Real-Time Editing, Synthesis, and Rendering of Infinite Landscapes on GPUs

Jens Schneider, Tobias Boldte, Rüdiger Westermann

Computer Graphics & Visualization Group
Technische Universität München
Boltzmannstrasse 3, 85748 Garching
Email: {jens.schneider, westermann}@in.tum.de

Abstract

Recent advances in algorithms and graphics hardware have opened the possibility to render large terrain fields at interactive rates on commodity PCs. Due to these advances it is possible today to interactively synthesize artificial terrains using procedural descriptions. Our paper extends on this work by presenting a new GPU method for real-time editing, synthesis, and rendering of infinite landscapes exhibiting a wide range of geological structures. Our method builds upon the concept of projected grids to achieve near-optimal sampling of the landscape. We describe the integration of procedural shaders for multifractals into this approach, and we propose intuitive options to edit the shape of the resulting terrain. The method is multi-scale and adaptive in nature, and it has been extended towards infinite and spherical domains. In combination with geo-typical textures that automatically adapt to the shape being synthesized, a powerful method for the creation and rendering of realistic landscapes is presented.

1 Introduction and Related Work

Based on ideas on the organizing principles of natural phenomena [13], there exists at least empirical evidence on fractals in geological structures such as landscapes, rivers and coastlines. Such structures obey the typical fractional Brownian or $1/f^{\beta}$ "motion", and they can thus be modelled as a random walk exhibiting certain stochastic properties. Based on this observation a number of different approaches for fractal terrain synthesis have been suggested over the last decades. For a thorough overview we refer to [5]. Today, fractal models are most commonly generated using Fourier filtering [25], midpoint displacement [14] or noise synthesis [20]. As the approach used here is of the latter type, we will briefly describe the underlying concept.

Perlin [20, 21] introduced a synthetic, functional fractal model as a summation of several, appropriately scaled-down copies of a band-limited, stochastic *noise* function. The *Rescale-and-Add* method by Saupe [26, 27] extends the heuristic formulas of Perlin to a full fractal model. It allows a 2D fractal displacement noise with locally varying fractal dimension (a measure of the surface roughness) to be created by summing vector-valued Perlin noise functions at different scales and frequencies. Simultaneously Musgrave [16] introduced *noise synthesis* to enable local control of fractal dimension and other parameters of the generating random process. As these methods use multiple scalings to locally control the statistic properties of the random process underlying the fractal synthesis, the type of fractals they generate is referred to as *multifractals*.

The advantages of the functional approach are:

1 The function can be evaluated locally at only those locations required during rendering, enabling efficient terrain synthesis on programmable graphics hardware (GPUs) [7, 8].
2 The level of detail of the terrain being generated can be adapted to the local image resolution, thus anti-aliasing the fractal [19].
3 All parameters of the function may vary locally over the object domain.
4 The basis functions underlying the fractal synthesis process can be modulated to adapt locally to particular design features.

Due to the high degree of sophistication to which functional approaches for fractal modelling have developed, it is by now possible to create landscapes

at impressive realism, including many different geological formations as well as terrain specific textures [2, 18]. While being acknowledged as a fine technique as such, the most demanding part is to deal with the ”parametric nightmare” of the synthesizer's high dimensional feature space. So far, fractal landscape editors either restrict the designers flexibility or overwhelm the user with a plethora of parameters. An embedding of these parameters into a tight visual feedback loop such that the user can directly monitor the effects of editing actions is not yet available. Such an embedding allows simulating many kinds of different landscapes in an intuitive way, and helps to match the modelled scenery with the designer's expectations while shortening both the design process and time to market. Especially in applications like computer games or computer-animated films this is of particular interest.

Techniques that are open to this kind of interaction can also be directly integrated into virtual environments to continuously modify outdoor scenarios based on context. By modifying landscapes depending on external parameters like game status, player's expertise, or difficulty, the immersiveness of such scenarios can be increased significantly. Even more, they allow the user to create its own synthetic worlds that can be altered directly at runtime by individual and context-specific guidelines (similar to the *Populous* series of games [15]). As it is impossible to pre-compute the per se infinite variety of different structures, parameterized evaluation methods enabling continuous transitions between different formings have to be considered.

The reason why the aforementioned scenario has not yet become reality is twofold. Firstly, terrain generation methods featuring arbitrary local control of surface properties are in general too expensive to be used for interactive synthesis of high-resolution terrains. Secondly, the creation of synthetic images of *dynamic* terrain models is time- and memory-consuming, and is hence not suited for use in interactive environments without further ado. For instance, ray-tracing of procedural fractal height values [17, 18] is far from interactive. But even for non-procedural terrains, ray-tracing and scanline algorithms [1, 4] require *static* height fields to harness their full potential. Since scanline algorithms also involve streaming renderable primitives to the GPU, the performance is considerably limited due to bandwidth requirements.

1.1 Contribution

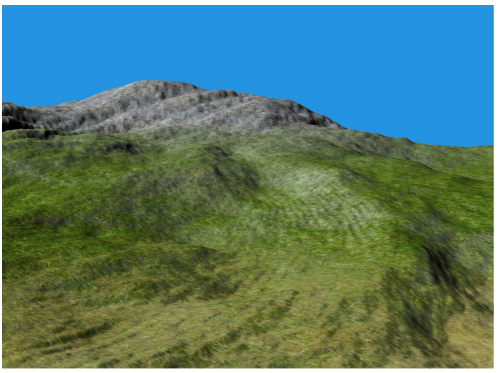

Figure 1: *A fractal landscape generated by our method. Note the striking geological features and geo-typical textures. On recent GPUs, synthesis and rendering on a 1280×1024 viewport runs at 70 fps.*

In this paper, we present a real-time fractal terrain synthesizer. The algorithms we propose are designed with respect to the aforementioned requirements. They are combined into a WYSIWYG interface to allow the intuitive design of highly detailed terrain models. Our approach involves two distinct procedures: *editing* and *rendering*, where the synthesis of the landscape is directly integrated into the rendering procedure. This avoids any intermediate data structures for storing height values other than the ones needed to form the final image. Editing and rendering involve a number of novel techniques and they provide many features not available in previous methods.

Editing: Because the user interacts directly with the same representation used to form the final image, the fractal's parameter space can be managed conveniently. We use painting and brushing on gray-scale images representing the fractal's basis functions for editing. Against common knowledge, editing basis functions can be highly intuitive, but only if coupled with immediate visual feedback. This approach turns out to be an amazingly powerful paradigm enabling multi-resolution terrain editing via the frequency bands of basis functions.

Rendering: We exploit the functional nature of the terrain, which allows for point-wise synthesis at arbitrary positions. Rather than performing the synthesis as a distinct and decoupled procedure, it is tightly integrated into the rendering process,

such that both steps can be implemented on the GPU. This exploits parallelism and memory bandwith while limiting bus transfer to compact and local editing updates. We utilize a projected grid approach to minimize both the number of point evaluations and to achieve near-optimal sampling. For this, a screen-space aligned grid is projected onto the fractal's surface by deforming grid points out of the base domain. Anti-aliasing is performed for each point by computing the appropriate level of fractal detail. To further increase the landscape's realism, geo-typical materials are synthesized *by example* (see Figure 1). For each point of the projected grid, so-called *proto-textures* [2] are combined, either by using a slope/height parameterized weighting function [3], or using editable weights.

The remainder of this paper is organized as follows. In Section 2.1 we describe the theory behind noise synthesis. We then present the WYSIWYG interface used for fractal editing in Section 3. In Section 4 we discuss synthesis and rendering of the terrain on the GPU. Finally, we conclude the paper and discuss directions for future work.

2 Multifractal Terrain Synthesis

In the following we will briefly review the theoretical basis behind noise synthesis and stochastic fractals. The functional approach described here is well suited for evaluation on recent GPUs, as it requires only simple arithmetic operations and locally contiguous texture access.

2.1 The Rescale-and-Add Method

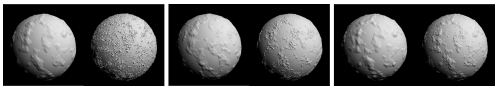

Figure 2: *Noise synthesis using Equation* (1) *with varying parameters. The three image pairs illustrate the meaning of varying lacunarity, Hurst exponent, and number of octaves (the respective values are larger in the right images).*

A random 2D grid of N(0,1) Gauss-distributed numbers with zero mean and variance equal to one, called the *noise lattice*, is generated first and stored in a 2D texture map. On the GPU this grid is extended by means of bi-linear interpolation and texture repetition into a C^1 continuous, periodic function, called *noise* [20] or, more accurately, *auxiliary function* $S(\vec{x})$ [26]. A 2D fractal is generated by superposition of several appropriately scaled-down copies of the auxiliary function:

$$\mathcal{H}(x_1, x_2) = \mathcal{H}(\vec{x}) = \sum_{k=k_0}^{k_1} \frac{1}{r^{kH}} S(r^k \vec{x}) \qquad (1)$$

The evaluation can be done for any arbitrary point independently of its neighbors. It can be efficiently performed in a pixel shader on the GPU that is parameterized with the point coordinates $\vec{x}$, the lacunarity r, and the Hurst exponent $H = 3 - D$, where D is the fractal dimension of the surface. Equipped with these parameters, the shader computes fractal height values using multiple evaluations (texture fetches) of the noise lattice. The meaning of the different parameters is illustrated in Figure 2.

The summation limits are calculated in accordance with the smallest and largest structures (frequencies) desired at a certain position of the terrain. k_0 determines the global fractal structure and is calculated only once for a sequence, while k_1 defines the smallest level of visible detail (high frequency information), and is usually changed from point to point depending on the perspective distortion. In Section 4 we will explicitly discuss how to select k_1 in a way such to avoid aliasing artifacts.

2.2 Domain Warping

The Rescale-and-Add method creates convincing terrain fields on medium scales. On larger scales, however, the result is too homogeneous to appear realistic. The reason is that the auxiliary function $S(\vec{x})$ is designed to be stochastically invariant under rotation, translation, and scaling. This results in very regular and artificially looking geo-structures.

A more natural appearance can be achieved by using *domain warping*. Instead of synthesizing the terrain over the domain $\mathcal{D} \subseteq \mathbb{R}^2$, a continuous re-parametrization $\Phi : \mathcal{D} \mapsto \mathcal{D}$ is utilized, on which the synthesis is then performed. In [5] such a mapping was suggested to simulate breaking waves on a 2D-only domain embedded in 3D. To overcome the homogeneous structure of the terrain we suggest roughness- and height-dependent rotation and translation of the domain. This breaks up the stochastic invariances of the basis functions in a controlled and restricted way.

We therefore assume that each domain coordinate (x_1, x_2) is transformed by a general rotation/translation matrix (where ϕ is the angle of rotation and $\vec{t}$ the translation vector):

$$\begin{pmatrix} x_1' \\ x_2' \end{pmatrix} = \begin{pmatrix} \cos\phi & -\sin\phi & t_1 \\ \sin\phi & \cos\phi & t_2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}$$

To avoid evaluations of trigonometric functions we represent the rotation by a unit vector $\vec{r} = (\cos\phi, \sin\phi)$:

$$\begin{pmatrix} x_1' \\ x_2' \end{pmatrix} = \begin{pmatrix} r_1 & -r_2 & t_1 \\ r_2 & r_1 & t_2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}$$

Observing that geological structures and formations in nature vary depending on surface height [16], we introduce a height-dependent roughness $\mathcal{R}_i = \mathcal{R}_0 \cdot \mathcal{H}_i$, where $\mathcal{R}_0 := 1/(r \cdot H)$. $\mathcal{H}_i$ is the height of a surface point after evaluation of the first i terms of Equation (1), i.e., the height after accumulating octave i. To add domain warping to the synthesis process we apply a different mapping depending on $\mathcal{R}_i$ for every octave i:

$$\vec{r}_{i+1} = \vec{c_1} \cdot \mathcal{R}_i + \vec{c_2} + \vec{r}_i$$
$$\vec{t}_{i+1} = \vec{c_3} \cdot \frac{\mathcal{H}_i}{\mathcal{R}_i}$$

The rotation vector $\vec{r}_{i+1}$ is re-normalized after every iteration. To perform the above updates, three additional constant 2D vectors $\vec{c_1}$, $\vec{c_2}$, and $\vec{c_3}$ are introduced. $\vec{c_1}$ controls the amount of rotation determined by the current roughness, while $\vec{c_2}$ performs a constant update. The third vector $\vec{c_3}$ controls the influence of the current height and roughness on the translation. The basic idea is that the amount of rotation increases with roughness, resulting in turbulent structures, similar to cooled-down lava, while stretching those regions more significantly that are less rough or at higher altitudes to additionally make them appear *washed out*.

Even though this procedure comes at the expense of evaluating more parameters in the innermost loop of the synthesizer, it effectively breaks up the homogeneity of the terrain, since it adds organic shapes that resemble natural geo-evolution. Two examples that show the effects of the domain warping described here are shown in Figure 3. These examples were generated using $\vec{c_1} = (0.35, 0.16)^t$, $\vec{c_2} = (-0.07, 0.13)^t$, $\vec{c_3} = (0.11, 0.17)^t$, $\vec{r}_0 = (1, 0)$, and $\vec{t}_0 = (0, 0)$. Different formations can be achieved by modifying the control vectors $\vec{c_i}$ using the fractal editor described in the next chapter.

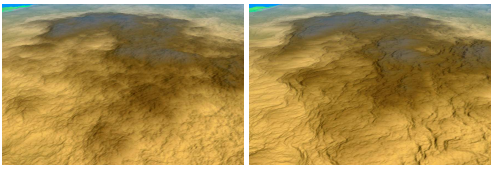

Figure 3: *The effect of domain warping is illustrated. A simple noise function was used for the auxiliary function.* ***Left:*** *Without domain warping the terrain looks rather homogeneous and uniform.* ***Right:*** *By using domain warping curved, lava-like features are generated.*

3 Interactive Fractal Editing

An important feature of the proposed system is the fractal editor. In the spirit of a WYSIWYG interface, the editor provides the user with immediate visual feedback to each action. The user can literally paint basis functions represented by gray-scale images. Several of these gray-scale images I_i, replacing the aforementioned *noise lattice*, are then composed into the auxiliary function $S(\vec{x})$ (see Equation (1)) using individual weights w_i:

$$S(\vec{x}) = \sum_i w_i \cdot I_i(x) \qquad (2)$$

where the w_i are renormalized on the GPU to sum up to 1 after each editing command. Thus $S(\vec{x})$ is a convex combination of basis functions, allowing to achieve a particular look of the terrain such as 30% desert and 70% craters easily. To offer maximum flexibility, the user can also choose to load images from disk to act as basis functions or weights. This keeps the interface intuitive and simple, yet offers the user the possibility to resort to any painting or imaging program. In addition to standard painting tools, a series of image filters such as low- and high-pass and normalizing filters is implemented.

Internally, all basis functions and weights are stored in 2D textures. Exploiting the fact that textures may comprise up to four channels, and that basis functions and weights will always be fetched at the same position to compute Equations (1) and (2), weights and basis functions are packed together to minimize fetches from different textures. This allows the various editing features as well as the filters to be implemented highly efficient on the GPU.

Also, bus transfer between CPU and GPU is largely avoided, allowing the rapid visual feedback needed to quickly design new virtual worlds.

Subsequently we will refer to this part of the editor as the *fine-scale synthesizer* because it controls the *global look* of the fractal terrain. This is achieved by repeating the texture globally over the 2D base domain, which also allows infinite landscapes to be generated. However, care has to be taken to avoid artifacts due to texture boundaries. This ensures that images are always tiling by offering only painting operations that wrap around the image in two dimensions. If the user chooses to load an image that is not tiling, the lacunarity r as well as the domain warping can be adjusted very easily to generate artifact-free images.

To decouple the fine-scale appearance from the base shape, i.e., the appearance of structures vs. their positions, a low-frequency fractal height field is added to the structures being generated by the fine-scale synthesizer. As base-level and detail are largely uncorrelated, the designer can roughly develop a prototype of the shape of the landscape by sketching mountains, valleys, seas and oceans on a coarse-resolution base domain. The final look-and-feel of the landscape is then modelled by means of the fine-scale synthesizer.

The benefits of this basis function oriented approach are obvious: Interacting with gray-scale images representing weights or heights is highly intuitive, and permits various external tools to be used in order to achieve the desired look rapidly. Furthermore, all images to describe a fractal planet can be compressed using standard image compression schemes. This could become interesting in the near future, since 3D gaming becomes uqiquitous [22], yet traditionally handheld devices feature only narrow communication channels.

The editing system also provides control over fractal parameters, such as roughness, lacunarity, and water level, as well as the vectors used for domain warping. Due to the intuitive use of all editing options including direct visual feedback, user experiments have shown that persons not familiar with the editor achieve very convincing results within only a few minutes. On the colorplate (Figure 8 right) a snapshot of a typical editing session is shown, including a fine-scale input texture (top-left), a low-frequency sketch-pad (bottom-left), and the final editing result (right).

4 Rendering

The renderer evaluates the fractal height field procedurally only for the visible fraction of the terrain. For every vertex in the viewport, Equations (1) and 2 are evaluated by using the input textures and fractal parameters as specified by the user. The entire rendering procedure is performed on the GPU by means of shader programs. To determine the vertices that lie within the view port, a projected grid [9, 10] is utilized and extended towards the rendering of spherical domains.

4.1 Projected Grid and LOD

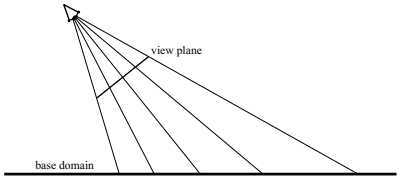

Figure 4: *The basic idea of the projected grid is to start with a regular grid in screen space, to project this grid onto the base domain, and to deform the vertices of this grid out of the base domain according to the height values.*

The basic idea of the projected grid is as follows.

1. Start with an uniform grid in screen space.
2. Project this grid onto the base domain.
3. Evaluate the height at the respective domain coordinate and displace the grid.
4. Render the displaced grid.

This method has some beneficial properties. Firstly, the projected grid tries to optimize object space triangles such that they project to approximately the same area in screen space. Secondly, since the topology of the grid is static, for a given camera angle the grid projected to the base domain can be cached on the GPU. Since no topologic restrictions apply on a per-vertex basis, vertices can be processed independently of each other. Thirdly, the amount of triangles and thus the amount of workload on both the CPU and GPU is known a priori, and can be adapted to the available processing power easily. However, there are also some drawbacks. The computation of normals on the projected grid is more involved than on a regular grid. The grid also has to be extended slightly beyond the vertices visible to avoid holes at the viewport boundaries. In order to circumvent the latter problem, the maximum height of the terrain needs to be known a priori (see Figure 5). Then a second,

so-called *projector frustum* is used, which is constrained to match the camera frustum as close as possible. However, it may diverge from the camera frustum for aesthetic reasons, i.e. to prevent artifacts that would otherwise occur at grazing camera angles. For more details we refer to the work of Johanson [10].

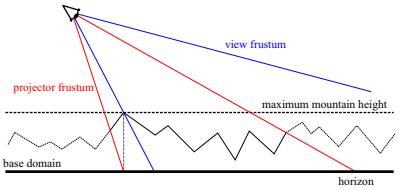

Figure 5: *If using a projected grid for rendering, the maximum height of the terrain has to be known a priori. Otherwise, features might be missed. The projector frustum is then required to include each point in the base domain* ***potentially*** *contributing to the final image.*

For each vertex of the projected grid, a level-of-detail can be computed by projecting neighboring vertices into the base domain. Since the grid is regular in screen space, only this spacing is necessary to obtain an estimate of the local object space grid spacing δ. The number of octaves λ required to evaluate Equation (1) is then obtained by:

$$\lambda = \frac{\log \delta}{H \cdot \log r}$$

λ is just the logarithm to the base r^H, the constant quotient between two consecutive amplitudes in the rescale-and-add method. As λ generally is not an integer value, we interpolate linearly between $\lfloor\lambda\rfloor$ and $\lceil\lambda\rceil$ octaves. To do so, $\lfloor\lambda\rfloor$ octaves are summed up during synthesis and the next octave weighted by the fractional part $\lambda - \lfloor\lambda\rfloor$ is added. Thus, geomorphing [6, 24] is virtually for free.

To perform fractal terrain synthesis over a spherical domain, the planar basis domain is warped around a sphere *after* the vertices have been projected. This is illustrated in Figure 6. To avoid anisotropic sampling around the sphere two concepts are used. The first is to introduce an artificial horizon behind which no landscape will be synthesized. The rationale is that atmospheric or fogging effects typically limit the viewing distance behind a certain point. Also, such an approach has traditionally offered games and interactive environments an intuitive quality vs. performance tradeoff. The second idea is to move samples closer together towards the viewer, in such a way that consecutive grid cells obtain the same angular distance with respect to the sphere's center (see Figure 6). Since each vertex already stores its relative, pre-projective grid position, no further information is needed. If the viewer moves close to the surface, we switch back to the conventional projected grid.

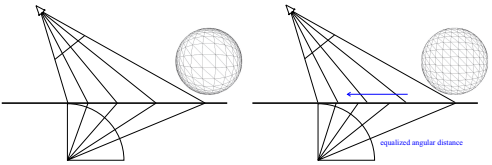

Figure 6: *Extending the projected grid to spherical domains.* ***Left:*** *A simple mapping of the planar base domain to a sphere results in anisotropic sampling patterns.* ***Right:*** *Samples are moved closer to each other such that they cover equal angular distances with respect to the sphere's center.*

Both variants are evaluated in the fragment shader, just before the synthesis of the terrain takes place. After the vertices in the base domain have been generated, the fractal's height is computed for each position, and the deformed vertex is stored in an intermediate vertex texture for future rendering.

Once the vertex texture describing the height field has been computed, additional properties for the rendering process are derived in a second pass. This includes normals for lighting, and the slope (i.e. normal magnitude) and water depth used for the proto-texturing described in the next section. The water depth is defined with respect to an user-defined water-level. All properties are then stored in textures with floating point precision.

4.2 Proto-Texturing

Proto-texturing is a common method to generate geo-typical textures [2]. The idea is to use a set of textures that serve as prototypes for the simulation of real material, i.e. grass, sand, rock, snow etc. A height/slope-dependent weighting function is typically utilized to blend the textures together. In this way, many of the textural variations found in nature can be reproduced. On the other hand, if proto-textures are rendered and the user moves further away from the height field, artifacts emerge due to the use of periodically repeated textures.

To avoid this effect, Dachsbacher et al. [3] proposed to use color information only and to syn-

thesize the texture procedurally at every point during rendering. The intrinsically complex shader is amortized by caching parts of the results on the GPU. We suggest a different strategy, based on the observation that the periodic structures are essentially caused by the variance of the color values in the proto-textures. Consequently we build a custom mipmap such that the variance is continuously decreased with each level. This can be accomplished by first applying the smoothing filter as is usually done to compute mipmaps, and then take a weighted average between the smoothed image and the global mean of the colors. The user can control the process by selecting suitable filters and by providing a weighting parameter to affect the variance distribution across the levels. Exploiting log-step texture reduce operations [11] this step can be fully implemented on the GPU. The effect is demonstrated in Figure 7.

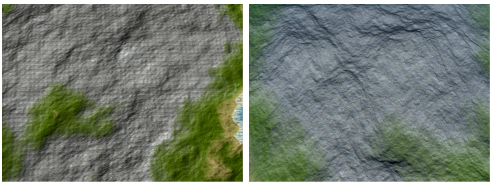

Figure 7: *The effect of reducing the variance of the texture through the mipmap levels is demonstrated.* ***Left:*** *Normal mipmap using a Lanczos filter for downsampling.* ***Right:*** *mipmap with reduced variances.*

In the colorplate (Figure 8 left), an example demonstrating the potential of proto-textures is shown. As can be seen, at steep rock formations (where $\|\nabla\mathcal{H}\|_2$) is large) the grass texture is suppressed due to a low weight, while the rock texture gets assigned a relatively high weight and hence is clearly visible.

4.3 Water

Water depths are computed per vertex according to an user-selected water level. As proposed by Schneider et al. [28], the water surface can then be rendered without major performance impact. While in the original paper the water surface was synthesized, we use a time-resolved normal map to obtain the appearance of moving waves. To achieve a different visual look for shallow and deep water, the strength of the bump effect, the reflectivity and the color of the surface are modulated with the water depth. This gives shallow water a more transparent, less reflective look, while deep oceans get the green-ish hue observed in nature. Since the geometry of the ground is considerably more complex than the simple pool scene in the original paper, determining the length of the transmitted ray can no longer be done using simple ray/hemicube intersection. Instead, we approximate the length by assuming a locally flat ground and computing a ray/plane intersection taking the interpolated per-pixel water depth into account. The resulting length can then be used to approximate caustics and extinction as proposed by Hall et al. [23]. The result is shown on the colorplate (Figure 10).

4.4 Results

We have used the proposed fractal synthesizer to generate a number of different scenes including auxiliary functions composed of several basis functions, proto-textures, and a texture-based water surface. All of our tests were run on a single processor Pentium 4 equipped with an nVidia GeForce 7800 GTX. The described system was implemented using OpenGL. In all of our tests the landscape was evaluated at the vertices of a 512×512 projected grid. The final rendering of this grid was done onto a 1280×1024 frame buffer. Up to ten octaves where added to procedurally evaluate the fractal landscape. The results are shown on the colorplate (Figures 8 to 10).

Besides the appealing quality of the synthesized landscapes, even at these high resolutions the synthesizer still runs at highly interactive rates. All scenes are synthesized and rendered at about 70 fps on our target architecture. Of this time, roughly 20% is spent for projecting the grid vertices and evaluating the fractal at the projected grid points. The remaining time is spent for level of detail computations and texturing, of which about 60% are consumed by the latter task.

5 Conclusions and Future Work

We have presented an interactive fractal landscape synthesizer on programmable graphics hardware, which exploits the intrinsic strengths of GPUs to generate *and* render high-quality, high-resolution, textured and shaded terrains. Since the user interacts with the same data that is used to form the

image, the parameter space can be intuitively managed. To our best knowledge this is the first time an interactive WYSIWYG interface for the synthesis of high-quality fractals has been proposed. Since the synthesis step is directly integrated into the rendering procedure, our method requires neither any polygonal representation nor a pre-processing stage. The suggested method is well suited for applications where the shape of the landscape is permanently modified by the user.

In the future we aim at enhancing the synthesizer about local displacement overlays to enable realistic simulation of global and local erosion features. Such overlays can be used to encode changes in height caused by natural processes, and they can easily be integrated into the rendering process to modify the synthesized height values.

We will further try to extend our work towards the simulation of entire virtual planets, including atmospheric effects as well as surface vegetation on the surface. Especially plants being created automatically based on the current terrain characteristics seem to be a challenging but also promising task.

To deal with the aforementioned drawbacks of the projected grid, we would like to investigate the suitability of the geometry clipmap approach [12].

References

[1] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. BDAM – batched dynamic adaptive meshes for high performance terrain visualization. In *Eurographics*, 2003.

[2] G. Corpes. Procedural landscapes. Presentation at Game Developer's Conference, 2001. //www.cix.co.uk/∼glennc/gdcetalk_files/frame.htm.

[3] C. Dachsbacher. *Shader X*, chapter Cached Procedural Textures for Terrain Rendering. Charles River Media, 2005.

[4] M. A. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. ROAMing terrain: real-time optimally adapting meshes. In *IEEE Vis '97*, pages 81–88, 1997.

[5] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing & Modelling - A Procedural Approach, 3rd Edition*. Morgan Kaufmann Publishers, 2003.

[6] R.L. Ferguson, R. Economy, W.A. Kelly, and P.P. Ramos. Continuous terrain level of detail for visual simulation. In *IMAGE V Conference*, 1990.

[7] S. Green. *GPU Gems 2*, chapter Implementing Improved Perlin Noise, pages 409–416. Addison-Wesley Professional, 2005.

[8] J.C. Hart. Perlin noise pixel shaders. In *Graphics Hardware Workshop*, 2001.

[9] D. Hinsinger, F. Neyret, and M.P. Cani. Interactive animation of ocean waves. In *ACM Symposium on Computer Animation*, pages 161–166, 2002.

[10] C. Johanson. Real-time water rendering - introducing the projected grid concept. Master's thesis, Lund University, 2004. http://graphics.cs.lth.se/theses/projects/projgrid.

[11] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. In *ACM SIGGRAPH*, 2003.

[12] F. Losasso and H. Hoppe. Geometry clipmaps: Terrain rendering using nested regular grids. In *ACM SIGGRAPH*, 2004.

[13] B. Mandelbrot. *The Fractal Geometry of Nature*. W.H. Freeman, 1982.

[14] G.S.P. Miller. The definition and rendering of terrain maps. In *ACM SIGGRAPH*, 1986.

[15] P. Molyneux. Populous. Bullfrog Productions, 1989.

[16] F. K. Musgrave, C.E. Kolb, and R.S. Mace. The synthesis and rendering of eroded fractal terrains. In *ACM SIGGRAPH*, pages 41–50, 1989.

[17] F.K. Musgrave. *Texturing & Modelling - A Procedural Approach, 3rd Edition*, chapter 17. Morgan Kaufmann Publishers, 2003.

[18] Pandromeda. MojoWorld. www.mojoworld.com.

[19] D.R. Peachey. Antialiasing solid textures. ACM SIGGRAPH, 'Functional Based Modeling' Course Notes, 1988.

[20] K. Perlin. An image synthesizer. In *ACM SIGGRAPH*, pages 287–296, 1985.

[21] K. Perlin and E.M. Hoffert. Hypertexture. In *ACM SIGGRAPH*, pages 253–262, 1989.

[22] K. Pulli. Ubiquitous 3D - graphics everywhere. Point-Based Graphics keynote presentation, 2005. http://research.nokia.com/people/kari_pulli/.

[23] D.P. Greenberg R.A. Hall. A testbed for realistic image synthesis. In *"IEEE" Computer Graphics and Applications*, pages 10–20, 1983.

[24] S. Röttger, W. Heidrich, P. Slussalek, and H.P. Seidel. Real-time generation of continuous levels of detail for height fields. In *WSCG*, pages 86–93, 1998.

[25] G. Sakas. Modeling and animating turbulent gaseous phenomena using spectral synthesis. *The Visual Computer*, 9(4):200–212, April 1993.

[26] D. Saupe. Point evaluation of multi-variable random fractals. In *Visualisierung in Mathematik und Naturwissenschaft, Bremer Computergraphik Tage*, 1988.

[27] D. Saupe. Simulation und Animation von Wolken mit Fraktalen. In *Informatik Fachberichte 222, Proc. GI Jahrestagung*, 1989.

[28] J. Schneider and R. Westermann. Towards real-time visual simulation of water surfaces. In *Vision, Modeling and Visualization*, 2001.

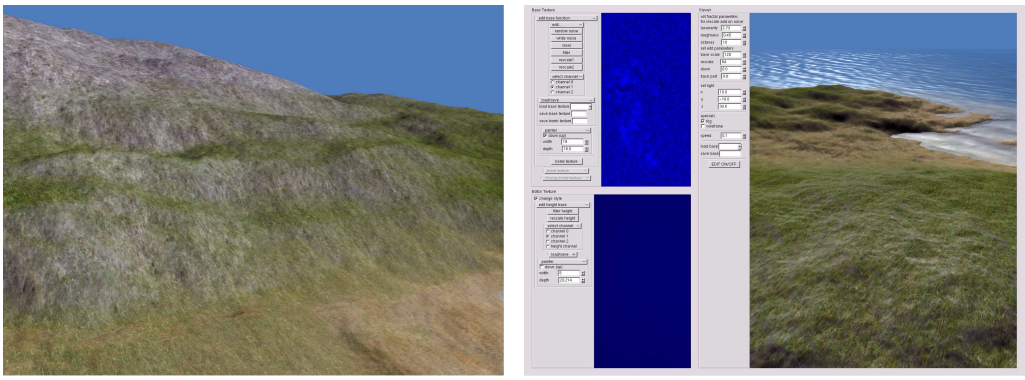

Figure 8: ***Left:****Combination of proto-textures using a height/slope-parameterized blending function. Note the absence of grass on steep rock formations.* ***Right:*** *The WYSIWYG user interface of the fractal landscape editor. A fine-scale input texture(top-left), a low-frequency sketch-pad (bottom-left) and the final exiting result using proto-texturing (right) is shown).*

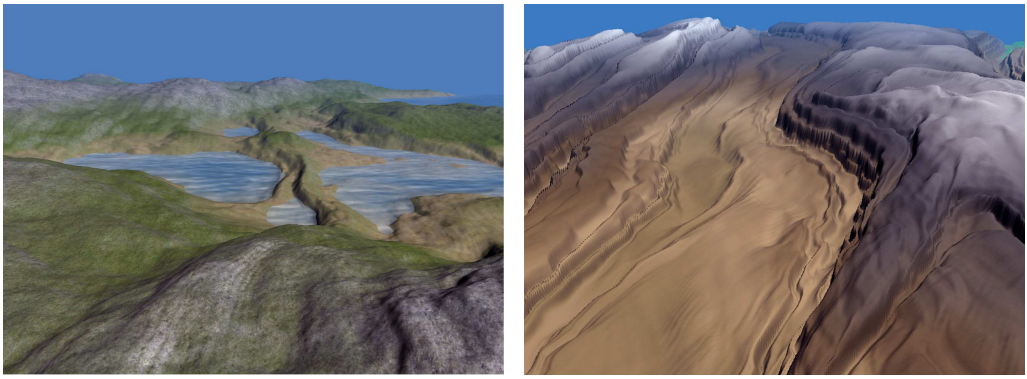

Figure 9: ***Left:*** *Rock formations using domain warping.* ***Right:*** *Sand dunes made possible by domain warping.*

Figure 10: ***Left:*** *Animated water surfaces are integrated into the fractal synthesizer without sacrificing performance.* ***Right:*** *Lake in a fractal landscape.*