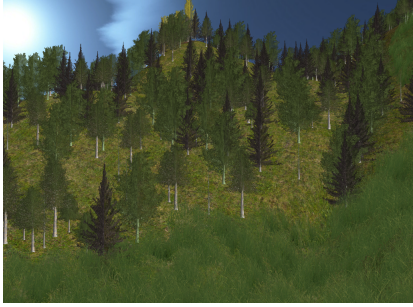


Procedural Terrain

Final project for class CS6491 Computer Graphics

Sebastian Weiss
November 30, 2015

I. Objective

Terrain is a topic of endless discussion in computer graphics. It has to cover a large area of the world and at the same time it has to provide enough details for close-up shots. It is extremely time consuming to create a large terrain that is also interesting when viewed closely.

In this project we present a framework for generating terrain, from a coarse grained height map to vegetation generation. The focus lies on combining existing techniques for performing the different tasks. The result should be an island because it provides a natural border of the world.

Applications of procedural terrain include computer games, movies and geographic visualization and simulation. The project was heavily inspired by a talk about "The Good Dinosaur" from Pixar Animation Studios.

II. Related Work

A lot of work has been done already in the field of generating height maps. There are in general three different approaches to this problem. Generating the height map completely from scratch using perlin noise, fractals, voronoi regions, software agents or genetic algorithms are described in [1], [2] and [3]. These models often lack realism because of the missing physically and geographic background. This is addressed in a second approach, hydraulic erosion models, as described in [4] and [5]. They start with existing height maps and increase the realism by simulating fluid to erode the terrain and forming rivers in the end. The other way is used in [6], here we start with a river network and build the terrain with respect to the river flow. The last approach copes with the lack of control in the previous described methods. They define the terrain by control features provided by the user, see [7] and [8]. On a 2D-scope, [9] should be mentioned because it describes a complete new approach not using height map grids, but voronoi regions as base primitives.

After the terrain is created, it must be populated with vegetation. Rendering of grass is described in e.g. [10] and [11]. Trees must be generated first and for that, [12] and [13] should be mentioned.

III. Overview

The framework consists of the following steps that are executed one after another:

1) generating an initial map using voronoi regions, chapter IV
2) editing the terrain by user-defined features, chapter V
3) simulating water and erosion to increase the realism, chapter VI
4) define biomes and populate the scene with grass and trees, chapter VII

For consistency, all examples in this paper show the same terrain in the various stages of development.

IV. Polygonal Map

As a first step, we have to come up with a good initial terrain to help the user with the terrain features. These techniques are taken from [9].

A. Voronoi Regions

We could start with a regular grid of squares or hexagons, but using a Voronoi diagram provides more randomness.

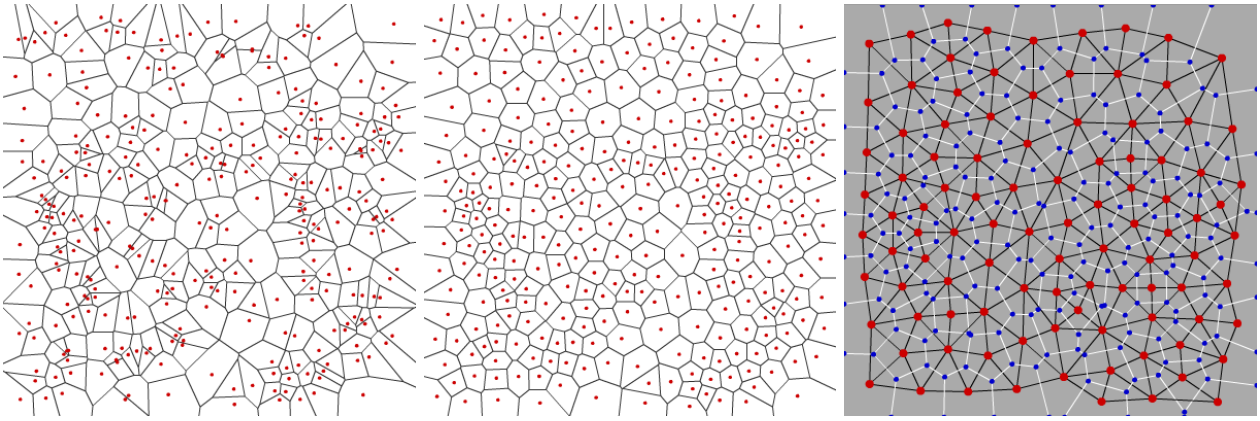
At the beginning, we generate random points, called center points, in the plane from -1 to 1 in x and y coordinate and compute the Voronoi diagram (Fig. 1a). The Voronoi diagram assigns each center point a region on the plane to where this center point is the closest. These regions are always convex polygons. The dual problem, the Delaunay triangulation, connects two center points by an edge if and only if the two voronoi regions of the two center points have a common border.

Because the shapes of the polygons are too irregular, we perform a relaxation step by replacing each center point by the average of the polygon corners (Fig. 1b). Applying this step several times results in more and more uniform polygons. In practice, two or three iterations provide good results.

B. Graph representation

For further processing of the polygons, we have to build a graph representation (Fig. 1c). The graph datastructure consists of the following three classes:

```
class Center {
    int index;
    Vector2f location;
    boolean water;
    boolean ocean;
    boolean border;
    Biome biome;
    float elevation;
```



(a) initial region (b) relaxation (c) graph representation

Figure 1: Initial polygon map

```
    float moisture;
    float temperature;
    List<Center> neighbors;
    List<Edge> borders;
    List<Corner> corners;
}

class Edge {
    int index;
    Center d0, d1; //Delaunay
    Corner v0, v1; //Voronoi
    Vector2f midpoint;
    int riverVolume;
}

class Corner {
    int index;
    Vector2f point;
    boolean ocean;
    boolean water;
    boolean coast;
    boolean border;
    float elevation;
    float moisture;
    float temperature;
    int river;
    List<Center> touches;
    List<Edge> incident;
    List<Corner> adjacent;
}
```

The class Center stores the center of a voronoi region, and the vertices of the voronoi polygons are represented by Corner. Because of the duality between the Voronoi Diagram and Delaunay triangulations, an edge connects both two corners (as an edge of a voronoi polygon) and two centers (as an edge in the delaunay triangulations). The other properties of the three classes are needed in the next step.

C. Island shape

As a next step, we have to define the general shape of the island. A simple way to get the desired shape is to combine perlin noise[1] with a distance from the center. Let (x, y) be a point and we want to know if it is in the water or not.

$$noise := \sum_{i=0}^{n} \text{noise}(x\ o_b\ o_s^i, y\ o_b\ o_s^i)\ a_b\ a_s^{-i} \quad (1)$$

$$water(x, y) := noise < offset + scale\ (x^2, y^2) \quad (2)$$

The function $water(x, y)$ returns true iff the point (x, y) is in the water. The function noise(x,y) returns the value of the perlin noise at the specified position.

There are many values to tweak: n defines the number of octaves that are summed together, o_b specifies the base octave, o_s the octave scale, a_b the base amplitude and a_s the amplitude scale. $scale$ defines the influence of the distance to the center of the map versus the perlin noise and $offset$ specifies the base island size. In our experiments, we use the following values: $n = 4, o_b = 6, o_s = 2, a_b = 0.5, a_s = 2.5, scale = 2.2, offset = -0.2$.

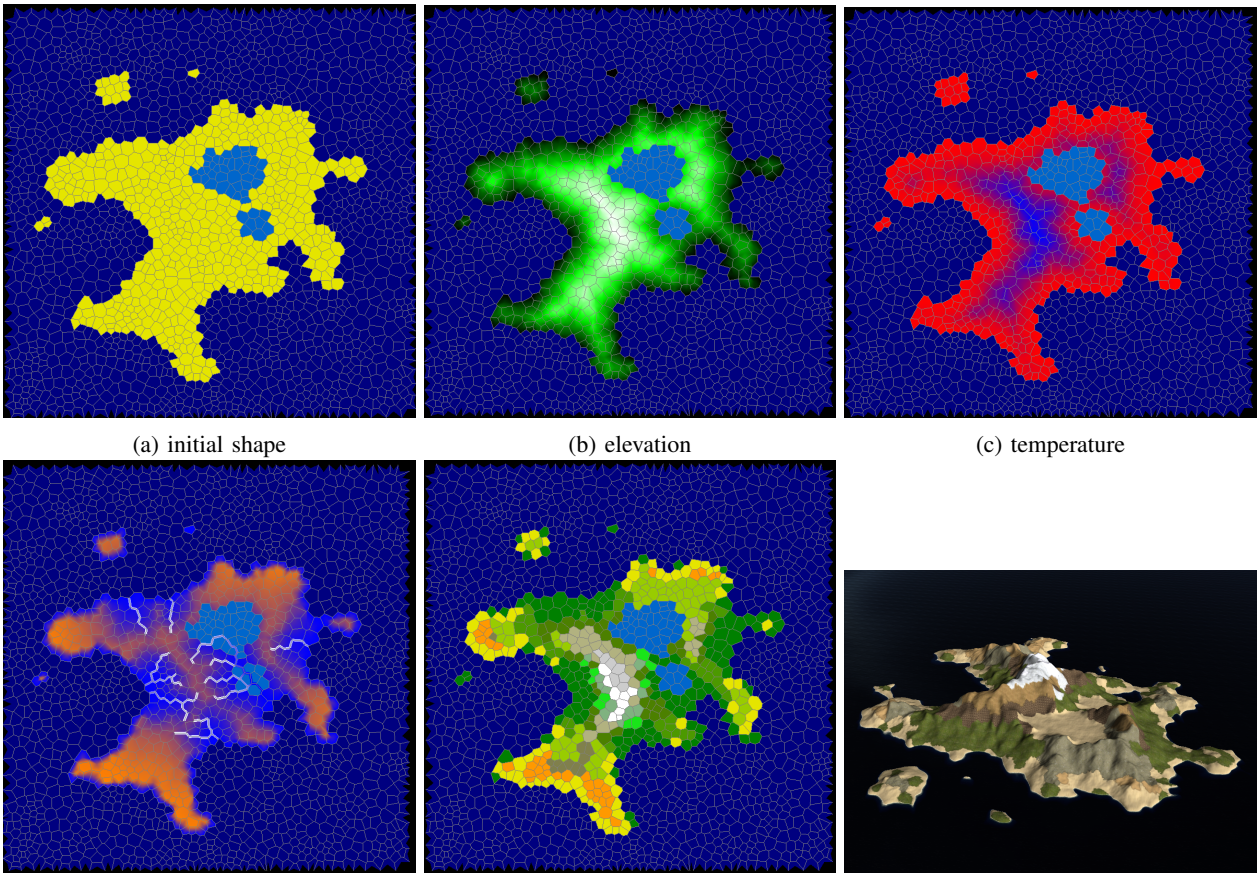
This function is evaluated for every corner (stored in the water-property). A center is labeled as water if at least 30% of the corners are water. The result can be seen in Fig. 2a

To distinguish between oceans and lakes, we use a flood fill from the border of the map and mark every water polygon on the way as 'ocean' until we reach the coast. Corners are marked as 'coast' if they touch centers that are ocean and land.

D. Elevation

After creating the initial island shape, we assign elevation values to every corner and center (Fig. 2b). For that we start from every coast corner and traverse the graph using breath-first along the corners. When traversing over land, we increase the elevation and over the sea we decrease the elevation. By that we obtain mountains and a continental shelf. Only when walking along or over lakes, we do not increase the elevation. This leads to flat areas that are later filled with water.

[1] http://mrl.nyu.edu/ perlin/doc/oscar.html



(a) initial shape (b) elevation (c) temperature

(d) moisture (e) biomes (f) generated heightmap

Figure 2: Elevation and Moisture

From the elevation, we directly derive the temperature: The higher it is, the colder it is. (Fig. 2c).

Until now, the elevation and temperature were defined for corners. To get these values for centers as well, we simply average them.

E. Moisture

To obtain moisture values, we start with creating rivers.

First we select random corners that are not coast or water and from there we follow the path of the steepest descent until reaching the ocean. At each step, we increase the amount of water the river carries. We then initial the moisture value of the river corners with the water amount. To achieve more distributed rivers, you might start a new river only at corners that are at least one or two steps away from an existing river.

Second we perform a breath-first-search starting from river corners. At each step, we decrease the moisture by a multiplicative factor of e.g. 0.8 (works well in our experiments). By that, the rivers spread their moisture over the land depending on the amount of water they carry.

Third we average again the moisture of each corner to obtain the moisture per center. The result can be seen in 2d.

As a last step, we assign a biome to each center based on its temperature and moisture (Fig. 2e). This acts as a starting point for chapter VII. More details on the biomes are described in section VII-A.

F. Generating the height map

1) Heightmap: The next steps in the processing pipeline all require a height map. A height map is just a rectangular grid of values where each cell stores the height at this point. We also use the same datastructure to store moisture and temperature values.

Notation: Let H be the heightmap. Then we identified the cell at position i, j with either $H_{i,j}$ or $H[i, j]$, depending on the readability. When coordinates lie outside the map, we clamp them. When the coordinates are not integers, but lie between cells, we perform a bilinear interpolation between the neighboring cells. When two heightmaps, or in the general case two scalar- or vectorfields, are added or multiplied together in this paper, these are always element-wise operations.

2) Base elevation: Since we already display the polygon graph from the previous steps using triangle meshes, we can just render the elevation mesh from step IV-D to obtain a base elevation of the generated terrain.

However, the straight corners of the polygonal cells are still visible. Therefore we have to distort the height map by replacing each height with the height of the cell an offset away. The offset is defined by a perlin noise. For more details on this, see chapter "Combination and perturbation" in [2]. The increase the height difference between flatlands

and mountains, we apply the following scaling to the height values:

$$h \leftarrow sign(h) \cdot |h|^{1.5} \tag{3}$$

3) Adding noise: The terrain still looks very boring, we need more noise to create hills and mountains. Therefore we add a multi fractal noise to the heightmap. The equation is taken from Chapter 4.3 of [8].

$$N(x,y) := A(x,y) \sum_{k=0}^{n} \frac{\text{noise}(r^k x, r^k y)}{r^{k(1-R(x,y)}} \tag{4}$$

noise is again the perlin noise function, r is the octave factor (here $r = 2$) and n is the number of octaves (we use $n = 5$). The scalarfield A defines the amplitude of the noise at the given terrain position and R the roughness (how the octaves are mixed together). In our experience we compute A and R based on the biomes using the values from table I. The noise

Biome	A	R
Snow	0.5	0.7
Tundra	0.5	0.5
Bare	0.4	0.4
Scorched	0.7	0.3
Taiga	0.4	0.3
Shrubland	0.5	0.2
Temperate desert	0.1	0.1
Temperate rain forest	0.3	0.2
Deciduous forest	0.3	0.4
Grassland	0.4	0.5
Tropical rain forest	0.3	0.2
Tropical seasonal forest	0.3	0.2
Subtropical desert	0.3	0.6
Beach	0.3	0.5
Lake	0.2	0.2
Ocean	0.2	0.1

Table I: Noise properties

value N is then simply added to the final height map. At the end, we normalize the map so that a height of zero is the ocean level, 1 is the highest mountain and -1 is the deepest sea.

The result can be seen in Fig. 2f. (The seed used is 658175619cff)

G. User interaction

In our implementation, the user can modify the elevation, temperature and moisture values starting from the presented initial values. By that, the user can create custom island shapes, raise mountains, build valleys and define the biomes by editing the temperature and moisture values.

With the terrain created in this step, we can proceed to the next step.

V. Terrain Features

Starting from the terrain from the previous chapter, the user now has the ability to fine-tune the terrain features. This is an implementation of [8] with some extensions. More examples and further explanations are available in this paper.

A. Feature curves

In the center of this processing step stand feature curves. Feature curves are added by the user and a solver (V-B) then modifies the terrain with respect to them. Example feature curves forming hills can be seen in Fig. 3a (white spheres are the control points and the blue tupe is the interpolated curve) and the resulting terrain in Fig. 3b.

Each feature curve consists of two or more control points. Each control point has the following properties:

- a position (x, y, z) in the world
- a boolean if it constraints elevation
- a float $plateau$, specifying the size of the flat plateau on top of the curve
- four floats $s_l, \varphi_l, s_r, \varphi_r$ specifying the size and angle of the slopes left and right of the curve

In the example in Fig. 3, the slope sizes are zero, so no slope constraints are applied.

Between the control points, the position is interpolated using cubic hermit splines and quadratic hermit splines at the start and end point (I use the code from the CurveAverage project). The other constraining values are linearly interpolated.

B. Diffusion solver

The task of the diffusion solver is to modify the terrain so that it remains smooth while preserving the feature constraints. The diffusion solver iteratively updates the terrain using the following recursion: Let H^i be the height map at the i-th iteration.

$$\begin{aligned} H^{k+1}[i,j] &= \alpha[i,j]E[i,j] \\ &+ \beta[i,j]G^{k+1}[i,j] \\ &+ (1-\alpha[i,j]-\beta[i,j])L^{k+1}[i,j] \end{aligned} \tag{5}$$

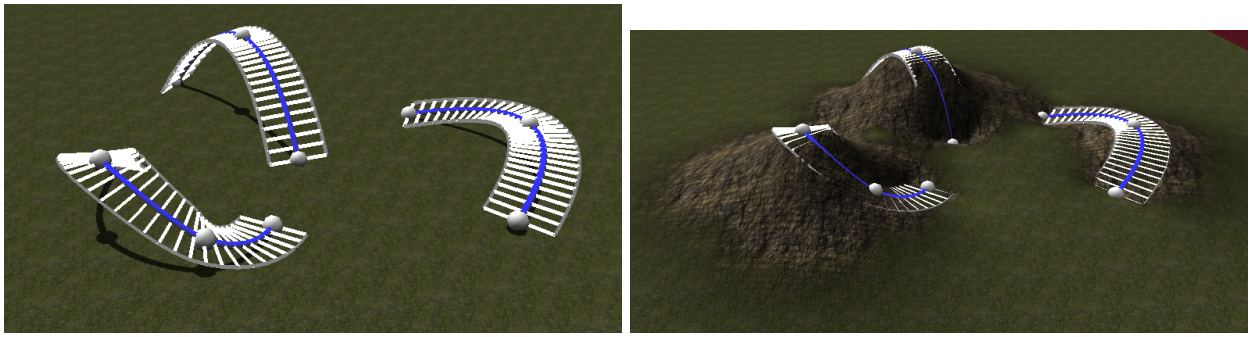
E is the forced elevation at the given point, it is directly created from the elevation constraints of the feature curves. G describes the gradient or the slope and L is a Laplace smoothing term. The values α and β, with $0 \leq \alpha \leq 1, 0 \leq \beta \leq 1, \alpha + \beta \leq 1$, specify the weighting of the single term. On the plateaus of the feature curves we set $\alpha = 0.9$ and $\beta = 0$. This forces the elevation to match the desired height while adding a little bit of smoothing to it. During slopes of the feature curve, we use $\alpha = 0$ and $\beta = 0.5$. By that, both the slope constraints and smoothing constraints are satisfied. Outside of the influence of the feature curves, we set $\alpha = 0, \beta = 0$.

The gradient term is defined in the following way:

$$G^{k+1}[i,j] = N^k[i,j] + G[i,j] \tag{6}$$

$$N^k[i,j] = \mathbf{n}_x^2 H^k[i - \text{sign}(\mathbf{n}_x), j] + \mathbf{n}_y^2 H^k[i, j - \text{sign}(\mathbf{n}_y)] \tag{7}$$

The gradient equation tends to satisfy the provided gradient G. G is directly calculated from $\sin(\varphi_l)$ and $\sin(\varphi_r)$, i.e. from the slope angles defined at the control points. The vector $\mathbf{n} = (\mathbf{n}_x, \mathbf{n}_y)$ describes the normalized direction of the gradient. This vector is orthogonal to the feature curve when projected in the plane and point away from the curve. In Fig. 4b, some of them are displayed as gray lines going away from the plateau.



(a) feature curves (b) resulting terrain

Figure 3: Defining feature curves

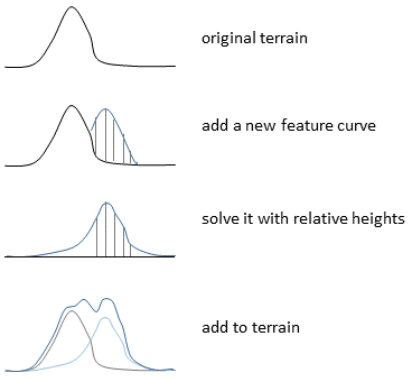
The Laplace term smoothes the terrain by averaging the neighbor cells:

$$L^{k+1}[i,j] = \frac{1}{4}(H^k[i-1,j] + H^k[i+1,j] + H^k[i,j-1] + H^k[i,j+1]) \quad (8)$$

C. Integration into existing terrain

Defining the whole terrain only with the feature curves is very tedious. Therefore we want to start with an existing terrain, as described in chapter IV. However, simply starting with the existing terrain as H^0 does not work because the Laplace term would just smooth out every feature apart from the feature curves.

[7] proposes a different solution. They compute the elevation and gradient constraints relative to the original terrain, then solve the diffusion equation on a flat map and add them together. The idea is visualized in the following image:



This works as long as there are no sharp features already existing right next to the new feature curve. If that is the case, as in the image above, adding the curves together produces unwanted results because of these inferences.

This case happens quite often as seen in Fig. 5. Here we define a new feature curve right next to an existing mountain. Although the curve's slope does not touch the existing terrain, the smoothing term inside the diffusion solver also increases the height outside of the feature curve. This leads to a growth of the mountain on the left (the tip of the mountain is now much higher than the control point, before it was about the same height). From a designer's perspective, this behavior might not be desired because it is hard to predict and control.

D. Extension of the solver

We propose now another solution that both keeps existing terrain features while it preserves the local control and removes the inferences mentioned before. The idea is to limit the smoothing term L to influence only a customizable region around a feature curve. For this, we add two new parameters, l_l and l_r, to each control point. They specify the local radius of an envelope around the plateau and slope of the feature curve.

We then set γ to 1 inside this envelope and to 0 outside and modify equation 5 to include γ:

$$H^{k+1}[i,j] = \alpha[i,j]E[i,j] + \beta[i,j]G^{k+1}[i,j] + \gamma[i,j](1-\alpha[i,j]-\beta[i,j])L^{k+1}[i,j] \quad (9)$$

Because we no limit the smoothing area, we can apply the solver directly on the original height map without smoothing out existing features. There is no need any more to solve the diffusion equation on a separate map with relative heights as done in the previous section.

In Fig. 5c you see the problematic situation again, but this time, the smoothing region is bounded as described before. The border of the smoothing envelope is visualized by the black lines. Note that the old mountain is preserved and the new feature is blended into the terrain without discontinuities.

VI. Hydraulic Erosion

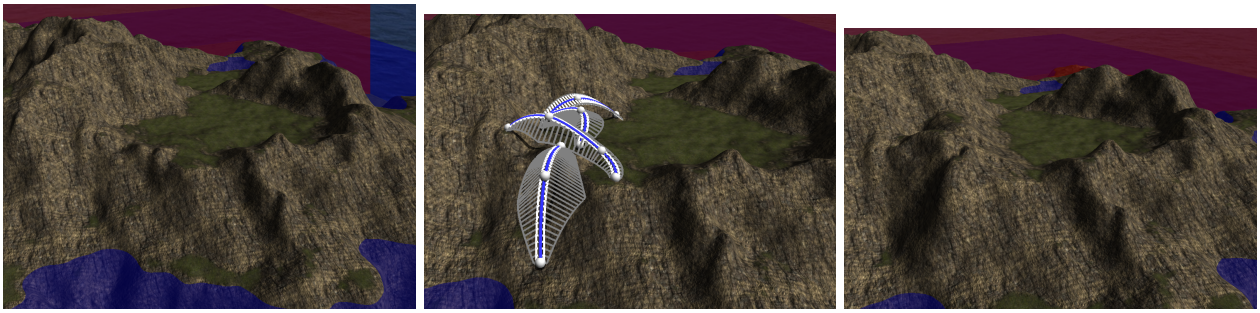
Now the shape of the terrain is fully defined, and the next step is to increase the realism by performing erosion.

Erosion comes in many shapes in the nature: there is thermal erosion which decomposes larger stones into smaller ones, wind erosion which carries sand over long distances and finally water erosion. Water erosion or hydraulic erosion is caused by flowing water that erodes the terrain, transports sediment along the river and finally deposits it somewhere. Since the hydraulic erosion is the strongest one, we focus on this type of erosion in the presented framework.

We use an adapted version of the model presented in [5].

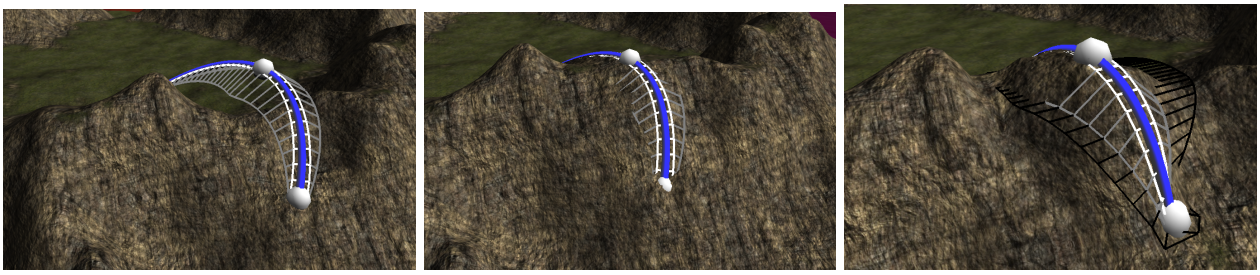
A. Erosion Model

The algorithms follows the following steps, which are repeatedly executed using the timestep Δt (e.g. $\Delta t = 0.01$):



(a) original terrain (b) adding feature curves (c) resulting terrain

Figure 4: Adding features to existing terrain



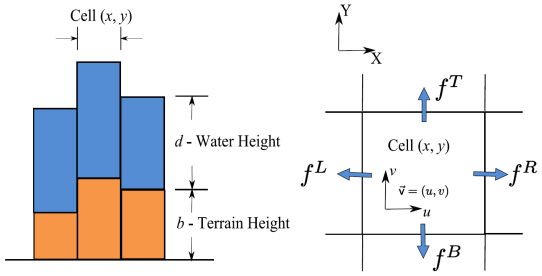
(a) original terrain with new feature curve (b) inference, left mountain is modified (c) limiting smoothing to remove inference

Figure 5: Inferences between new feature curves and existing terrain features

1) water increment by rainfall
2) flow simulation
3) water volume change
4) erosion, deposition and sediment transportation
5) evaporation

In the next sections, we use the following notation for describing the involved variables, all of them are scalarfields or vectorfields on the grid of the terrain:

- terrain height b
- water height d
- suspended sediment s
- outflow flux $\mathbf{f} = (f^L, f^R, f^T, f^B)$
- water velocity $\mathbf{v} = (u, v)$



1) Water increment: Rainfall is simulated by sampling water drops over the terrain. The probability of each cell to 'emit' a water drop is equal to the moisture. The moisture map is initialized with the information created at the very beginning, in chapter IV-E. The user can edit this moisture value before starting the simulation. By using random rain drops instead of a global water increment, we add more randomness to the scene. A single raindrop erodes the terrain a little bit, forming a tiny river bed. Next drops that are created nearby are now more likely to follow this path as well. This then leads to formation of river beds instead of just removing sediment from slopes uniformly.

In addition, the user can place river sources on the terrain. They act as a constant source of water.

Let $r_t[x, y]$ be the water that arrives at the current time step at position (x, y). Then the water height is modified in the following way:

$$d_t[x, y] = d_t[x, y] + \Delta t \cdot r_t[x, y] \tag{10}$$

2) Flow simulation: In the flow simulation, we simulate the way water flows from higher positions to lower positions. We approximate this by using virtual pipes that connect two adjacent cells in the grid. The amount of water that flows from one cell to the other is called the outflow flux.

The outflow flux from the current cell to the neighbor cell to the left is computed as follows:

$$f^L_{t+\Delta t}[x, y] = \max(0, f^L_t[x, y] + \Delta t \cdot g \cdot \Delta h^L_t[x, y] \tag{11}$$

Whereby the height difference to the left cell is calculated using:

$$\Delta h^L_t[x, y] = b_t[x, y] + d_t[x, y] - b_t[x-1, y] - d_t[x-1, y] \tag{12}$$

f^R, f^T and f^B are calculated in the same way. The gravitation constant g specifies the amount of acceleration of the flow by the height difference. In our experiments, we set $g = 10$.

It can now happen that after subtracting the sum of the outflow flux from the water height (see VI-A3), the water height becomes negative. To avoid this, all four outflow

flux values are scaled with the following scaling factor K (evaluated cell-wise).

$$K = \min(1, \frac{d}{(f^L + f^R + f^T + f^B) \cdot \Delta t}) \tag{13}$$

3) Water volume change: After computing the outflow flux, we can define the change of the water volume as:

$$\begin{aligned} \Delta V[x,y] &= \Delta t \cdot (\sum f_{in} - \sum f_{out}) \\ &= \Delta t \cdot (f^L_{t+\Delta t}[x+1,y] + f^R_{t+\Delta t}[x-1,y] \\ &\quad f^B_{t+\Delta t}[x,y+1] + f^T_{t+\Delta t}[x,y-1] \\ &\quad - \sum_{i \in \{L,R,T,B\}} f^i_{t+\Delta t}[x,y]) \end{aligned} \tag{14}$$

The water height per cell is then modified using:

$$d_{t+\Delta t}[x,y] = d_t[x,y] + \Delta V[x,y] \tag{15}$$

For the next step, VI-A4, we need to compute the horizontal velocity of the water $\mathbf{v} = (u, v)$. Let $\bar{d}$ be the arithmetic average of the water height before and after equation 15. Then the velocity along the x-axis is computed as:

$$u = \frac{f^R[x-1,y] - f^L[x,y] + f^R[x,y] - f^L[x+1,y]}{2\bar{d}} \tag{16}$$

v, the velocity along the y-axis is computed similarly.

4) Erosion, deposition and sediment transportation: To compute the sediment erosion and deposition, we first define the slope at a cell as:

$$\begin{aligned} \alpha[x,y] = \max(\quad & |b[x,y] - b[x-1,y]|, \\ & |b[x,y] - b[x+1,y]|, \\ & |b[x,y] - b[x,y-1]|, \\ & |b[x,y] - b[x,y+1]| \ ) \end{aligned} \tag{17}$$

Then the sediment transportation capacity C is calculated as:

$$C[x,y] = K_c \cdot \alpha[x,y] \cdot |\mathbf{v}[x,y]| \tag{18}$$

In our experiments, we set the sediment capacity constant K_c to 0.1. We recommend to clamp the values of α to be between e.g. 0.005 and 0.5 before computing C. By this, the erosion on steep areas does not grow to infinity and on flat areas some erosion still takes place.

Then we compare C to the suspended sediment s_t. For $C > s_t$, sediment is eroded using:

$$\begin{aligned} b_{t+\Delta t} &= b_t - K_s(C - s_t) \\ s' &= s_t + K_s(C - s_t) \end{aligned} \tag{19}$$

And for $C < s_t$, sediment is dissolved:

$$\begin{aligned} b_{t+\Delta t} &= b_t + K_d(s_t - C) \\ s' &= s_t - K_d(s_t - C) \end{aligned} \tag{20}$$

K_s and K_d are erosion and deposition constants. In our experiments, we use $K_s = K_d = 0.002$.

After sediment is eroded, we have to transport it to next cells using the water velocity. For that purpose, we use a backward Euler-step:

$$s_{t+\Delta t} = s'[x - u \cdot \Delta t, y - v \cdot \Delta t] \tag{21}$$

5) Evaporation: As a last step, some water is evaporated in the air every step. We model this simply as

$$d_{t+\Delta t}[x,y] = d_{t+\Delta t}[x,y] \cdot (1 - K_e \cdot \Delta t) \tag{22}$$

With K_e being an evaporation constant, e.g. $K_e = 0.1$.

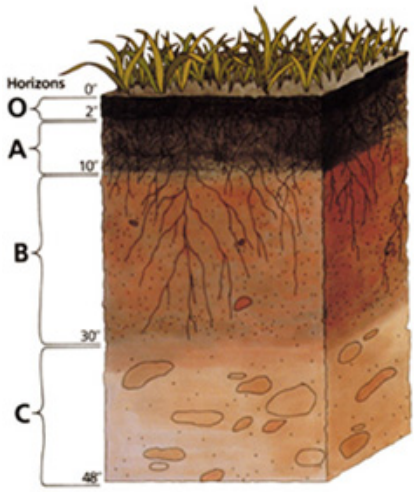
B. Adaptions

We made a few changes to the original model that in our opinion increase the realism and visual appearance.

1) non-uniform temperature: The original model assumes a uniform temperature over the whole scene. Since we want to model both hot deserts and cold snow areas, we cannot apply this in our situation. We give the user the ability to edit a temperature map T (from 0=cold to 1=hot) that is initialized with the temperature from IV-D. We then modify equation 22 to

$$d_{t+\Delta t}[x,y] = d_{t+\Delta t}[x,y] \cdot (1 - K_e \cdot \Delta t \cdot T[x,y]) \tag{23}$$

2) non-uniform erosion and deposition constants: In VI-A4 we assumed the erosion constant K_s and the deposition constant K_d to be constant. However, in a natural soil, this is not the case.



[2]

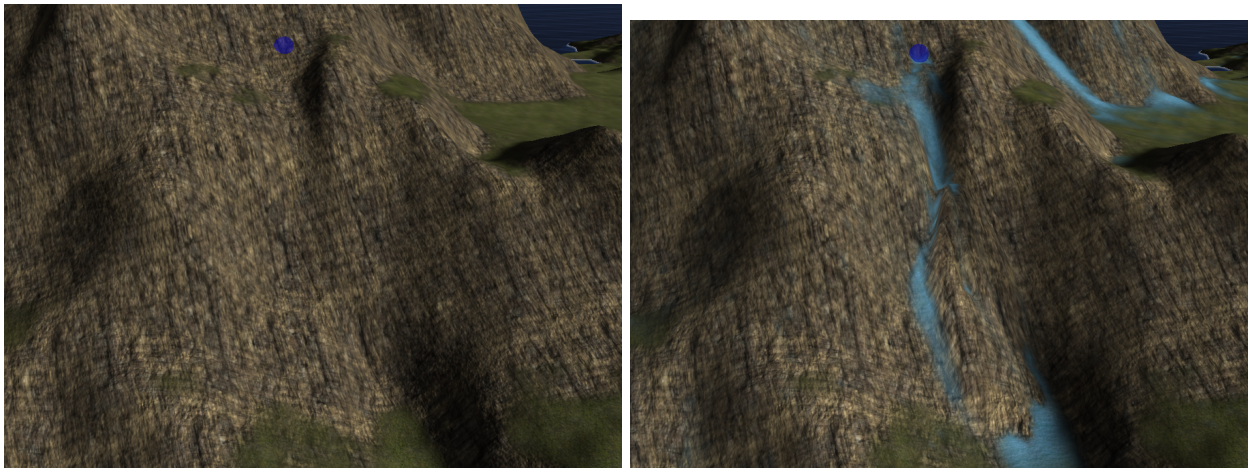
The top layers (soil) are eroded more easily than the bottom layers (stone). Newly dissolved sediment is eroded even better again because it consists mostly of loose stones and sand. We model this by representing K_s and K_d as functions of the amount of eroded terrain Δb.

$$\begin{aligned} \Delta b_t[x,y] &= b_{\text{original}}[x,y] - b_t[x,y] \\ K_s(\Delta b) &= K_{s,\text{base}} \cdot 2^{-\Delta b / f_s} \\ K_d(\Delta b) &= K_{d,\text{base}} \cdot 2^{\Delta b / f_d} \end{aligned} \tag{24}$$

$K_{s,\text{base}}$ and $K_{d,\text{base}}$ are set to 0.002 as before; f_s and f_d specifies the amount of terrain that has to be eroded/deposited before the speed of the erosion/deposition is halved.

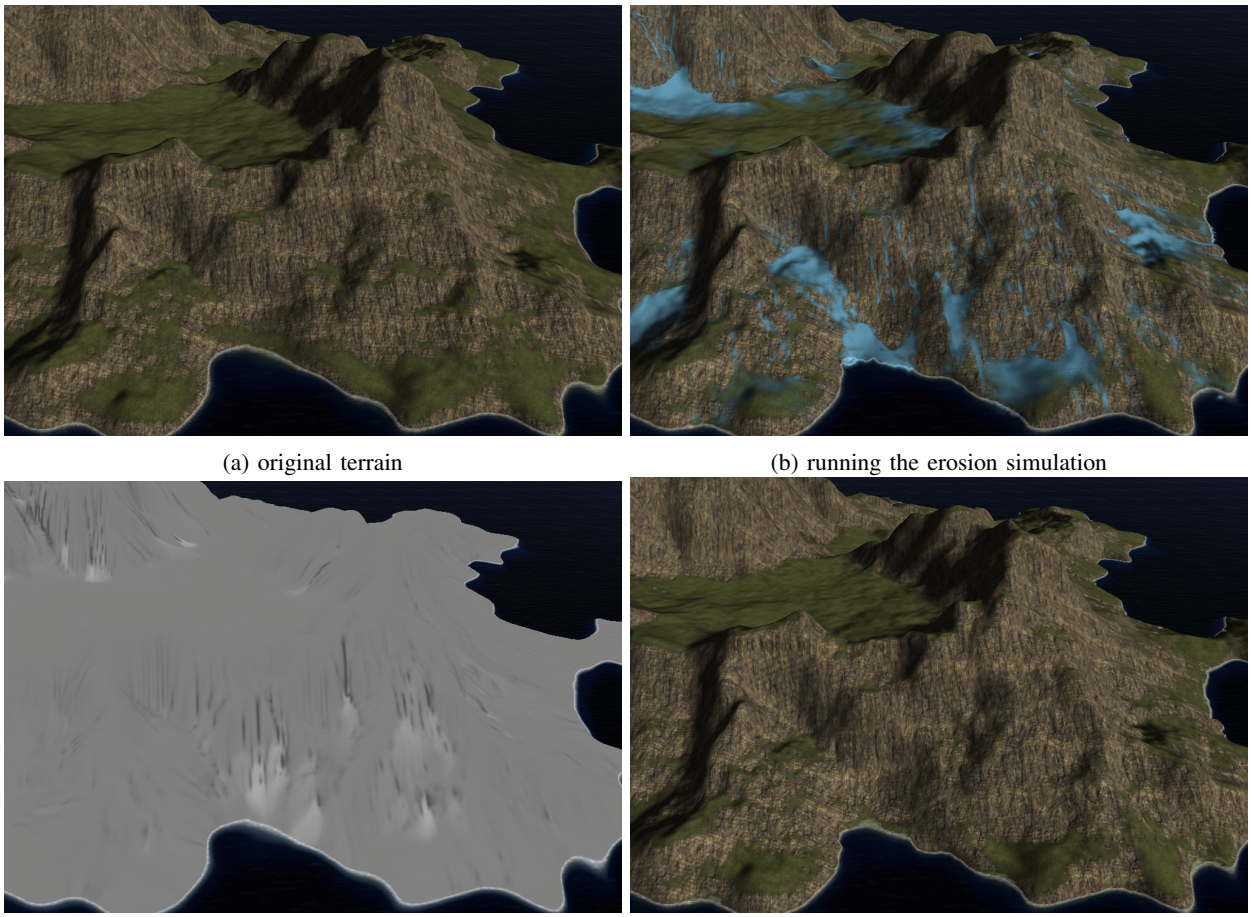
For total accuracy, you would have to integrate these functions over the whole erosion/deposition process. For simplicity, we just evaluate them once and assume them to be constant for one time step.

[2]https://upload.wikimedia.org/wikipedia/commons/9/95/Soil_profile.png



(a) original terrain with river source (b) river bed is carved out

Figure 6: Erosion and deposition of river beds



(a) original terrain (b) running the erosion simulation

(c) changes in the height map (d) final terrain

Figure 7: Erosion and deposition of mountains

In addition, we limit the total amount of eroded or deposited terrain per cell. This is needed because when the simulation is running for a longer time, very steep cliffs are carved out of regions with a steep slope. A bound on the change of the terrain provides visually more pleasant results.

3) Boundary conditions: Since we assume that our terrain forms an island, we have to treat the case when water reaches the ocean in a special way. In the case of a cell that is below the see level, we simply set the water height, outflow flux and velocity to zero and add the whole sediment amount to the terrain height.

C. Limitations

The described model is very sensitive to the chose of the parameters. The timescale must be small enough so that no numerically instabilities are created, but this directly leads to a long computation time. Also the constant values in the erosion model are very crucial: if the factors are too high, the terrain is eroded very quickly. This looks very unrealistic. The presented values produce good results in our experiments, but a deeper analysis of the parameters is necessary in a future work.

D. Results

In Fig. 6, you can see how the erosion simulation carves out a river bed from a specified river source.

Fig. 7 shows the simulation on a larger scale. The original terrain is shown in 7a, then in 7b, the simulation is running. The displayed terrain height is the terrain height plus the water height, so lakes, waves and so on can be seen. The water height also specifies how much "blue" is drawn on top of the terrain. The terrain is textured with grass until a specific slope angle is reached, then stone is used (with blending between them). By that, you can get a feeling of the local slope angle. In 7c, you can see the difference from the original terrain to the new terrain. White indicates depositions, black stands for erosion. Finally, 7d shows the final terrain without water. Especially note the change in the coast line and that the terrain is much more "bumpy", the stone texture is used more often.

VII. Vegetation

Terrain without vegetation looks a little bit boring. Therefore, the last step of the framework is to define biomes and add trees and grass based on them.

A. Biomes

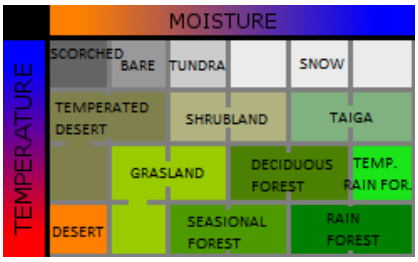


Figure 8: Biomes

Biomes are a system to classify the nature. A biome defines a region with common climate, soil and vegetation. An attempt to classify the different biomes based on rainfall and temperature was made in [14]. We are using an adapted version of the classification from [2], see Fig. 8.

In our implementation, the biomes are initialized with the temperature and moisture information from the previous steps. The user can then paint the different biomes directly on the terrain to fine-tune the biome assignment.

The terrain can be rendered using different textures for each biome (Fig. 9). Furthermore, the biomes specify the existence and types of grass and trees.

B. Grass

For rendering grass, we use the concepts presented in [10]. An implementation of it comes with the 3d engine we are using, the jMonkeyEngine[3]. A blade of grass is displayed using two transparent quads that are orthogonal to each other (Fig. 10a). The shades are then randomly placed on the terrain, the density distribution varies with the biomes. Finally, a texture is applied to complete the grass rendering (Fig. 10b).

C. Trees

Generating trees is a little bit more complicated. Trees have a very difficult 3d shape because they have many branches and leaves. In most computer games, trees are rendered using very primitive objects: stems are rendered as textured cylinders and bunches of leaves and small branches are combined and rendered using a single textured quad. This often looks good when viewed from the distance but the illusion breaks when you get to close to them.

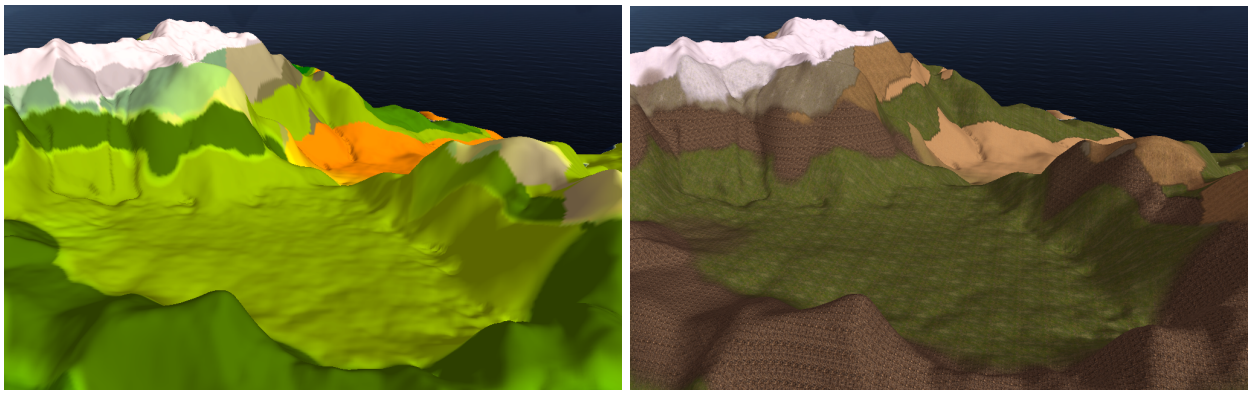
1) Tree generation: Since we also want to generate the trees procedurally, we have to go another way. In this framework, we use the algorithm presented in [13], implemented in [15]. Trees are recursivly defined by a bunch of parameters for each layer. The first layer is the stem, the last layer contains the leaves. These parameters include curvature, branch radius, splitting, branching and textures.

2) Level of Detail: The procedural tree generation algorithm has one huge drawback: it generates too many triangles. Since every leave is modeled as an individual quad and the stem and branches are very smooth, a typical tree contains up to one million vertices and half a million triangles. These are way too many to render more than a few trees at the same time.

Furthermore, because the tree model contains many small details, aliasing effects become a problem when the tree is rendered in the distance. The branches become so small that they are only rendered sometimes by the rasterizer. This leads to flickering effects.

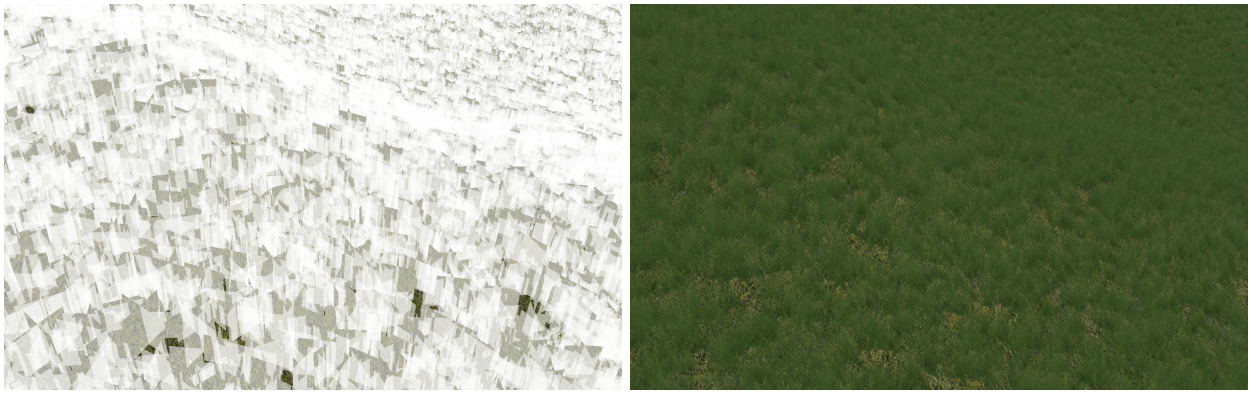
Both problems are adressed with a very simple technique based on impostors for rendering distant trees. impostors are textured quads that are rotated around a central axis. We are using eight quads for trees (Fig. 12). Note the similarity of the impostors to the way grass is rendered. Each quad is textured

[3]http://jmonkeyengine.org/



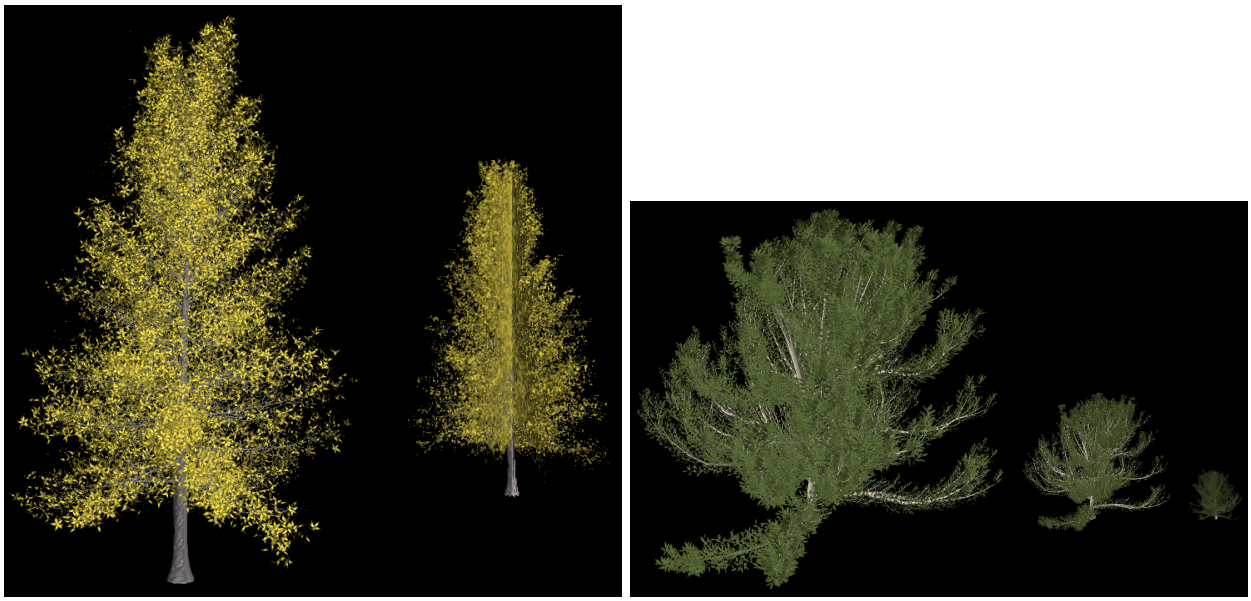
(a) biomes are painted on the terrain (b) textured terrain based on the biomes

Figure 9: Assigning biomes to the terrain



(a) transparent quads forming the grass blades (b) adding a texture leads to the final appearance

Figure 10: Grass



(a) black tupelo in fall (b) black oak

Figure 11: Grass

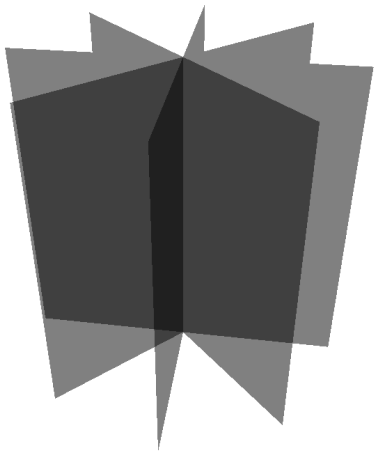


Figure 12: Impostor

with how the tree would look like when seen from the angle of that quad. To create these textures, we render the trees from every of these eight view angles into a offscreen surface and use this as the texture. During the creation of the textures, the tree must be rendered with exactly the same light setting as in the final scene. Otherwise, the lighting of the tree impostors is not consistent with the lighting of the surrounding terrain. This is especially necessary when a strong directional light is used.

To avoid popping effects when changing from the high resolution model for close-up shots to the impostors for distance rendering, we use alpha blending:

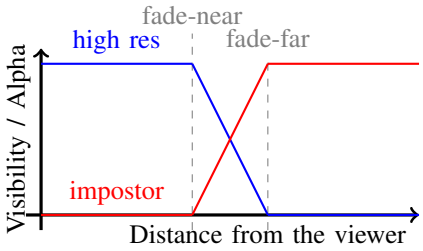


Figure 13: Blending between high resolution model and impostor

The parameters fade-near and fade-far specify the distances from the camera where the fading between the high resolution model and the impostor takes place. Because the memory of the graphics card is also limited, we only load the high resolution model into the graphics memory when the distance to the tree is smaller than the fade-far value.

Since the downscaling of textures can be done much smoother by the graphics card, impostors do not lack the problems of aliasing like the high resolution models do.

In Fig. 11 you can see two types of trees. The high resolution tree is rendered on the left (see VII-C1) and the low resolution version for distance rendering is drawn on the right (see VII-C2).

3) Tree placement: As a last step, the trees must be placed in the scene. For that, we first build a table by hand listing which trees occur in which biome with what probability. Then for every position on the height map (integer positions) we place a tree with a probability based on the general density of trees in the biome at that position and a constant density factor. This density factor is a parameter that has to be adjusted to match the desired global tree density. If a tree should be placed, we sample the type of tree from the tree table built at the beginning. To add more randomness, we place the tree at a random point on the terrain within the area of the current cell.

VIII. Conclusion and Future Work

With the framework described in this paper, we are able to produce realistic terrain with many possibilities for artistic control. The user can control the shape of the island, can grow mountains and other terrain features, and can define vegetation based on biomes. To increase the realism of the terrain, we apply a hyrdaulic erosion scheme to it.

There are many possibilities for extensions of this project. A more sophisticated approach for the rendering of grass and trees, as described in [11], could be used. We further only support on type of grass that is planted on a fixed subset of biomes. Especially in rain forests, many bushes, farns, flowers, lianas and other plants apart from trees grow which are not included in our simulation. Furthermore, the diffusion solver (V) and erosion solver (VI) are implemented on the CPU. A port to the GPU would increase the performance significantly.

The next two pages show some screenshots from the terrain used as an example through this paper.

References

[1] J. Doran and I. Parberry, "Controlled procedural terrain generation using software agents," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 2, pp. 111–119, 2010.

[2] Jacob Olsen, "Realtime procedural terrain generation," 2004.

[3] Teong Joo Ong, Ryan Saunders, John Keyser, John J. Leggett, "Terrain generation using genetic algorithms," 2005.

[4] Bedrich Beneš, "Real-time erosion using shallow water simulation," 2007.

[5] X. Mei, P. Decaudin, and B.-G. Hu, "Fast hydraulic erosion simulation and visualization on gpu," in *15th Pacific Conference on Computer Graphics and Applications (PG'07)*, pp. 47–56.

[6] Jean-David Genevaux, Eric Galin, Eric Guerin, Adrien Peytavie, Bedrich Benes, "Terrain generation using procedural models based on hydrology," 2013.

[7] Flora Ponjou Tasse, Arnaud Emilien, Marie-Paule Cani, Stefanie Hahmann, Adrien Bernhardt, "First person sketch-based terrain editing," 2014.

[8] H. Hnaidi, E. Guérin, S. Akkouche, A. Peytavie, and E. Galin, "Feature based terrain generation using diffusion equation," *Computer Graphics Forum*, vol. 29, no. 7, pp. 2179–2186, 2010.

[9] Amit Patel, "Polygonal map generation for games," 2010. [Online]. Available: http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/

[10] Kurt Pelzer, "Rendering countless blades of waving grass," in *GPU Gems*.

[11] K. Boulanger, "Real-time realistic rendering of nature scenes with dynamic lighting," 2005.

[12] Adam Runions, Brendan Lane, Przemyslaw Prusinkiewicz, "Modeling trees with a space colonization algorithm," 2007.

[13] J. Weber and J. Penn, "Creation and rendering of realistic trees," in *the 22nd annual conference*, S. G. Mair and R. Cook, Eds., pp. 119–128.

[14] Marietta College, "Marietta college main biomes page." [Online]. Available: http://w3.marietta.edu/~biol/biomes/biome_main.htm

[15] Wolfram Diestel, "Arbaro," 2013. [Online]. Available: http://arbaro.sourceforge.net/

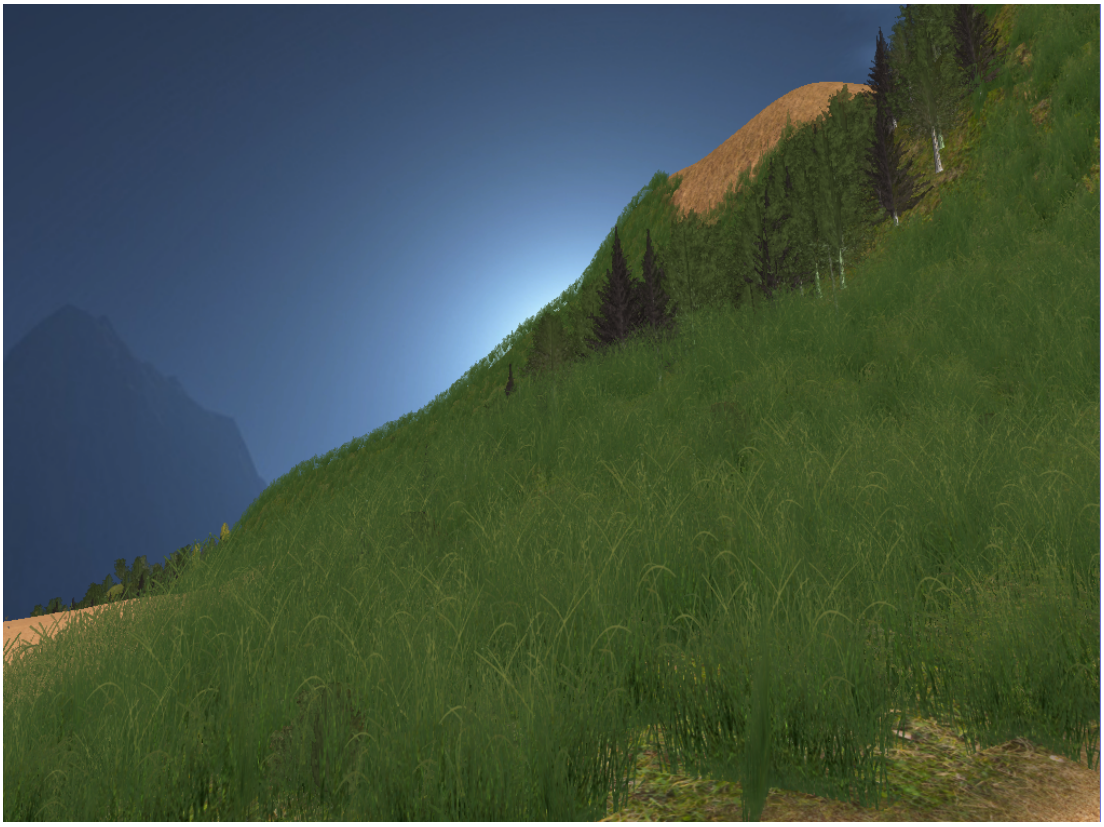


Figure 14: grassland

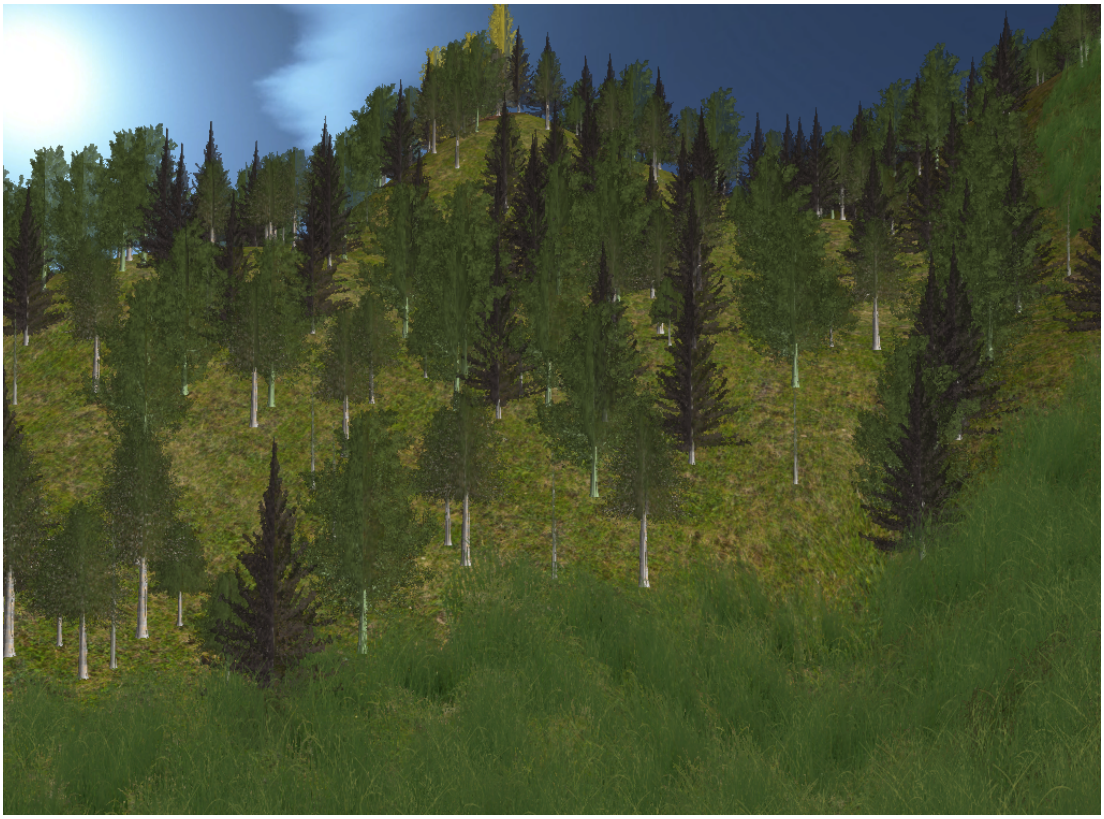


Figure 15: forest

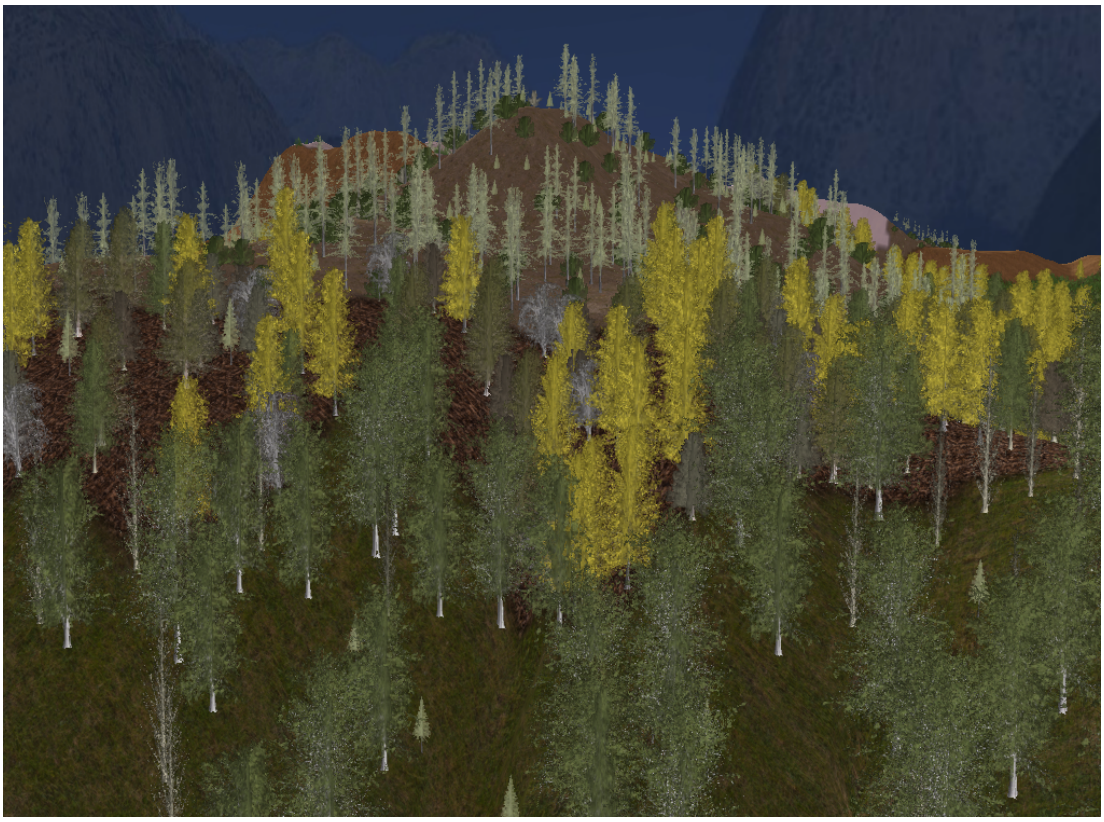


Figure 16: deciduous forest (foreground), taiga (background)

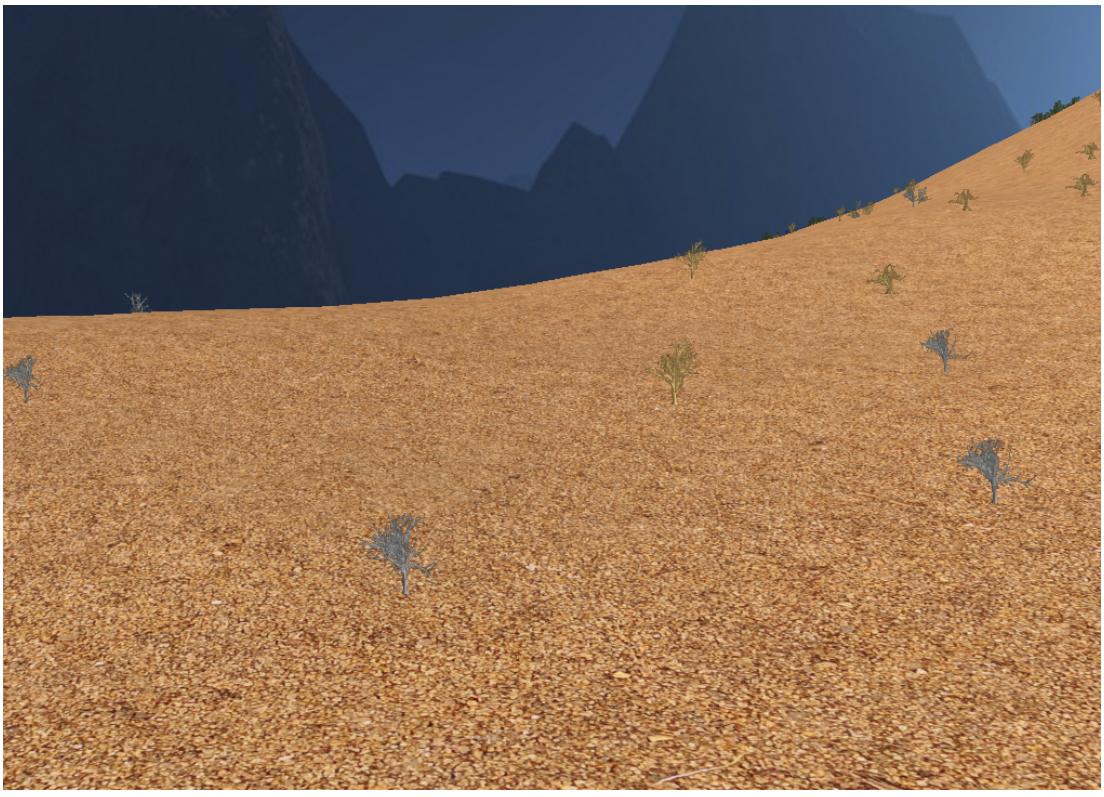


Figure 17: desert