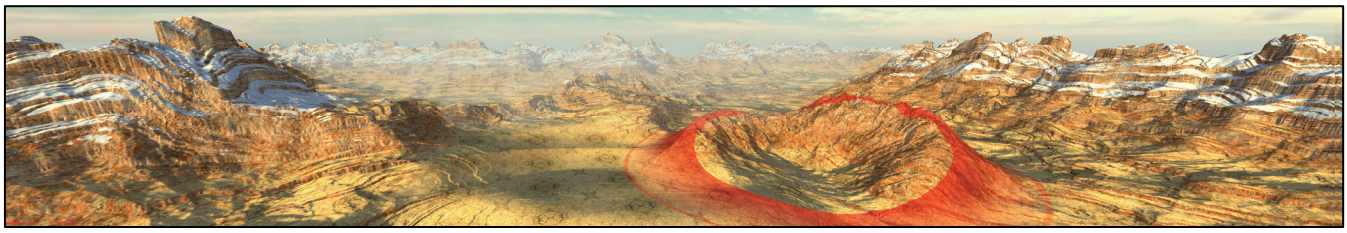
Interactive GPU-based procedural heightfield brushes

Giliam J.P. de Carpentier
W! Games
Damrak 20
1012 LH Amsterdam
The Netherlands
giliam@wgames.biz

Rafael Bidarra
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands
r.bidarra@ewi.tudelft.nl



ABSTRACT
Virtual outdoor terrain used for games is generally created by a level designer, using a variety of tools. These tools are currently based either on local interactive brush-based terrain sculpting or on global, parameterized algorithmic synthesis/adaptation of complete heightfields. Both tool types have largely complementary benefits and drawbacks. In this paper, we present *procedural brushes*, which combine the strengths of both tool types, offering a seamless transition from local control to fully automated generation, depending on the brush size. To optimize the execution speed of the computationally-intensive procedural algorithms, we propose to use the huge processing power of today's graphics hardware. For this, the procedural algorithms have been translated to shaders, and are used as part of a pipeline to render changes on a heightfield in video memory. We present a GPU brush editing pipeline for graphics hardware supporting Shader Model 3.0, coping with hardware restrictions regarding blend modes, precision and texture size. Several implemented procedural algorithms are described as well, two of which are novel. Experiments showed that the implemented system resulted in a speedup of roughly one order of magnitude over a reference CPU pipeline implementation. This made it possible for users to apply both trivial and complex procedural brushes at interactive rates, thus leading to a more efficient creation of complex virtual worlds.

Categories and Subject Descriptors
I.3.3 [**Computer Graphics**]: Hardware Architecture – *Graphics processors*; I.3.4 [**Computer Graphics**]: Graphics Utilities – *Paint systems*; I.3.6 [**Computer Graphics**]: Methodology and Techniques – *Interaction techniques*; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism – *Fractals*

Keywords
terrain modeling, procedural synthesis, heightfields, GPGPU, shader programming, graphics hardware

1. INTRODUCTION
User expectation and technological sophistication of computer games is increasing with every new generation of consoles. To create more detailed and realistic virtual worlds for the latest hardware, the content creation pipeline should get more powerful as well. This demands tools that improve efficiency and productivity of game level designers. With this motivation, we aimed at improving the tools available for heightfield creation by investigating novel combinations, variations and applications of related tools and techniques.

A heightfield, heightmap or (digital) elevation map is an ordered dataset that defines an elevation sample for each point on some planar regular grid, displacing these points perpendicularly to the plane. In the context of terrain, the grid is typically assumed horizontal and, consequently, the displacement is strictly vertical. Even though this representation disallows any overhangs and caves, it is still used in many games because of its compact representation and the availability of optimized rendering algorithms (e.g. [1], [2] and [4]). Heightfield data can both be scanned from real-world areas or created from scratch for fictional worlds. To create new heightfields, several tools are currently available. These tools can roughly be divided into two categories, which are described next.

Firstly, there are tools that allow the user to 'brush' over the terrain using input strokes of a mouse or tablet. These strokes are then used to interactively apply brush operations in the brushed areas. Because the operations are kept relatively crude and mathematically simple, they can be applied at interactive rates. Brushes offer level designers maximum control to create hills, valleys and plains but require much time and expertise to create detailed and realistic terrains with. Typical operations include terrain raising, lowering and leveling, and common brush parameters include brush radius and effect scale.

Secondly, there are tools that can output realistic heightfield using specialized algorithms. Two algorithmic classes can be distinguished here: simulation and procedural synthesis. Simulation algorithms globally transform terrain by directly imitating a geological process. These algorithms are typically only (over-)simplifications of the complex and sometimes poorly understood natural processes. Still, with some experimentation, impressive results can be achieved [11]. However, the vast amount of data and the considerable number of required iterations makes running these simulations very slow. Due to this fact, they will not be considered for this paper. In contrast to simulation algorithms, procedural synthesis algorithms do not try to simulate natural processes directly, but rather approximate the fractal-like semi-random patterns empirically found in the results of these processes (e.g. smooth hills or rough mountains). By a process of data amplification using a number of user-set parameters, some procedural multi-resolution synthesis scheme and a pseudo-random noise generator, large terrains can be synthesized with minimum effort from the user. Because procedural synthesis algorithms do not require multiple iterations to be evaluated, they are many times faster than simulations. Still, they are typically too slow to be executed for large terrains at interactive rates.

Procedural synthesis tools also have another disadvantage. As long as the user is satisfied with the limited level of control offered by these tools, their benefit over interactive brushes is obvious. However, when the user desires more precise control over the location of specific terrain features (e.g. mountains, plains and valleys), the parameters offered are too limited, as these are used globally (i.e. everywhere on the heightfield). This can render these tools of limited use. Furthermore, they most often only excel in the creation of one type of natural terrain. The creation of a wide range of terrain types could be achieved by either offering a large set of different complex algorithmic tools, or supporting a setup of applying a smaller set of simpler algorithmic tools as building blocks in a cascading node-based tool graph that can be designed by the user. One might even use such a graph to mask different areas of the output of different nodes and compose these masked outputs to create more spatially varying and controllable results. From the user's perspective, the latter would be more flexible and customizable to his specific needs, but would also require a more thorough understanding of these nodes, as well as proficiency with the more mathematical treatment of operations when setting up a useful tool graph.

In this paper, we propose the application of procedural synthesis tools as brushes, which we believe is both more natural and more intuitive to the user. As a result, the user is able to seamlessly select the right amount of local control and automation by simply varying the brush size, as small brushes provide more control, enabling the user to do most of the sculpting, while large brushes allow large features to be generated algorithmically. As brushing requires interactive feedback to be effective, any implementation of such a system must be fast enough to execute the required algorithms at interactive rates. Interactive feedback rates are most needed when maximum user control is desired and, thus, smaller brushes are used. Fortunately, these small brushes are inherently also fastest to evaluate. Still, complex algorithms require large amounts of computational power, especially when the brushes apply procedural algorithms. To this end, we propose utilizing the processing power of today's graphics cards to accelerate the evaluation of these algorithms.

Executing algorithms on the GPU that are not directly related to rendering is a relatively new approach, which is largely made possible by the increasing programmability of this hardware. The combined processing power of the large number of highly parallel processing units easily surpasses the processing power of the CPU, as long as an algorithm and its data structures can be mapped efficiently [13]. For several applications, including many physics simulations, there have been reported speedups of roughly one order of magnitude when implemented on the GPU [12].

Due to restrictions on the length of this paper, not all algorithmic details could be given here. For additional details, the reader is referred to the thesis on which this paper is largely based [3].

The remainder of the paper is organized as follows: Section 2 briefly surveys previous work from a number of different fields. Section 3 discusses different components needed in a basic brush system. Section 4 describes the GPU-based brush pipeline and Section 5 presents both common procedural terrain algorithms and two novel algorithms for the proposed GPU pipeline. Section 6 covers results obtained by experimental evaluation of our implementation of this system within a terrain editor. Finally, Section 7 concludes the paper.

2. PREVIOUS WORK

The concept of virtual brushes is well known from many image editing applications. There, the user is able to 'paint' with highly-customizable tools like pencils, erasers and clone stamps. Combined with functionality such as multiple undo, layers and masks, it forms a powerful, yet intuitive concept to create and edit images with. Several terrain editing applications have adopted the concept of brushes. However, these implementations generally only support the most basic operations. Furthermore, results are often found dependent on frame rate, which itself is dependent on brush size.

In addition to tools that enable users to brush terrain themselves, a different set of tools can synthesize a whole terrain algorithmically. Mandelbrot was the first to observe the similarity between a trace of the one dimensional fractional Brownian motion over time and the contours of mountain peaks [9]. This idea was later generalized to fractional Brownian motion fractal surfaces with a $f^{-\beta}$ power spectrum. Many fractal synthesis algorithms have been devised over the years that directly or indirectly approximate this power spectrum. Examples are midpoint displacement [6], Poisson faulting [8] and Fourier synthesis [16], all with different advantages and disadvantages. This paper focuses on noise synthesis [10], as it is flexible, it is known to produce few artifacts and is suited to be run on parallel hardware, including graphics hardware.

Noise synthesis approximates the desired power spectrum by calculating the weighted sum of several different band-limited noise functions. An often used function that produces well-behaved band-limited noise is the Perlin noise function [14]. As Perlin noise is a well-known and generally useful feature in procedural imagery, different implementations of the function as a GPU pixel shader or a vertex shader have already been developed for different shader versions. Our work is based on the pixel shader by Green [7], but is optimized to efficiently calculate a noise scalar as a function of a continuous 2D position and a discrete seed number. A number of extensions to Perlin-based

noise summing are found in literature and in many applications, all transforming inputs and outputs differently to get different shapes, including ridged and billowy noise. A more complex variation is described by Quilez [15]. All these variations can be efficiently implemented on the GPU as procedural brushes, offering the user a wide set of terrain types to choose from.

3. BRUSH SYSTEM

This section describes a basic brush system, which will be extended and mapped to the GPU in subsequent sections. The brush system lets the user apply a brush tool along an arbitrary brush stroke. It is assumed that a brush tool can be positioned and activated on a heightfield using an input device like a mouse or tablet, and is circular in shape. The effect of the brush on the terrain must be calculated and visualized at interactive rates. Assuming the user can view the terrain in 3D from an arbitrary angle, the mouse or tablet pointer must be projected onto the 3D terrain. These 3D positions are then transformed into the local 3D heightfield space. As the brush is circular, it must also have a radius parameter. Although it might be possible to analytically calculate the effect of a brush along a complete or piece-wise brush stroke, we chose the more flexible approach of sequentially applying individual brush instances along the stroke. Together, the individually applied instances approximate a would-be continuous application of the brush. The distance between the center positions of the instances should always be (much) smaller than the brush radius; see Figure 1. The optimal distance is a tradeoff between performance and quality, both depending on the actual brush instance implementation, and possibly the user's preference. As a specific brush type might have additional parameters influencing performance or shape, the combination of a brush-specific heuristic and a user-controlled scale is best used. To get the positions for the individual instances, we use the local input positions of the stroke to create a spline, which is then sampled at the desired spatial intervals. This prevents the result from being frame-rate dependent.

The individual brush instances will apply the brush algorithm, which can take a noticeable amount of time for complex procedural algorithms and/or a large brush radius. These instances are best applied in parallel with other tasks like rendering, or at least be executed asynchronously. For this, an instance FIFO work queue can be used, adding new instances when they become available, and processing as much instances as possible per frame without affecting the other tasks too much, thus keeping the frame rate at a workable level at all times. Note that using a queue can result in the brush effect lagging behind the user's input. However, this is still preferable to low and irregular frame rates.

We can assume the whole heightfield is represented as one 2D data array containing the individual elevation levels. Each element in this array represents a local height value $H(u,v)$, normalized between 0 and 1. As 8-bit heightfield elements typically would results in banding artifacts, either 16-bit integers or 32-bit floating point numbers are used in practice.

When a brush instance i is applied, the circular area in H at center position $\boldsymbol{c}_i$ and radius r_i. is affected. Individual brush types might use additional parameters defining the shape and scale of the brush effect inside this circular area. These parameters are used as input to the brush algorithm, which will update the rectangular area $[\boldsymbol{c}_{i,x} - r_i, \boldsymbol{c}_{i,y} - r_i] - [\boldsymbol{c}_{i,x} + r_i, \boldsymbol{c}_{i,y} + r_i]$ of H. For example, an

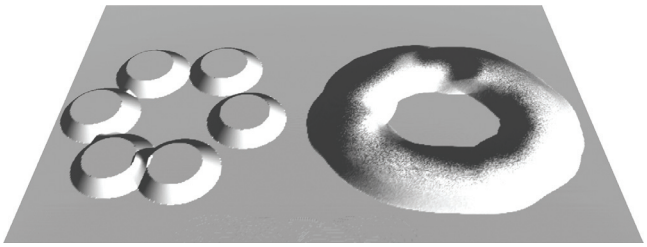


Figure 1. Approximating a circular brush stroke with a low (left) and high (right) brush instance density.

algorithm used for a simple 'terrain-raising' brush will read the local old height value, add a small value to this, and output it. Consequently, the output height value h_t for heightfield element t at UV position $\boldsymbol{p}_t$ can be defined as

$$h_t = H(\boldsymbol{p}_t) + s \cdot o \cdot f_i \cdot m \cdot b, \tag{1}$$

where s is some constant based on the heightfield's internal data format and dynamic height range, o is the user-controllable strength of the brush, f_i is the pressure of user's stylus tip at instance i, and m is a brush-specific multiplier. For the terrain-raising brush, a fixed $m = 1$ is used. We define b as

$$b = 1 - \min((\boldsymbol{p}_t - \boldsymbol{c}_i) \bullet (\boldsymbol{p}_t - \boldsymbol{c}_i) / r_i^2,\ 1). \tag{2}$$

This calculates a scale factor based on the squared distance between position $\boldsymbol{p}_t$ and instance center $\boldsymbol{c}_i$. This squared distance will cause the brush instance to exhibit a parabolic fall-off effect, adding $s \cdot o \cdot f_i \cdot m$ to $H(\boldsymbol{p}_t)$ in Eq. (1) for an element that lies precisely at $\boldsymbol{c}_i$ and simply returning $H(\boldsymbol{p}_t)$ for all elements t that lie outside the instance's circular area (i.e. for $\|\boldsymbol{p}_t - \boldsymbol{c}_i\| \geq r_i$). If desired, Eq. (2) can easily be extended to incorporate additional parameters like an inner radius and a shape of a fall-off ramp towards the (outer) radius r_i.

After any instances have been applied successfully to the heightfield, the renderer must be notified to use the updated data. Depending on the used rendering algorithm, the rendered geometry might need to be updated as well, as the algorithm might not render the heightfield directly, but instead uses geometry buffers derived from the heightfield. Obviously, rendering algorithms that require little or no heightfield preprocessing are better suited for terrain editing. The heightfield rendering algorithm we chose for this paper is called GeoMipMapping [2]. This level-of-detail, tile-based rendering algorithm can easily be extended and optimized to share and reuse vertex and index buffers. Also, all terrain lighting and texturing was done in real-time, thus avoiding expensive preprocessing.

4. GPU PIPELINE

To accommodate GPU-accelerated brush calculations, the effect of a brush instance must be represented as some rendering operation. Furthermore, the heightfield must be present in video memory, represented as one or more textures. For the moment, we will assume the whole heightfield is stored as one single-channel texture, representing $H(u,v)$. Note that when this texture is interpreted as a greyscale image, the minimum and maximum height values are represented by black and white, respectively.

The bit depth of the heightfield texture must be at least as large as the bit depth of the original data. This means that it might be necessary to use a 16-bit integer or 32-bit float texture format. Note that 16-bit float texture formats would typically be too coarse and would result in visual banding artifacts. Alternatively, a bit packing technique can be used, representing 16-bit integer values as multiple components of a lower bit depth.

A simple and efficient way of implementing a terrain-raising brush based on Eq. (1) would be to use the latest H texture as a render target and render a (small) quad on top of it for brush instance i using the additive blend mode. The center of this quad needs to be at $\boldsymbol{c}_i$, and it should be just large enough to apply a texture of a non-black circle of radius r_i on a black background. This rendering operation would whiten, and thus raise, the area of H covered by the circular brush instance i. However, this method is not very flexible, as differences in brush types would be limited to applying different textures and using the supported render blend modes. Furthermore, (additive) blending of 16-bit or 32-bit textures might not even be supported on all hardware. Therefore, a more flexible, better supported alternative scheme was chosen instead, even though it is more complex to set up and optimize for performance.

To accommodate read-backs and custom processing of values in H in order to calculate the output of more complex brush types, a brush instance algorithm must be implemented as a pixel shader which will have access to H. This pixel shader will then be used to render an updated version of H. In contrast to the previous scheme, the output 'pixel color' does not have to be blended in additively, but will simply replace the old values in H. But as a texture cannot be read from and written to from within the same render call, we chose to use a 'ping pong' rendering scheme. This requires the use of a second (render target) texture of the same dimensions as the original H. For the application of the first (and possibly only) brush instance in the queue, this second texture is used to render to. When the render call has been completed, the render target texture will contain all values of the updated H. Assuming the original H texture can be used as a render target as well, the roles of the two textures can now be swapped, allowing the next brush instance to be applied by rendering to the original texture again, and so forth.

Each render call, applying the effect of a single brush instance, should at least bind the brush algorithm similar to Eq. (1) in the form of a pixel shader, the 'input' heightfield texture, the 'output' heightfield render target texture, and the brush instance position $\boldsymbol{c}_i$ and radius r_i. Of course, any brush-specific parameters might be set as well as one or more shader constants or supplementary input textures. The render call will render a quad using this pixel shader that will cover (at least) the area affected by the brush instance. When the output texture is not (known to be) identical to the input texture in areas other than the affected area because it might just have been allocated, or it does not contain the latest updates, the values from the input texture (the previous version of H) must be copied to the output texture (the updated version of H) as well. This is easily accomplished using a trivial 'copy' pixel shader that is applied to quads covering these outdated areas but not covering the area affected by the brush instance.

The ping pong rendering scheme requires many memory read/write operations. This can easily cause the memory bandwidth to become the main performance bottleneck. This overhead can be minimized by combining the calculations for multiple brush instances inside the pixel shader, combining separate results in order, and outputting the combined result as one pixel value. One good overall technique we found (for the used hardware and implemented brush types) was to always feed the pixel shader the oldest 16 instances. If the queue size was less than 16, some of these 16 slots were disabled (i.e. $o = 0$ in Eq. (1)). Other techniques like in-shader dynamic branching, and automatic compilation and static selection of a shader with the best capacity have been tried as well, but these did not perform significantly better, and did not justify the required increased complexity. Note that instance batching can only be implemented efficiently for algorithms that calculate a new height h_t based solely on $\boldsymbol{p}_t$, (e.g. most procedural techniques) but no other elements in H (e.g. a multi-tap filter). Otherwise, the interdependencies between inputs and outputs among these elements require either sharing or recalculation of the results for the intermediate instances in a batch.

This assumption also makes it possible to easily partition the heightfield H into smaller textures with minimal changes to the editing pipeline and pixel shaders. Partitioning a heightfield into a number of smaller fixed-sized textures has several advantages, especially for relatively large heightfields. For one, the maximally supported heightfield size is no longer dependent on the maximum texture size supported by graphics hardware. Also, it is no longer required to keep the complete heightfield in video memory, but only the smaller textures that are immediately required for editing, reducing the minimum footprint in video memory. Furthermore, smaller textures are generally faster to read from when rendering, thus improving performance. And lastly, when multiple versions need to be kept in memory to make it possible to undo the latest operations, all textures that have not been updated by these operations can be shared between versions, potentially saving a lot of (main) memory without implementing more complex and expensive compression techniques. Note that when a brush instance affects multiple textures, it must update each of these textures as a separate render operation, which is easily implemented by adding these updates as separate tasks in the instance work queue, each one affecting only one texture. This also has the advantage of reducing the granularity of these queued tasks, allowing for better load balancing between any per-frame editing operations and other GPU tasks.

5. GPU BRUSH VARIATIONS

The terrain-raising brush discussed in Section 3 assumes m in Eq. (1) to be 1. Alternatively, when $m = -1$ is used, the effect is reversed, resulting in a 'terrain-lowering' brush. By replacing m with some function of the available parameters, many different brush types can be represented. For example, when a brush is created with

$$m = B(T_i \cdot [\boldsymbol{p}_{t,x}, \boldsymbol{p}_{t,y}, 1]^T), \tag{3}$$

with B being some user-controllable texture and T_i being a 2 x 3 rotate/scale/translate transformation matrix, a transformed version of B will be added to H within the area the user applies the brush to. B could be a complete heightfield or some interesting tiling pattern. T_i can be defined per instance i if desired, which could, for example, be used to minimize repetition of any obvious patterns in B.

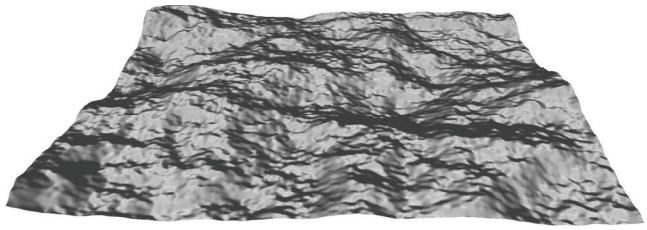


Figure 2. Basic noise summing

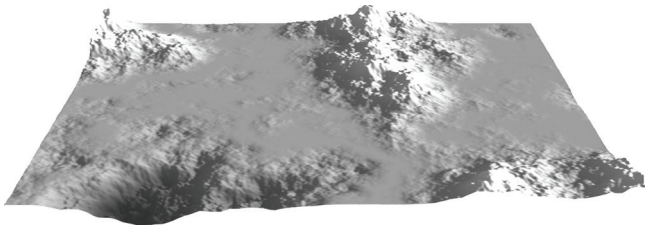


Figure 3. Range warping

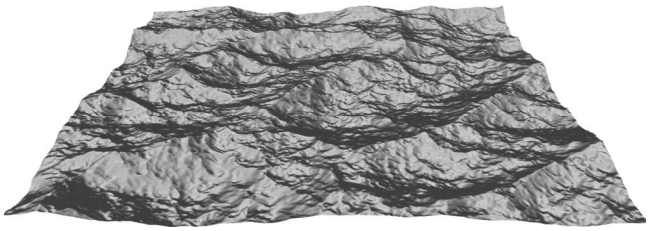


Figure 4. Ridged noise

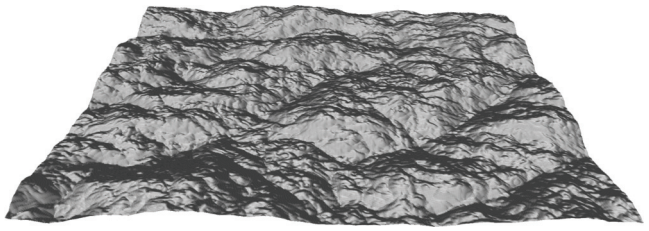


Figure 5. Billowy noise

Instead of B being defined by a 2D texture stored in video memory, it can also be defined implicitly by an algorithmic function of 2D space. As B might be sampled at any position in its domain, recursive and iterative algorithms are less suited for this purpose. A flexible and fast candidate is noise summing, based on the Perlin noise function. Perlin noise divides the domain into an integer lattice. The integer points are procedurally assigned a gradient by hashing the integer position using simple permutation and gradient look-up tables. The noise result of any input position is evaluated by calculating and interpolating the gradients of the nearest integer lattice points. For our purposes, the noise is assumed to be evaluated at some 2D position with a given seed number for variation, requiring a 3D Perlin noise function. But because we chose to support only integer seed numbers, we were able to simplify the original noise algorithm, requiring only the evaluation and blending of four points on the 3D integer gradient lattice, instead of eight. After all required hashing and gradient lookup tables were combined into two 2D 8-bit RGBA textures, the final Cg pixel shader noise function needed only three texture lookups and a minimum of logic. See [3] for more details.

5.1 Basic Noise Summing Brush

Noise summing requires the calculation of different noise bands. Each of these bands, or octaves, can be written as a Perlin noise function call with a differently scaled input position and output weight. The final weighted summing of the noise outputs effectively composes the separate pieces of the power spectrum together. The basic noise summing algorithm can be defined as follows:

$$B(\boldsymbol{p}) = \sum_{j=0}^{n-1} w^j \cdot N(\lambda^j \cdot \boldsymbol{p},\, e_j)\,. \tag{4}$$

Here, $N(\boldsymbol{p}, e)$ is the Perlin noise function, consistently returning the same scalar value for the same input consisting of 2D input position $\boldsymbol{p}$ and seed number e. Its output is assumed to lie in the range [-1, 1]. λ is called the lacunarity and represents the ratio between the mean frequency of subsequent noise bands j and j+1. For the best results, it is typically kept near, but not exactly at 2.0. w must lie in the range (0, 1) and is used to control the terrain roughness. See [10] for more details. The number of noise bands n directly affects the amount of detail levels and the time required to calculate $B(\boldsymbol{p})$, and is typically kept between 5 and 10, depending on the heightfield size and the desired quality. Note that T_i in Eq. (3) can be used to directly influence the scale of all created terrain features, and is typically best kept unchanged while applying a noise summing brush.

5.2 Basic Warping

The basic noise summing algorithm defined in Eq. (4) will result in basic procedural terrain shapes; see Figure 2. Even though changing the parameters will result in either rougher of smoother terrain, it cannot be used to generate many different terrain types with. However, it can easily be adapted to more complex forms by transforming different parts of Eq. (4) by (non-linear) functions:

$$B(\boldsymbol{p}) = T_{post}\Big(\sum_{j=0}^{n-1} w^j \cdot T_{in}(N(T_{pre}(\lambda^j \cdot \boldsymbol{p}),\, e_j))\Big)\,. \tag{5}$$

Any parameters for the T_{pre}, T_{in} and T_{post} transformation functions can be made available to the user of the system, allowing the user to experiment with these parameters. Many different combinations of transforms can be used. For example, when $T_{post}(h) = h^a$ with $a > 1$, peaks and valleys are made steeper, while areas in between are flattened; see Figure 3. The well-known 'ridged' noise (Figure 4) and 'billowy' noise (Figure 5) can be created with

$$T_{in}(h) = 1 - \mathrm{abs}(h), \text{ and} \tag{6}$$

$$T_{in}(h) = \mathrm{abs}(h), \text{ respectively.} \tag{7}$$

The T_{pre} function can be used to warp the input of N. For example,

$$T_{pre}(\boldsymbol{p}) = \boldsymbol{p} + [\ \alpha N(\beta\boldsymbol{p}, e_1),\, \alpha N(\beta\boldsymbol{p}, e_2)\ ]^T \tag{8}$$

will result in more swirly shapes, locally compressing, rotating and elongating features. Here, α and β are non-zero scale factors, and e_1 and e_2 are seed numbers that are independent of e_j. For even more complexly swirling landscapes, the noise function N in

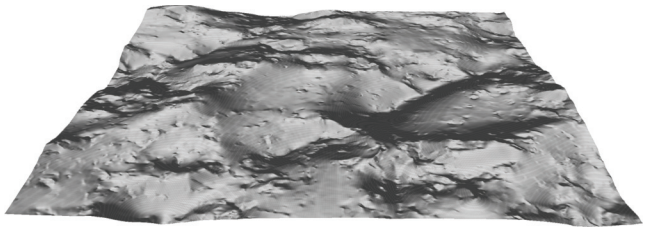


Figure 6. Quilez noise

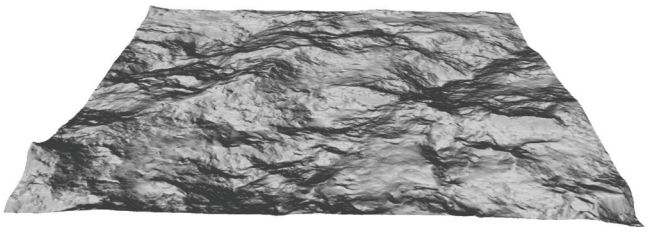


Figure 7. Distorted Quilez noise

Eq. (8) can be replaced by B from Eq. (5). Compare Figure 6 and Figure 7.

The techniques discussed above can create very detailed landscapes, but typically lack spatial variation of the terrain properties found in nature. For example, both valleys and peaks are generated equally rough. Of course, when these are applied as brushes, the user can both control the affected area and change brush settings at will, resulting in more natural landscapes when used well. Nonetheless, offering brushes that create more varied terrain algorithmically can be beneficial to the user. (Hybrid) multi-fractal [5] algorithms are examples of procedural techniques that offer a height-dependent roughness. However, results seem somewhat mathematical or 'synthesized'. A noise variation described in an article by Quilez [15] generates a richly varied landscape by defining a T_{pre} function for each band that scales the noise based on the spatial derivatives of N for all coarser bands; see Figure 6. Although the article claims to use a gradient lattice Perlin noise function and its derivative, it uses a simpler value lattice noise generator instead. Nonetheless, the algorithm generates complexly varying amounts of local detail, resulting in more natural landscapes. As the algorithm can be written as a variation of Eq. (5), it can simply be implemented as pixel shader code as well.

5.3 Directional Noise

Another application of T_i from Eq. (3) is to use it as a per-instance transformation matrix to compress the input at an angle to the local direction of the brush stroke. This can be combined with all the above techniques. The additional influence gives the user a range of new possibilities. For example, when such a brush is set up to compress the input perpendicularly to the brush stroke which is then used to brush along a mountain ridge, features will appear that resemble small gullies, carved out by down-hill streams. Note that compression of the input to N results in larger features. Compression in other directions relative to the local stroke direction can be achieved in a similar fashion but can be used in different ways. For example, when the compression is in the direction of the local stroke direction and is applied along a mountain ridge, the mountain face will get terrace-like features. When it is applied as a wavy brush stroke from top to bottom on a smooth area of a mountain, naturally flowing gullies will appear. See Figure 8 for an example of the different effects.

5.4 Erosive Noise

Instead of letting the user apply the above brush near existing mountain ridges, the above idea could also be applied when brushing new mountain ranges. So instead of adding features compressed along the brush stroke on top of an existing mountain range, new mountains are distorted with these features while being created. Note that defining such a function is probably as much an art as it is a science. Hence, the algorithm described next is only one of many possible approaches. As the 2D derivatives of $B(\boldsymbol{p})$ roughly point towards the ridge nearest to $\boldsymbol{p}$, they indicate the direction of the up/down-hill feature elongation. In order to compress the coordinates on a slope, $\boldsymbol{p}$ is 'pushed' towards the ridge top, in the direction of the gradient. In effect, the features on a slope will become elongated, while the features near the top will become compressed; see Figure 9. To get ridged mountains, the algorithm is based on ridged noise (i.e. Eq. (6)), now called $R(h)$. Note that $R(h)$ is a function of $N(x, y)$, assuming Eq. (5) is still used. The gradient of $R(h)$ has an inconvenient discontinuity in its spatial derivatives at $h = N(x, y) = 0$ due to the abs() in its definition. To overcome this, the gradient $G(x, y)$ of $R(N(x, y))$ is approximated by

$$\left[-N(x,y)\frac{\delta N(x,y)}{\delta x}\ ,\ \ -N(x,y)\frac{\delta N(x,y)}{\delta y}\right]^T . \qquad (9)$$

This approximation effectively scales the derivatives to 0 at the discontinuity. The derivatives of the Perlin noise function $N(x, y)$ are calculated analytically. As $B(\boldsymbol{p})$ from Eq. (5) is calculated band by band, so is the gradient that is used to displace the $\boldsymbol{p}$ for the next band of $B(\boldsymbol{p})$. The exact algorithm is described in pseudo code:

```
function calcErosiveNoiseAt(p_x, p_y)
 {
    freq = 1; amp = 1; B = 0;
    d_x = 0; d_y = 0; s = 1;

    for (j = 0; j < n; j++)
    {
        T_pre,x = freq*(p_x + d_x);
        T_pre,y = freq*(p_y + d_y);
        T_in = s*(1 - abs(N(T_pre,x , T_pre,y)));
        B = B + amp*T_in;

        d_x = d_x + amp*(α*s*G_x(T_pre,x , T_pre,y));
        d_y = d_y + amp*(α*s*G_y(T_pre,x , T_pre,y));
        s = s*min(1, max(0, β*B));

        amp = amp*w;
        freq = freq*λ;
    }
    return B;
}
```

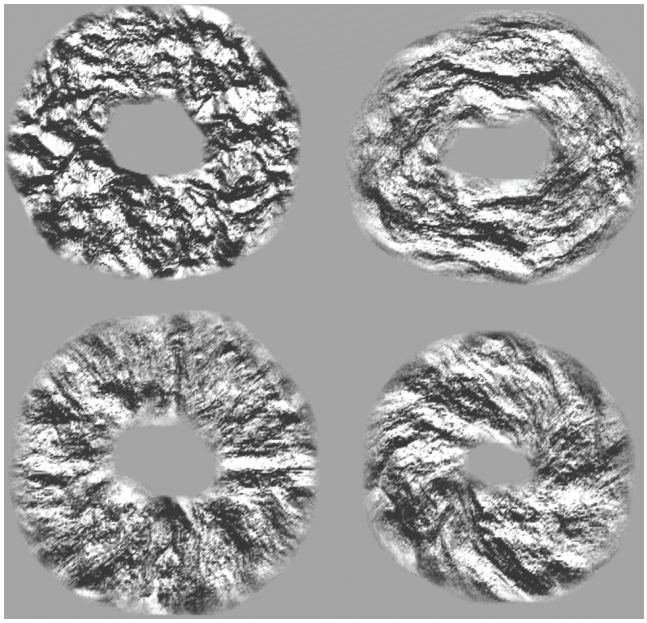


Figure 8. Directional ridged noise. Top left: Standard ridged noise features created by a circular brush stroke. Top right: Features compressed perpendicular to stroke. Bottom left: Features compressed in the direction of the stroke. Bottom right: Features compressed at an angle of 45°.

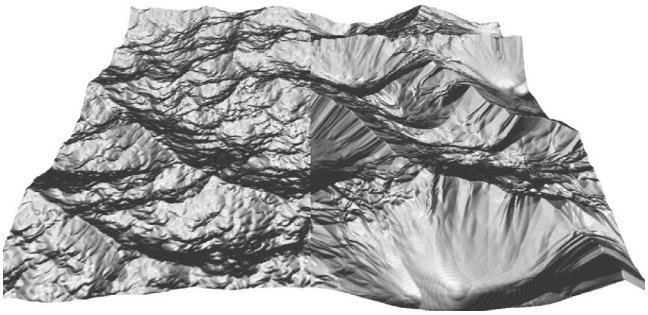


Figure 9. Ridged noise (left half) and erosive noise (right half).

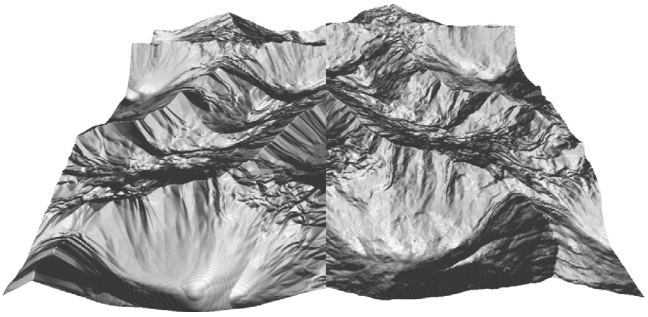


Figure 10. Erosive noise (left half), erosive noise with input distortion (right half).

The first four and last two lines in the `for` loop implement Eq. (5) for some T_{pre} and T_{in}. Logically, the gradient $G(x, y)$ is sampled at the same position as N. (d_x, d_y) accumulates a scaled G in a similar fashion to noise summing (see Eq. (4)) and displaces (p_x, p_y) to elongate features on a slope. s is an additional scale factor that scales down the amplitude of both $N(x, y)$ and $G(x, y)$ for finer bands at lower (intermediate) altitudes, causing the landscape to be relatively smooth near valleys. Because the effect of carved out gullies and smooth valleys approximates the effect of fluvial erosion, we named this algorithm *erosive noise*. The constants α and β control the amount of feature displacement and the amount of roughness near valleys, respectively. For Figure 9 and Figure 10, we used $\alpha = 0.15$ and $\beta = 1.1$. The above algorithm will create fairly straight gullies. However, the output, including the gullies, can easily be made more swirly by first warping the input (p_x, p_y) using Eq. (8). See Figure 10 for a comparison.

6. RESULTS

The proposed combination of procedural algorithms and brushes used on heightfields is both useful and powerful because the operations are executed fast enough to produce interactive brush feedback. Experiments showed that these rates were hard to achieve using CPU brush implementations. As the theoretical raw computing power of the GPU surpasses that of the CPU, the performance of a brush algorithm will increase when it is executed on the GPU instead of on the CPU, provided that it can be mapped efficiently to supported render concepts. The developed GPU pipeline easily outperformed reference CPU implementations, even though it introduced an additional I/O overhead. For example, a reference CPU implementation for the relatively simple terrain-raising brush from Section 3 executed at 10 fps when using a brush diameter of 500 heightfield elements, while a GPU implementation of this brush executed at 60 fps under identical circumstances. This was tested on an Intel T7200 / NVIDIA GeForce Go 7950 GTX machine. The overhead in the GPU pipeline is largely independent of the used brush algorithm. Consequently, more complex and computationally intensive brushes use the GPU computing power relatively more efficiently, thus further increasing the relative speedup. Furthermore, a larger brush radius will increase efficiency as well. Therefore, the user will benefit the most from the GPU pipeline for the more complex and larger, and thus slower, brushes. Overall, the performance of the GPU brush implementations reached speedups of up to one order of magnitude for large, complex procedural brushes.

Our fast and flexible procedural brush pipeline supports both fine editing control and effortless creation of complex procedural areas, by simply manipulating the brush radius and other intuitive settings. The terrain editor offering this toolset, including the

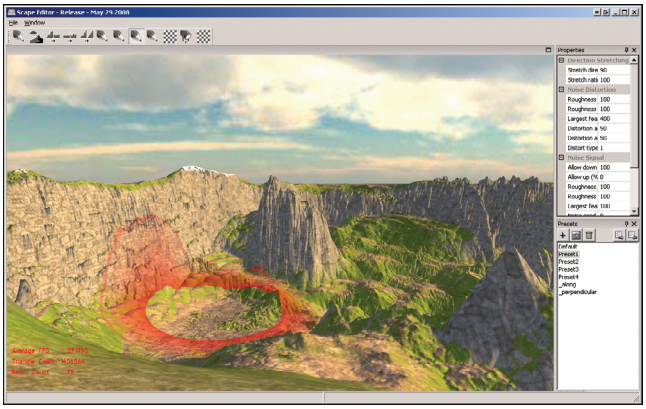


Figure 11. An example of a heightfield being created in our terrain editor.

novel directional and erosive noise algorithm brush (see Figure 11), has been implemented and used at Dutch game developer W!Games. It enables level designers to choose and mix different terrain types and operations, supporting efficient creation of effects that would be difficult to achieve by other means. However, the discussed GPU brush types are only a few possibilities of what can be achieved with the discussed pipeline, as many more (variations on) procedural CPU algorithms found in literature and applications could be mapped to GPU brushes using similar techniques, further expanding the toolset offered to the user.

7. CONCLUSIONS

Current demands in the computer games industry include creating increasingly detailed and realistic virtual worlds. This in turn can only be achieved by providing game level designers with a new generation of tools that efficiently boost the productivity of their creative design tasks, among which terrain creation plays a central role. Currently available terrain synthesis and editing applications fall short in providing either precise control, realistically complex output or interactivity. To solve these limitations we have introduced procedural brushes, which offer a seamless transition from local control to fully automated terrain generation. Our approach provides a flexible level of control ranging from that by low-level simplistic but precise tools, up to that by synthesis techniques.

Furthermore, two new algorithms have been described that are complementary to common synthesis algorithms and allow the user to brush more complex and realistic, typical terrain features. Firstly, directional noise has been introduced, providing a rather user-controlled variation that is specifically designed to be used as a brush. This variation yields features that are more dependent on the actual user brush strokes. Secondly, an erosive noise algorithm has been introduced, that excels in interactively creating eroded mountainous terrain with statistically different features in the created valleys, tops and slopes.

The proposed pipeline makes use of graphics cards by splitting brush strokes into separate instances and applying these instances as hardware-accelerated render operations, using textures to store the heightfield and pixel shaders to evaluate procedural brush algorithms. To implement this efficiently under the hardware restrictions of Shader Model 3.0, the heightfield is stored in separate page textures and instances are combined into one render call where possible. Even though this setup is more complex than a CPU implementation would need to be, the GPU pipeline accomplished a speedup of up to one order of magnitude for the more complex procedural brushes.

In short, the results described represent a considerable step towards simultaneously improving quality, speed and control of the tools offered to game level designers. Discussed techniques for this include currently available tools, ideas from other disciplines and novel algorithms. Experiments showed that even complex algorithms can be offered as interactive tools on today's hardware when parallelism is exploited. Therefore, achieving the ultimate goal of integrating these techniques within one single application can justly be expected to bring about significant improvements of the iterative workflow, a powerful enhancement of user control, and a considerable simplification in the creation of realistic terrain features.

ACKNOWLEDGEMENTS

We thank our colleagues Mike van der Voort and Jorik Blaas for their sharp comments on a preliminary version of the manuscript.

REFERENCES

[1] Asirvatham, A. and Hoppe, H. 2005. Terrain Rendering Using GPU-Based Geometry Clipmaps. In GPU Gems 2. R. Fernando, Ed., Addison Wesley, pp. 27-45.

[2] de Boer, W. 2000. Fast Terrain Rendering Using Geometrical MipMapping. Online article. http://www.flipcode.com/archives/article_geomipmaps.pdf.

[3] de Carpentier, G.J.P. 2008. Effective GPU-based synthesis and editing of realistic heightfields. M.Sc. Thesis. Delft University of Technology, The Netherlands.

[4] Duchaineau, M., Wolinsky, M., Sigeti, D.E., Miller, M.C., Aldrich, C. and Mineev-Weinstein, M.B. 1997. ROAMing Terrain: Real-time Optimally Adapting Meshes. In Proceedings of the IEEE Visualization '97. Los Alamitos, CA, USA. IEEE Computer Society Press, pp. 81-88.

[5] Ebert, D.S., Musgrave, F.K., Peachey, D., Perlin, K. and Worley, S. 2003. Texturing & Modeling: A Procedural Approach. Third Edition. The Morgan Kaufmann Series in Computer Graphics, pp. 498-506

[6] Fournier, A., Fussell, D. and Carpenter, L. Jun 1982. Computer Rendering of Stochastic Models. In Communications of the ACM, vol. 25, no. 6, pp. 371-384.

[7] Green, S. 2005. Implementing Improved Perlin Noise. In GPU Gems 2. R. Fernando, Ed., Addison Wesley, pp. 409-416.

[8] Krten, R. Jul. 2001. Generating Realistic Terrain. In Dr. Dobb's Journal: Software Tools for the Professional Programmer.

[9] Mandelbrot, B.B. 1982. The Fractal Geometry of Nature, New York, W.H. Freeman and Co.

[10] Musgrave, F.K. 1993. Methods for Realistic Landscape Imaging. Doctoral Thesis. Yale University.

[11] Musgrave, F.K., Kolb, C.E. and Mace, R.S. 1989. The synthesis and rendering of eroded fractal terrains. In Proceedings of the SIGGRAPH '89. ACM Press, New York, NY, pp. 41-50.

[12] Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E. and Phillips, J.C. May 2008. GPU Computing. Proceedings of the IEEE. Volume 96, Issue 5. pp. 879-899.

[13] Pharr, M. et al. 2005. Part IV: General-Purpose Computation on GPUs: A Primer. In GPU Gems 2. R. Fernando, Ed., Addison Wesley, pp. 451-589.

[14] Perlin, K. and Hoffert, E.M. 1989. Hypertexture. In Proceedings of the SIGGRAPH '89. ACM Press, New York, NY, pp. 253-262.

[15] Quilez, I. 2008. More noise. Online article. http://rgba.scenesp.org/iq/computer/articles/morenoise/morenoise.htm.

[16] Voss, R.F. 1985. Random Fractal Forgeries. In Fundamental Algorithms for Computer Graphics, R.A. Earnshaw, Ed., Springer-Verlag, Berlin.