of approaches generate terrain that satisfies user-given constraints [ST89, PGTG04, SS05, Bel07]. These algorithms could also be used for procedural modeling by defining the constraints procedurally. Prusinkiewicz and Hammel integrated midpoint displacement and the creation of a single river into a text rewriting system [PH93]. While a lot of research has focused on modeling streets using grammars, most recent approaches model streets as splines or graphs without grammars [GPMG10, LSWW11].

2.2. Erosion Simulation

Musgrave et al. proposed to simulate fluvial and thermal erosion on fractal terrain [MKM89]. During every step of erosion simulation, rain is dropped onto the surface and gathered in lakes until it leaks at the lake's lowest border and forms a river bed. As the simulation proceeds, many initial lakes are converted to river beds in this manner and a river network is created. Numerous extensions to erosion simulation have been proposed. Beneš and Forsbach proposed a data structure that stores several layers of material in a grid. For each layer, material information and thickness are stored [BF01a]. They also showed that if the terrain is split into strips, erosion simulation can run in parallel for each strip, only the boundary areas need to be treated separately [BF01b] and they demonstrated how to integrate evap-

Algorithm/Authors	Input	Output	Precomputation/Size	Water Effects	Level of Detail
Midpoint Displacement	Parameters	Mesh	Very fast	Global sea level	Adaptive
Kelley et al.	Parameters	Mesh	Very fast	River networks	Fixed
Erosion Simulation	Grid	Layered grid [BF01a]	Minutes	Many	Fixed
Constrained Modeling	Constraints	Grid	2.97s / 1024×1024 [Bel07]	Global Sea Level	Limited
Transplant Terrain	Two grids	Grid	2.5 s / 2794×394 [BSS06]	Global Sea Level	Fixed
Hnaidi et al.	Control curves	Grid	0.3 s / 1024×1024 [HGA*10]	User-defined rivers	Limited
Proposed algorithm	Parameters	Mesh	Very Fast (Table 4)	Realistic river networks	Adaptive (Figure 6)

Table 1: *Comparison of features in previous algorithms.*

oration into erosion simulations [BF02]. Rather than calculating erosion based on amounts of exchanged water, erosion can be simulated with particles [CMF97, KBKv09]. An optimized GPU implementation of erosion simulation has been proposed by Šťava et al. [SBBK08].

2.3. Terrain Rendering and Parallel Level-of-Detail

View-dependent simplification has been an active field of research over the last two decades. Hoppe [Hop96] introduced progressive meshes that smoothly interpolate between different levels of detail. Depending on the view position and distance, a sequence of split- or collapse operations is performed for the vertices to generate a view-dependent simplification. The inter-dependency of split operations can either be encoded explicitly [XV96] or implicitly [Hop97]. Hoppe later optimized the data structures and improved the performance of the refinement algorithm [Hop98]. Duchaineau et al. store triangles in a binary tree, where each level stores the geometry for a single level of detail [DWS*97]. Pajarola and DeCoro [Paj01, PD04] developed an optimized sequential view-dependent refinement algorithm. Losasso introduced the geometry clipmap, which stores geometry for quadratic regions centered around the user, similar to mipmapping [Los04]. Hu et al. [HSH09] proposed a parallel adaption algorithm for progressive meshes. They introduced a relatively compact explicit dependency structure that allows to group vertex splits and half-edge collapses into parallel steps. The drawbacks of this technique are that the explicit dependencies need additional memory and that only half-edge collapses are supported. A more compact data structure for progressive meshes was proposed by Derzapf et al. [DMG10]. It is based on Hoppe's original view-dependent refinement algorithm [Hop98] and supports a massively parallel adaption algorithm.

2.4. Analysis

Table 1 summarizes the features of the previous algorithms. Ideally, such an algorithm would be interactive, support realistic water and erosion effects, and immediately generate terrain at any desired level of detail while the user explores the terrain. For a number of algorithms, the level of detail is fixed. In other cases, the level of detail is limited by the functions that are used to define the terrain. At high polygon counts, additional polygons make features rounder but do not add further detail. In these cases, Table 1 reports the level of detail as limited. Midpoint displacement can be used to add further detail to a terrain [Bel07]. Unfortunately, the algorithm lacks the necessary rules to prevent introducing mountain peaks into rivers. Some algorithms do not support river networks. However, water effects are vitally important as river networks are a defining element of natural landscapes. As a result, in related work, either the level of detail is limited or river networks are missing.

Both midpoint displacement and the method of Kelley et al. come very close to satisfying the stated requirements. The one lacks river networks, while the other lacks adaptive level of detail. Both methods also operate on a mesh. This paper demonstrates how they can be combined into a new algorithm. A parallel implementation is required to reach sufficient performance for interactive applications.

3. Overview

Our approach consists of two phases. In the first phase, the algorithm creates a rough representation of the planet, the so-called base mesh: It creates a sphere, assigns continents, river networks, and altitudes (Section 4). The second phase is adaption (Section 5), where the algorithm interactively and adaptively refines the terrain while the user moves about freely. The adaption phase consists of a number of steps that were optimized to take advantage of massively parallel graphics hardware.

We use a mesh data structure because the algorithm of Kelley et al. requires labeled edges. While it would be possible to store the planet in a displacement map wrapped around a sphere, only eight directions are possible for transporting water between neighboring cells, and a solution would be needed that can exceed these 8 directions when zooming in. Otherwise, parallel rivers would emerge. The mesh data structure supports two atomic operations – edge split and vertex collapse – that are used to manage the level of detail. Edge split operations can be applied to increase the level of detail locally when the user comes closer to parts of the terrain. The reverse operation, vertex collapse, restores the representation at the lower level of detail. Figure 2 shows a split operation that creates faces f_3 and f_4, edges e_2, e_3, e_4 and vertex v_{nv}, while a vertex collapse operation reverses this operation by removing these entities. The designations in Figure 2 are used in the entire paper.

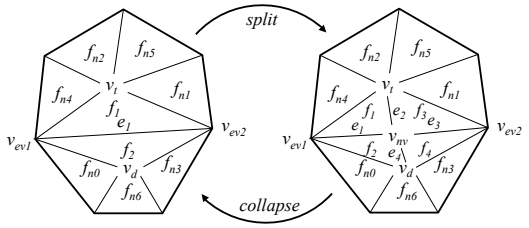

Figure 2: *Vertex collapse and edge split operation.*

3.1. Reproducibility

Procedural models can often be recreated from a random seed, but a naïve implementation of midpoint displacement would still produce different results when zooming into the terrain along a different path. This is because the adaptive refinements would apply the random values in different local order. However, when the terrain is reproducible, only its current representation needs to be stored, as geometry at different levels of detail can be created when needed. Otherwise, exploring the terrain would create new data until storage is exhausted. Therefore, our method asserts that the same terrain is generated regardless of the path of exploration, which allows several players to explore a planet without streaming geometry over networks. A single random seed is used to compute the entire planet. The base mesh is produced from the random seed using a deterministic algorithm and is thus reproducible, including the random seeds for all vertices. As the order of refinements may vary, reproducibility of the edge splits is guaranteed using the random seeds stored in the vertices. We use the sum of the seeds of vertices v_{ev1} and v_{ev2} as the seed of v_{nv}. However, the pseudo-random numbers alone are not enough to ensure that refinements are reproducible. We also need to preserve the local order of the operations, so we assign a split level to every edge. For each split, we increment the edge's split level and decrement it for each collapse. Similar to [ESV99], only the edge with the lowest split level can be split in each face.

3.2. Types of Edges and Faces

A flag in each face stores whether the polygon belongs to the sea or to a continent. Edges are marked as sea, coast, or continent depending on their surrounding polygons. In addition, edges between continental polygons may be flagged as rivers. If a river edge is split, two river edges and two continent edges are created. When a coast edge is split, there will be two coast edges, one sea edge, and one continent or river edge. If a sea edge is split, there will be four new sea edges. When a continent edge is split, the new vertex v_{nv} will have four new continent edges. If the new continent faces are not connected to the river network, the edge between the new vertex v_{nv} and an existing river vertex is converted to a river, to assert that polygons created later are still connected to the river network. While an edge only has one type, vertices have all the types of their incident edges.

3.3. Water Levels

We store ground and water altitudes in each vertex to define water levels for the sea and rivers. If the altitude of the water surface is higher than the ground altitude, the vertex is submerged. If the ground altitude of the river vertex is higher than the water altitude, the vertex is not submerged, but its edges may still be partially submerged by adjacent submerged vertices. While executing a split operation, the ground altitude and the water altitude of v_{nv} are calculated from the values of the adjacent vertices v_t, v_d, v_{ev1} and v_{ev2}. As a result, polygons that have both submerged vertices and vertices above water level are partially flooded and form coasts and river banks. We use a 2D lookup texture to define colors for the climate zones and water depth. The ground altitude, water altitude and geographical position of the vertex are used to compute the texture coordinates. A high quality shader is used to create the water effects.

4. Planet Generation

Base mesh creation must be fast as the user wants to start exploring the planet without delay but still all information for refining the geometry has to be generated. This process consists of two steps: First, we generate a base shape and the land masses. Then we produce the inital river networks.

4.1. Base Shape and Continents

The base shape is created by inserting vertices into an octahedron to form a sphere. During each split, each new vertex $\mathbf{v_{nv}}$ needs to be lifted to the surface of the base shape:

$$\mathbf{v_{nv}} \leftarrow (r + a_{nv}) \frac{\mathbf{v_{nv}} - \mathbf{c}}{\|\mathbf{v_{nv}} - \mathbf{c}\|} + \mathbf{c}, \tag{1}$$

where r is the sphere's radius, a_{nv} is the vertex's ground altitude, and $\mathbf{c}$ is the sphere's center. Positions for new vertices $\mathbf{v_{nv}}$ are chosen from a randomized weighted sum of the surrounding vertices:

$$\mathbf{v_{nv}} = (1-\eta)\,(\xi v_{ev1} + (1-\xi) v_t) + \eta(\xi v_d + (1-\xi) v_{ev2}), \tag{2}$$

where $\xi, \eta \in [0.25, 0.75[$ are uniformly distributed pseudo-random numbers.

In this context, a continent is a connected land mass above sea level. Initially, all faces are labeled as sea. For every continent, a starting face is selected and labeled as a continent. Faces that have at most one pure sea vertex can be added to the continent. The other vertices in a new face must already belong to the continent. This ensures that two continents are always separated by an edge. The face and its edges are labeled to belong to the continent. This is repeated until the percentage of the total land mass reaches a user-defined

value. Any edges and vertices between continental and sea faces are marked as coast. Alternatively, instead of creating a random base mesh, types of polygons, edges and vertices are read from a digital elevation model.

Ground altitudes for continental and coastal vertices are assigned during river network creation and are initialized with zero. Sea vertices are assigned an altitude below sea level and a water altitude that equals the sea level, ε_{sea}. If terrain below sea level is also required, pure midpoint displacement can be used to compute altitudes. In order to generate pseudo random numbers for the vertices in a reproducible manner, each vertex is assigned an initial random seed and its split count is set to zero.

4.2. Initial River Networks

At this point, continents consist only of continent and coast edges and vertices. We still need to generate river networks and compute continental vertex altitudes. First we define the maximal depth of the rivers $\varepsilon_{river} < \varepsilon_{sea}$. Creating the river networks starts at the river mouths. We consider all vertices in pseudo-random order, looking for continental vertices that are adjacent to a coast vertex. In a river mouth, typically only one river mouths into the sea, therefore the coast vertex should not have a river edge yet. The edge between the chosen vertices is flagged as a river edge. In order to complete the river networks, we pick edges that connect a river vertex with a continent vertex in pseudo-random order and convert these edges to river edges. Two rivers may merge in a river vertex, but if possible, alternative river edges should be used to prevent merging more than two rivers in a single vertex. When all continental vertices have been connected to the river network, the river networks are complete.

While the river network is created, ground altitudes and water altitudes are assigned to the river vertices, starting from the coast vertices at sea level:

$$a_v = a_u + e_a l_e \xi, \quad e_a = \frac{a_{max\_river}}{l_r}, \tag{3}$$

$$w_v = a_v + e_w l_e, \quad e_w = \frac{\varepsilon_{river}}{l_{cr}}, \tag{4}$$

where v is the current vertex, u is reached by v's outgoing river edge, a_u, a_v are the ground altitudes, w_v is the water altitude of v, average ground elevation e_a, average water elevation e_w, river length l_r, length of the river between v and river spring l_{cr}, length of the current edge l_e and $\xi \in [0,1[$ is a uniformly distributed pseudo-random number. Assigning river altitudes using a constant elevation leads to sharp cliffs in places where a branch of a long river neighbors a branch of a smaller river. Instead, we assign a maximum altitude a_{max_river} for the river spring, depending on the length of the river. The ground altitude of the river mouth is $\varepsilon_{sea} - \varepsilon_{river}$. The ground altitudes of the river springs are not allowed to exceed a_{max_river} and the water altitude is equal the ground altitude. The altitude for all other river vertices is interpolated linearly between these (see Figure 3). Vertex normals are stored for lighting. As river vertices are hidden, this variable is used to store a tangent towards the river mouth instead, which is used to generate round rivers during later vertex splits.

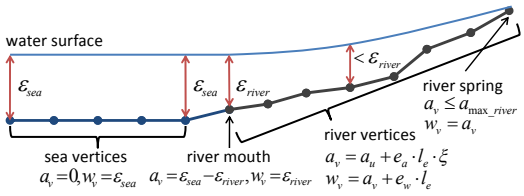

Figure 3: *Water and ground altitudes of the river vertices.*

Now, the river networks are separated by continent or coast edges but there are no river beds. Therefore, we insert a continent vertex into every continent edge between two river vertices or between a river vertex and a coast vertex. Coast edges with two river mouth vertices must be split in the same manner. While producing the river networks, pure mountain vertices are inserted to separate the rivers. These vertices are placed at higher altitude than their surrounding river vertices, so we compute an altitude a_{nv} for $\mathbf{v_{nv}}$ using the altitude a_r of the highest adjacent river vertex $\mathbf{v_r}$:

$$a_{nv} = a_r + e_m l_e \xi, \tag{5}$$

where e_m is the elevation of mountain edges, l_e is the horizontal length of the edge between $\mathbf{v_{nv}}$ and $\mathbf{v_r}$, and $\xi \in [0,1[$ is a uniformly distributed random number. This ensures that rivers follow the steepest decline, because the rivers are surrounded by continental vertices at higher altitude. Table 2 gives an overview of the parameters used by our algorithm.

Symbol/Name	Description	Value
Basic shape	Sphere, ring or flat polygons	Sphere
r	Size or radius	6371 km
Continents	Number of continents (optionally islands)	6
Land	Percentage of land area to total planet's surface	50%
Bitmap	Alternatively, to define the continents	
Edge length	Minimum edge length	1 cm
Base shape accuracy	Number of triangles for base shape	5000
2D Lookup Texture	To define colors for the climate zones	
a_{max_river}	Max. river altitude	12 km
$a_{max\_mountain}$	Max. mountain altitude	13 km
ε_{sea}	Global water level (sea level)	3 km
ε_{river}	Max. river depth	300 m
l_{rmin}	Min. river length	200 km
s_r	Min. river slope	1 m/km
s_{rb}	Min. riverbed slope	300 m/km
s_p	Min. slope near river	1 m/km
ε_{rb}	Riverbed edge height above water	10 m
s_g	Max. ground slope	1000 m/km

Table 2: *Main parameters used to define a planet in our algorithm.*

5. Runtime Algorithm

The adaption algorithm is divided into several consecutive steps to take advantage of massively parallel hardware. The

partitioning is required for thread synchronization while each step can be processed completely in parallel. First, we test which edges have to be split and which vertices must be collapsed to adapt the mesh to the new camera position. Then the selected operations are performed. This mesh is then used as input for the next frame to exploit temporal coherence.

The main data structures required for rendering are the vertex buffer containing the position and normals and the index buffer storing the connectivity of the adapted mesh. Both are stored as vertex buffer objects (VBOs) and are therefore separate from all other data. Table 3 gives an overview of our dynamic data structures that are discussed in the following.

Buffers	Elements	Memory (bytes per entity)	New entities (per new vertex)
active edges	face ID (×2)	8	3
	length	4	
	split count	2	
	type	1	
active faces	index VBO	12	2
	edge ID (×3)	12	
	normal	12	
	type	1	
active vertices	vertex VBO	24	1
	ground altitude	4	
	water altitude	4	
	edge ID	4	
	maximal edge length	4	
	seed	4	
	type	1	
	coast marker	1	
temporary	split flag	4	3
	collapse flag	4	1
	temp	4	3
Total memory (per vertex)			193 byte

Table 3: *Elements of the data structure.*

5.1. Vertex State Update

In the first step, we determine the necessary operations. If the vertex v needs to be split according to its refinement criteria, we set the *split* flag in its state. Otherwise, we set the *collapse* flag if the refinement criteria allow a collapse. In all other cases no operation is required for v. Then we determine the state of the edges by traversing all edges. An edge is split if one of the edge vertices meets the criteria for a split. As some splits and collapses cannot be executed immediately, an additional check has to be performed to remove conflicting operations. For the split operations, only the edge with the lowest split level can be split in each face f. In addition, if any edge is marked for splitting, no vertex of f can be collapsed. If no edge of the face needs to be split, we check the collapse operations. v_{nv} can only be collapsed if all incident edges (e_1, e_2, e_3 and e_4) have the same split level. This assures that only one edge in a face can be split and only one vertex can be collapsed to avoid race conditions. Split and collapse operations can be executed in parallel provided that each triangle is affected by only one operation. If there are several operations affecting a single triangle, the most important operation is executed immediately, while the other operations may be executed in one of the following frames. Due to camera movement, the priority of delayed operations may change. Splitting all three edges on a triangle would take three frames.

We check several criteria to remove invisible vertices. The most simple one is view frustum culling: A vertex can be collapsed if it lies outside the view frustum regardless of the screen space error. To prevent foldovers and popping artifacts when rotating or panning, we do however not simply collapse all vertices that are outside of the view frustum but modify the distance d of these vertices for the following LOD selection:

$$\tilde{d} = \left(c_{LOD} \left(\frac{\max(|x|,|y|,|z|)}{w} - 1 \right)^2 + 1 \right) d, \qquad (6)$$

where x, y, z and w are the homogeneous coordinates of the vertex after projective transformation and c_{LOD} is a constant value. In our experiments, $c_{LOD} = 20$ resulted in a smooth LOD falloff outside the view frustum. Then we perform backface culling and evaluate the screen space error. For splitting and merging vertices based on camera distance, we test $\frac{l}{\tilde{d}} > c$, where l is the maximal length of the edge assigned to the vertex (stored) and c is a constant value, and set $split/collapse$ flags according to the result of the test. Different values of c are used for sea and continent vertices because the sea can be rendered at a lower resolution to reach the same quality as the land. Algorithm 1 summarizes the vertex update operations.

```
foreach vertex v in parallel do
    if need_split(v)
        mark(v, split)
    elif may_collapse(v)
        mark(v, collapse)
foreach edge e in parallel do
    determine_edges_state(e,v_ev1,v_ev2)
    if edge_marked(e, split)
        level_min = get_min_active_split_level(f)
        unmark_dependent_splits(f, level_min)
        unmark_all_collapses(f)
foreach vertex v in parallel do
    if any_vertex_marked(v, collapse)
        if neighboring_edges_level_not_equals(v_nv, e1-e4)
            unmark_collapse(v_nv)
```

Algorithm 1: The parallel vertex states update.

5.2. Memory Management

Before split and collapse operations can be performed, we may need to adjust the size of the buffers. We always reserve slightly more memory than currently required to reduce the runtime cost for allocating memory and copying data when the size of an array is modified. If the size of the vertex, edge or face buffers is too small or significantly too large, new buffers are allocated and the content of the old ones is copied into them. When a reallocation is performed, the buffer size

is set to the number of faces n_f plus a user-defined threshold n_{alloc}. If the buffer is larger than $n_f + 2n_{alloc}$ it is reduced to $n_f + n_{alloc}$.

5.3. Parallel Edge Splits

After updating the state of all active edges and removing illegal splits and collapses, the operations can be applied. We first compact the splits [SHZO07] to ensure that each thread performs an operation to improve the thread utilization. Then, the following steps are performed for each split operation (summarized in Algorithm 2):

1. Calculate the seed $s_{nv} = s_{ev1} + s_{ev2}$ of the new vertex v_{nv}, where s_{nv} is the seed of vertex v_{nv}.
2. Two new faces f_3 and f_4 and three new edges e_2, e_3 and e_4 are generated and added to the buffers (Figure 2).
3. Change the connectivity of neighboring faces edges and vertices, as demonstrated in Figure 2.
4. Assign types to the new faces, edges and vertices using the rules from section 3.2. We assign the type of f_1 to the new face f_3 and new edge e_2. Similarly, the type of f_4 and e_4 is the type of f_2.
5. The position of the sea vertices is the center of the edge that is split. For continent and coast vertices the position of v_{nv} is calculated from the adjacent vertices v_{ev1}, v_{ev2}, v_t, and v_d using equation 2. The position of the river vertices is calculated from v_t and v_d only:

$$\mathbf{v_{nv}} = (1-\xi)v_t + \xi v_d, \tag{7}$$

where again $\xi \in [0.25, 0.75[$ is a uniformly distributed pseudo-random number, calculated with a seed of v_{nv}. In addition, ξ is biased to generate smooth rivers. If the edge length is less than the minimal river length l_{rmin} we use:

$$\xi' = \xi + \frac{(v_s - v_d)\cdot(v_t - v_d)}{\|v_t - v_d\|^2}, \tag{8}$$

with

$$v_s = \frac{v_{ev1} + v_{ev2}}{2} + \|v_{ev1} - v_{ev2}\|\frac{t_{ev1} - t_{ev2}}{4}, \tag{9}$$

where t_{ev1} and t_{ev2} are the stored tangents of v_{ev1} and v_{ev2}. Equation 9 assumes that the river flows from v_{ev1} to v_{ev2}. Additionally, the tangent is calculated for the new river vertex v_{nv} and stored instead of the normal. Finally, v_{nv} is added to the vertex buffer.
6. Calculate the ground altitude a_{nv} and water altitude w_{nv} of v_{nv}. The altitude of sea vertices is simply the sea bottom and the water altitude the sea level. For river vertices it is $\frac{1}{2}(w_{ev1} + w_{ev2})$ and $\frac{1}{4}(w_{ev1} + w_{ev2} + w_{vt} + w_{vd})$ for all others. For continent vertices we then check if we can construct a new river arm, where we set w_{nv} to a_{nv}. It can be generated if the split edge is not a river edge and one of the four edges can be converted into a river. This is only possible if the edge is longer than l_{rmin} and a_{nv} can be at least $l_e s_r$ above the other river vertex and below all neighboring non-river vertices c:

$$a_{nv} < \min_c (a_c - \min(l_e s_{rb}, \varepsilon_{rb} + (l_e s_p),)). \tag{10}$$

where l_e length of the edge between v_{nv} and neighboring non-river vertex v_c, s_r minimal river slope, s_{rb} minimal riverbed slope, s_p minimal slope near river and ε_{rb} riverbed edge height above water. For continent vertices similar bounds apply (see Figure 4):

$$a_{nv} > \min_{bw} (\min(a_{bw} + l_e s_{rb}, w_{bw}\varepsilon_{rb} + (l_e s_p))), \tag{11}$$

$$\max_c (a_c - (l_e s_g)) < a_{nv} < \min_c (a_c + (l_e s_g)), \tag{12}$$

where s_g maximal ground slope, bw are the neighboring river vertices or those covered by water and c the remaining ones. The final altitude is then a random value inside the previously computed bounds.
7. Calculate normals for f_1, f_2, f_3 and f_4 and the vertex normal of v_{nv} unless it is a river vertex. We compute normals for each vertex from the surrounding face normals.

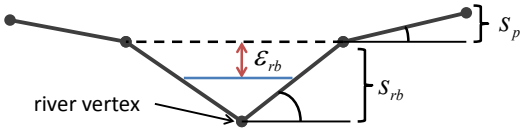

Figure 4: *Riverbed shaping parameters.*

```
compact(splits)
foreach split edge e in parallel do
    calc_seed(v_nv)
    add_new_faces_edges_vertex(e1-e4, f1-f4, v_nv)
    relink_neighbors()
    assign_types(e1-e4, f1-f4, v_nv)
    calculate_altitudes(v_nv)
    v_nv = split_edge(e)
    calculate_faces_normals(f1-f4)
```

Algorithm 2: Parallel edge split algorithm.

5.4. Parallel Vertex Collapses

After applying the split operations, the collapse operations need to be performed. Each collapse operation removes vertex v_{nv} and edges e_2, e_3 and e_4. In addition, faces f_3 and f_4 degenerate and are removed from the mesh. Then faces f_1 and f_2, edge e_1, and all incident faces and edges are relinked (Figure 2). We use the member variable `edgeID` in each vertex to store an edge that contains the vertex. This helps us locate the other edges around the vertex quickly for collapse operations. When all operations have been applied, the maximal length of the adjacent edge of the new vertices and the vertices in the neighborhood of the split and collapse operations need to be recalculated. Algorithm 3 shows the parallel processing of the edge collapse operations and how maximal lengths are recalculated.

```
foreach vertex v in parallel do
    if marked(v, collapse)
        remove_faces_edges_vertex(v)
        relink_neighbors(v)
foreach vertex v in parallel do
    if required_length_recalc(v)
        recalc_max_length(v)
```

Algorithm 3: Parallel edge collapse algorithm.

5.5. Buffer Compaction

If vertices were removed, the active vertices (including the vertex VBO), active faces (including the index VBO), and active edges are compacted in the final step of adaption. Note that when compacting the vertices, faces or edges, the references to them must be updated accordingly. While the compaction of the faces and thus the indices is mandatory since the index VBO are used for rendering, the compaction of the vertices and edges is not. The latter two only need to be compacted every few frames to prevent bloating of the buffers. To save time and memory, we use a specialized in-place compaction algorithm [DMG10] since the ordering does not need to be preserved.

6. Results

Our test system consists of a 3.333 GHz Intel Core i7-980X CPU with 6 GB of DDR3-1333 main memory, 16 lanes PCIe 2.0 slot, and a GeForce GTX 580 (841/4200MHz). OpenGL is used for rendering and CUDA for the adaption algorithm. Terrains were reproducible over networks and on different PCs. All images in this paper were generated under real-time conditions at a resolution of 1920×1080 with an edge length of 0.5 pixel for the land and 5 pixels for the water. We limit the number of polygons dynamically to a value that ensures that 20 to 30 frames can be rendered per second, to adjust the algorithm to the capabilities of different hardware.

Table 4 lists the number of rendered faces, the total time (rendering and adaption per frame), total and adapt number of triangles per second (TPS), and the memory consumption for the views shown in Figure 5 and the fly through in the accompanying video, where the numbers are taken from the most complex frame. We used a base mesh with approximately 5000 faces. Creating the base mesh took 0.27 seconds. Table 2 shows the values that were used to produce the accompanying results. While the user explores the planet, the dynamic data structures reside on the graphics card only. This has the advantage that the data can be rendered and adapted without passing it over the PCIe bus. We can process up to 34/100 (total/adapt) M TPS for static views. We use high quality shaders to demonstrate that our algorithm is suitable for real-time rendering of terrains at high quality. However, the shaders require 50% to 80% of the rendering time. With simpler shaders, the adaption time lies between at 40% to 70% of the total frame time.

model	*# rendered faces*	*memory (MB)*	*frame time (ms)*	*total/adapt M TPS*
orbit view	1,153,435	115.8	35.8 (38.3%)	32.2/84.1
ground view 1	1,296,562	130.2	38.5 (33.6%)	33.7/100.2
ground view 2	858,562	92.6	33.8 (36.9%)	25.4/68.8
video (max.)	1,041,970	139.5	58.4 (46.8%)	17.8/38.1

Table 4: *Memory consumption, total rendering time and total number of triangles per second (TPS) of the different views. The ratio of adaption time compared to total time is given in parenthesis.*

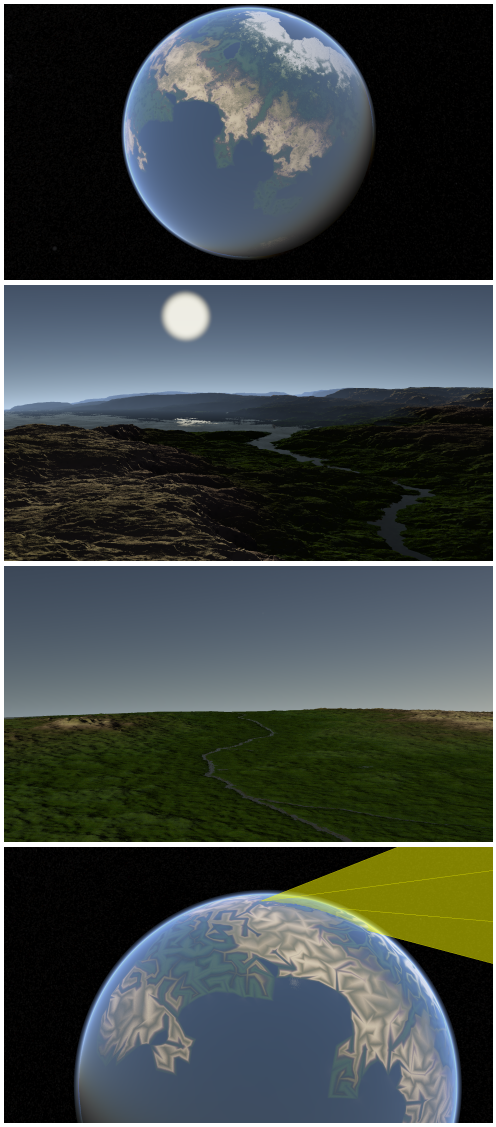

Figure 5: *The images show the models as rendered from the point of view. The bottom image demonstrates that parts of the terrain that are outside the view frustum (yellow) are reduced to the base mesh.*

Timings for adaption and rendering together with memory consumption and the number of active faces for the fly through in the accompanying video are shown in Figure 6. The data structures consumed less than 139.5 MB and the average frame rate is 30 frames per second (fps). The peak performance for dynamic views is 27/72 MTPS (total/adapt) and up to 7/15 MTPS can be generated. Our approach quickly reacts to changes of the view direction with fast adaption of the terrain complexity. Due to the high adaption performance, only few popping artifacts are visible in the video despite the fast movements. Figures 1, 5, and 8 show example planets.

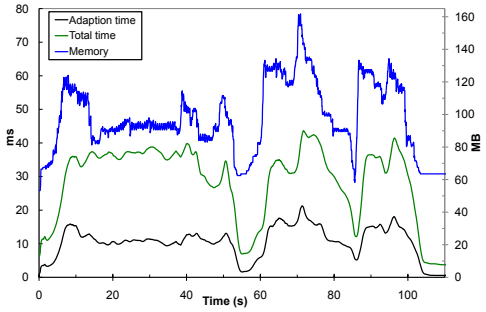

Figure 6: *Timings and memory consumption for the scene using a pre-recorded camera path.*

Finally, we analyze the runtime of each step inside the adaption and rendering algorithm in Figure 7. The most expensive step of our algorithm is the state update, because it must be performed for each active vertex. The time spent on mapping and unmapping the index and vertex buffers for access by CUDA/OpenGL cannot be reduced or prevented. Rendering takes up to 63% of the frame time.

7. Discussion and Conclusion

We have proposed a new procedural algorithm that spontaneously creates planets or terrain parts with continents and

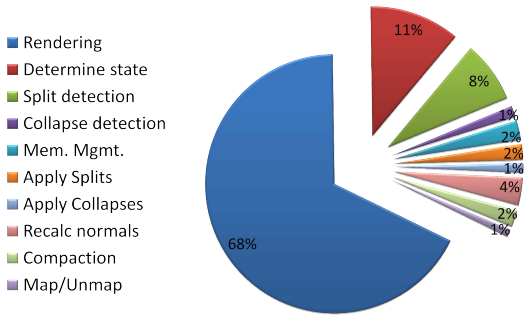

Figure 7: *Relative time of the several adaption steps compared to rendering.*

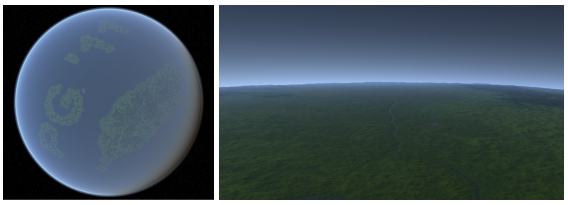

Figure 8: *Left: Alternatively, our algorithm can be used to generate river networks for user-provided maps. Right: Meandering rivers.*

realistic river networks. It is specifically developed for massively parallel view-dependent adaption. While previous algorithms are not able to generate planets at adaptive level of detail within seconds while ensuring consistency of the river networks, we are the first to present a parallelized pipeline that combines these features. The algorithm correctly models how valleys and mountain tops differ in that rivers flow through the valleys. While many applications use a fixed level of detail, interactively adapting the level of detail to the camera perspective is a necessity when dealing with large terrains, such as planets. The algorithm does not require storage except for the mesh representing the terrain, and the geometry does not have to be streamed over network, even when several users wish to explore the same terrain. Only a small parameter set and an initial random seed are needed to completely recreate a planet because of reproducibility.

The system by Kelley et al. produces only a single river network, and it does not support refinement [KMN88]. Our technique is suitable for interactively adapting large terrains to a moving camera. The only other technique that can do this is the midpoint displacement algorithm by Fournier et al. [FFC82], which does not produce river networks. When using midpoint displacement on realistic terrain, rivers may be interrupted by mountains, which would likely block the rivers, resulting in large endorheic basins or inconsistencies. In contrast to that, the rules implemented by the proposed algorithm ensure consistency of the river networks.

Erosion simulation can also produce eroded terrain with continents and river networks, but that family of algorithms requires much more computation time and has to run supervised. Otherwise, either too many large lakes remain, or the rivers carve too deeply into the terrain. Bad choices for the parameter values often mean that the entire simulation must be restarted, whereas the presented algorithm produces a viable solution much faster. While it may be possible to combine erosion simulation with adaptive level of detail, ultimately similar problems would have to be solved, and users might note changes in the terrain caused by the simulation. In contrast to that, the terrain generated by our algorithm is immediately realistic and stable.

Our system yields correct results for glaciofluvial erosion. Thermal and eolian erosion are modeled by limiting terrain

slope. These effects are computed by our method without time-intensive simulations. The algorithm can be used for simulations that require spontaneously created terrain. This includes computer games and learning to steer vehicles, e.g. flight school.

References

[Bel07] BELHADJ F.: Terrain modeling: a constrained fractal model. In *Proc. of AFRIGRAPH 07* (2007), AFRIGRAPH '07, pp. 197–204. 2, 3

[BF01a] BENEŠ B., FORSBACH R.: Layered data representation for visual simulation of terrain erosion. In *SCCG '01: Proceedings of the 17th Spring conference on Computer graphics* (2001), p. 80. 2, 3

[BF01b] BENEŠ B., FORSBACH R.: Parallel implementation of terrain erosion applied to the surface of mars. In *Proc. of AFRIGRAPH 01* (2001), pp. 53–57. 2

[BF02] BENEŠ B., FORSBACH R.: Visual simulation of hydraulic erosion. In *WSCG* (2002), pp. 79–94. 3

[BSS06] BROSZ J., SAMAVATI F. F., SOUSA M. C.: Terrain synthesis by-example. In *Proceedings of the first International Conference on Computer Graphics Theory and Applications* (2006). 2, 3

[BW06] BOKELOH M., WAND M.: Hardware accelerated multiresolution geometry synthesis. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games* (2006), I3D '06, pp. 191–198. 2

[CMF97] CHIBA N., MURAOKA K., FUJITA K.: An erosion model based on velocity fields for the visual simulation of mountain scenery. *The Journal of Visualization and Computer Animation 9*, 4 (1997), 185 – 194. 3

[DMG10] DERZAPF E., MENZEL N., GUTHE M.: Parallel view-dependent refinement of compact progressive meshes. In *Eurographics Symposium on Parallel Graphics and Visualization* (2010), Eurographics Association, pp. 53–62. 3, 8

[DWS*97] DUCHAINEAU M., WOLINSKY M., SIGETI D., MILLER M., ALDRICH C., MINEEV-WEINSTEIN M.: ROAMing terrain: real-time optimally adapting meshes. In *Proceedings of the 8th Conference on Visualization'97* (1997), pp. 81–88. 3

[EF01] EFROS A. A., FREEMAN W. T.: Image quilting for texture synthesis and transfer. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), SIGGRAPH '01, pp. 341–346. 2

[ESV99] EL-SANA J., VARSHNEY A.: Generalized view-dependent simplification. *Computer Graphics Forum 18*, 3 (1999), 83–94. 4

[FFC82] FOURNIER A., FUSSELL D., CARPENTER L.: Computer rendering of stochastic models. *Communications of the ACM 25*, 6 (1982), 371–384. 2, 9

[GPMG10] GALIN E., PEYTAVIE A., MARCHAL N., GUÉRIN E.: Procedural generation of roads. In *Computer Graphics Forum: Proceedings of Eurographics* (2010), vol. 29. 2

[HGA*10] HNAIDI H., GUÉRIN E., AKKOUCHE S., PEYTAVIE A., GALIN E.: Feature based terrain generation using diffusion equation. *Computer Graphics Forum 29*, 7 (2010), 2179–2186. 2, 3

[Hop96] HOPPE H.: Progressive meshes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (1996), pp. 99–108. 3

[Hop97] HOPPE H.: View-dependent refinement of progressive meshes. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997), pp. 189–198. 3

[Hop98] HOPPE H.: Efficient implementation of progressive meshes. *Computers & Graphics 22*, 1 (1998), 27–36. 3

[HSH09] HU L., SANDER P. V., HOPPE H.: Parallel view-dependent refinement of progressive meshes. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games* (2009), pp. 169–176. 3

[KBKv09] KRISTOF P., BENEŠ B., KRIVANEK J., ŠT'AVA O.: Hydraulic erosion using smoothed particle hydrodynamics. In *Proceedings of Eurographics 2009: Computer Graphics Forum 28 (2)* (2009). 3

[KMN88] KELLEY A. D., MALIN M. C., NIELSON G. M.: Terrain simulation using a model of stream erosion. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques* (1988), pp. 263–268. 2, 9

[Los04] LOSASSO F.: Geometry clipmaps: terrain rendering using nested regular grids. *ACM Transactions on Graphics 23* (2004), 769–776. 3

[LSWW11] LIPP M., SCHERZER D., WONKA P., WIMMER M.: Interactive modeling of city layouts using layers of procedural content. *Computer Graphics Forum (EG 2011)* (2011). 2

[Man83] MANDELBROT B. B.: *The fractal geometry of nature*. Freeman and Company, New York, 1983. 2

[MKM89] MUSGRAVE F. K., KOLB C. E., MACE R. S.: The synthesis and rendering of eroded fractal terrains. *SIGGRAPH Computer Graphics 23*, 3 (1989), 41–50. 2

[Paj01] PAJAROLA R.: Fastmesh: Efficient view-dependent meshing. In *9th Pacific Conference on Computer Graphics and Applications* (2001), pp. 20–30. 3

[PD04] PAJAROLA R., DECORO C.: Efficient implementation of real-time view-dependent multiresolution meshing. *IEEE Transactions on Visualization and Computer Graphics 10*, 3 (2004), 353–368. 3

[PGTG04] POUDEROUX J., GONZATO J.-C., TOBOR I., GUITTON P.: Adaptive hierarchical rbf interpolation for creating smooth digital elevation models. In *GIS '04: Proceedings of the 12th annual ACM international workshop on Geographic information systems* (2004), pp. 232–240. 2

[PH93] PRUSINKIEWICZ P., HAMMEL M.: A fractal model of mountains with rivers. In *Graphics Interface '93* (1993), pp. 174–180. 2

[SBBK08] STAVA O., BENES B., BRISBIN M., KRIVANEK J.: Interactive terrain modeling using hydraulic erosion. In *Eurographics/SIGGRAPH Symposium on Computer Animation* (2008), Gross M., James D., (Eds.), pp. 201–210. 3

[SHZO07] SENGUPTA S., HARRIS M., ZHANG Y., OWENS J. D.: Scan primitives for GPU computing. In *Graphics Hardware 2007* (2007), pp. 97–106. 7

[SS05] STACHNIAK S., STUERZLINGER W.: An algorithm for automated fractal terrain deformation. In *Proceedings of Computer Graphics and Artificial Intelligence* (2005), pp. 64–76. 2

[ST89] SZELISKI R., TERZOPOULOS D.: From splines to fractals. *SIGGRAPH Computer Graphics 23* (1989), 51–60. 2

[XV96] XIA J. C., VARSHNEY A.: Dynamic view-dependent simplification for polygonal models. In *VIS '96: Proceedings of the 7th conference on Visualization '96* (1996), p. 327 ff. 3

[ZSTR07] ZHOU H., SUN J., TURK G., REHG J. M.: Terrain synthesis from digital elevation models. *IEEE Transactions on Visualization and Computer Graphics 13*, 4 (2007), 834–848. 2