IGR 205 - Interactive Layered Terrain Generation

Lucas Teissier, Octave Le Tullier, Thomas Poyet - Télécom Paris

Abstract

We present the implementation and the results of different methods and algorithms regarding terrain modeling. We broach thermal and hydraulic erosion, algorithms to generate or modulate terrains. We also go further with the implementation on GPU of some of these algorithms. We expose the issues we faced and the solutions we found.

1 Introduction and related work

Interactive layered terrain generation can be very useful in lots of fields, for instance with open world and simulations. Hence, we decided to tackle this subject and to implement different algorithms to play with the terrain. A lot of related works already exist, so we somewhat followed an overview of digital terrain modeling [1] to implement thermal erosion and algorithms for terrain generation (with noise or faults). Then, we delved into hydraulic erosion [3] and implementation on GPU [2]. Throughout the project, we tried to understand issues we faced, to fix them, and to tune the different parameters to get better results.

2 Thermal Erosion

Our first objective was to implement thermal erosion to simulate the matter that falls down when the slope is too steep (for instance the snow on top of a mountain).
Let h be the height in a point p, $s(p)$ the slope at this point - defined by the norm of the gradient of h at p -, θ the angle of talus, dt the step of integration, k a coefficient of erosion. The amount of matter dh lost by the point p is given by:

$$\frac{\partial h}{\partial t} = \begin{cases} -k(s(p) - tan(\theta)) & \text{if } s(p) > tan(\theta) \\ 0 & \text{else.} \end{cases}$$

So if $s(p) > tan(\theta)$, the point p gives

$$dh = -k(s(p) - tan(\theta))dt$$

2.1 Strategy A : with gradient

To start with, we chose to first implement the strategy of the paper [1]. It consists in computing the gradient to get the slope and moving the matter accordingly to the norm and the direction of the gradient, which give respectively the amount of the matter lost and the vertices that receive it : they are the vertices pointed by the opposite direction of the gradient.
However, with this method, we faced many issues, especially because of discretization, so we added parameters to let the user modulate this strategy.
First, we can choose how to compute the gradient. If h is the height, we can choose the forward method : $h(i+1,j) - h(i,j)$ and $h(i,j+1) - h(i,j)$, the backward method : $h(i,j) - h(i-1,j)$ and $h(i,j) - h(i,j-1)$ and the central method : $\frac{h(i+1,j)-h(i-1,j)}{2}$ and $\frac{h(i,j+1)-h(i,j-1)}{2}$.
We provide those three methods because different artifacts are created depending the method chosen (Figure 1). The last one gives the best results. But, even if artifacts are reduced, they are still present.

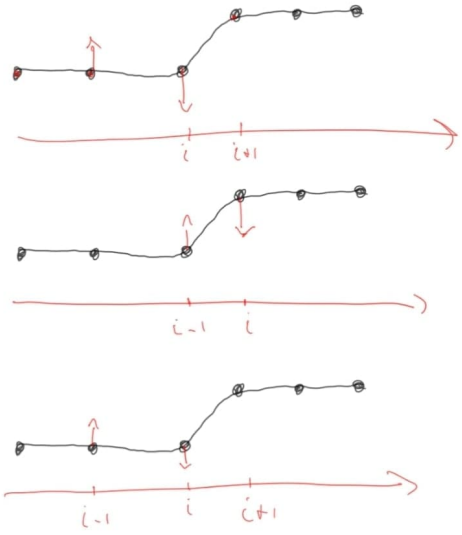


Figure 1: Illustration 1D of material transfer issues, according to the chosen gradient

With the forward method (first one), the point at i has a gradient proportional to $h(i+1) - h(i)$. If the norm of this gradient is greater than the slope given by the talus angle, this point will lose matter and give it to the neighbor pointed by the opposite direction of the gradient i.e. the point $i-1$. This will create an unwanted ditch at the point i. The expected behavior would be that point $i+1$ loses its material to give it to point i.
In this case, this issue is fixed with backward gradient (the second graph) but not with central gradient (third graph). Other cases can be found where the forward, or backward, or central method is better than the others.
The neighbors receiving the lost material can be tuned. We first implemented, as in the paper, the matter falling down to the neighbor pointed by the opposite direction of the gradient. But, due to discretization, issues can be observed. For instance, in Figure 2, the color intensity represents the height, the blue arrow is the direction of the gradient - computed with the central method -, the yellow arrows are the directions with neighbors lower than the central pixel. So, if the gradient direction is used to compute the neighbor receiving the matter, the matter will be received by the point above the central one i.e. to a point higher than the central one.

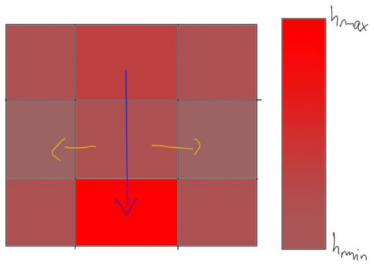
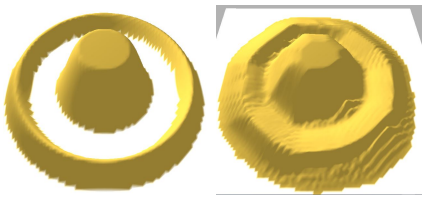


Figure 2: Issue with the descent's direction

To fix this issue, we can just give the matter to the lowest neighbor (so here to the one at the left or the right).
Finally, in the paper, they give matter only to one neighbor, but it's not really appropriate if there are many neighbors lower than the central one. Hence, the user can choose to give matter to all neighbors or to a unique neighbor. The amount of matter dh lost by a point is still given by the previous formula, but we split it according to the difference of height between all the lower neighbors $(i_l, j_l)_l$, so that each one receives a proportion of matter and the lowest one receives more matter :

$$dh_{(i',j')} = \frac{h(i,j) - h(i',j')}{\sum_{i_l,j_l}(h(i,j) - h(i_l,j_l))} dh$$

However, the first problem of ditch formation is an inherent problem of this implementation using the gradient. Therefore, we decided to adopt another strategy that does not require computing the gradient. A result of this strategy is shown in Figure 3.



(a) Original (b) Eroded terrain

Figure 3: Result of thermal erosion with strategy A

2.2 Strategy B : without gradient

The second strategy was inspired by the pipe model used in the hydraulic erosion and described in the part 3.1.
This model consists in computing for each point the amount of material it gives to each of its neighbors. A point (i,j) gives matter to its neighbors (i',j') - in 4 or 8 connexity - if the point $h(i,j) > h(i',j')$ and if the difference of height between them is greater than a threshold given by the talus angle of the material :

$$dh_{(i,j)\to(i',j')} = max(0, k(\frac{h(i,j) - h(i',j')}{d_{(i,j)\to(i',j')}} - tan(\theta))dt)$$

with $d_{(i,j)\to(i',j')}$ the distance between the two point.

However, the amount of matter lost by (i,j) should not exceed the thickness of the layer at (i,j). In this case, the all thickness at (i,j) is distributed to the lower neighbors according to a coefficient $\alpha_{(i,j)\to(i',j')}$ given by, if the neighbors of (i,j) are denoted by (i_n, j_n)

$$\alpha_{(i,j)\to(i',j')} = \frac{dh_{(i,j)\to(i',j')}}{\sum_{(i_n,j_n)} dh_{(i,j)\to(i_n,j_n)}}$$

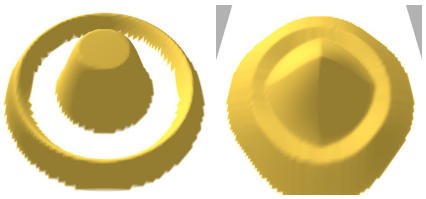
Which is equivalent to the following normalization, with $b_{(i,j)}$ the thickness of eroded material at (i,j) :

$$K = min(1, \frac{b_{(i,j)}}{\sum_{(i_n,j_n)} dh_{(i,j)\to(i_n,j_n)}})$$

$$dh_{(i,j)\to(i',j')} = K dh_{(i,j)\to(i',j')}$$

Finally, the thickness at the point (i,j) is updated by doing the sums of the amount of matter it loses and the amount of mater it receives from each of its neighbors.

$$b_{(i,j)} = b_{(i,j)} - \sum_{(i_n,j_n)} dh_{(i,j)\to(i_n,j_n)} + \sum_{(i_n,j_n)} dh_{(i_n,j_n)\to(i,j)}$$



(a) Original (b) Eroded terrain

Figure 4: Result of thermal erosion with strategy B

2.3 Other parameters

To improve the results of thermal erosion, we added other parameters that work for both strategies.

- The connexity for the distribution of matter : 4 or 8 (8 being better).
- The parameters of the formula : angle of talus θ, coefficient of erosion k, integration step dt (the smaller the smoother).
- The number of iterations.
- The way we want to give the matter : the matter is given to the top layer or to the same layer. For instance, imagine we have the layers from bottom to top : bedrock, rock, sand. If the layer rock is eroded for example, it can fall down and become sand : we increase the height of the top layer. Or it can fall down and give the same matter : we increase the thickness of the layer rock (under the layer sand), so we keep the same matter which makes more sense. However, the rock that falls goes under the sand next to him - if there is sand -, which may seem strange.

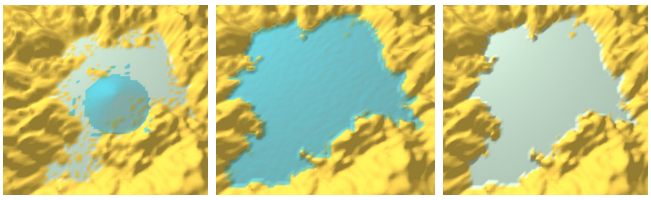
3 Hydraulic Erosion

3.1 Flows

We implemented the first part of the hydraulic erosion algorithm from [2]. It is made of fluid simulation called the pipe model where each point is connected to its four neighbors thanks to virtual pipes. These pipes allow the water to flow between point. The flow is updated according to the difference of height between the point which creates a different of pressure.
We got two problems linked to the clamping. We used unsigned int for the coordinates x and y, so we had issues when they became negative. Then, when the flow computed was out of range of the map, we returned the same closest flow which leads to an infinite increase of water. So, we saw that we had to return a null flow when the indices were outside of the box, which seems logical as there is no water there.

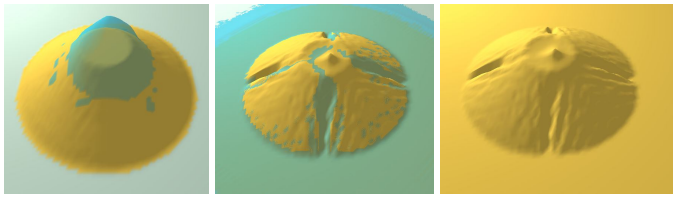
3.2 Erosion, Sediments Deposition and Sediment Transportation

Then, we implemented the second part of the algorithm which is "Erosion, deposition and Sediment transportation". The model is made with a coefficient C. If C is high, the water is saturated in sediment and so, the water drops some sediment. If C is low, the water takes more sediments. The first try showed a total divergence of the border which was due to the C coefficient. We looked at references and we found that it was better to multiply C by the height of the water clamped between 0 and 1. For the tilt angle, we used a function called getGradient that we created before, and then we could compute the sinus of the tilt angle. We also added an offset to the sinus and the velocity to avoid some 0 issues.



(a) Original (b) Water spreading (c) Terrain eroded

Figure 5: Results of hydraulic erosion



(a) Original (b) Water spreading (c) Sediments moved

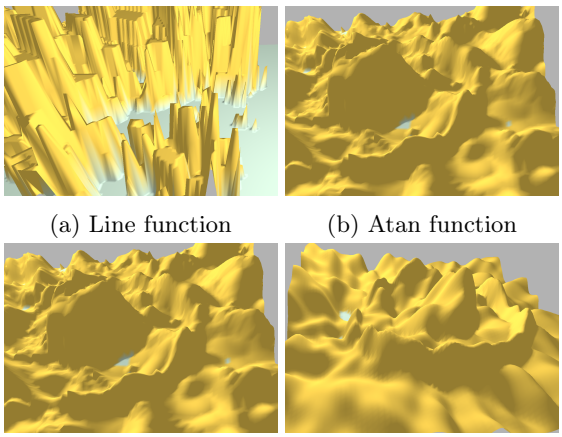
Figure 6: Deposition of sediments

4 Fault algorithm

In the paper [1], we discover what the fault algorithm was and we wanted to implement it. We found a way to determine a random line crossing the whole rectangle mesh. Then, we just added a constant positive or negative, according to the side of the line, displacement.

After the algorithm was working, we could try more specific and adequate functions for different results. We chose a displacement varying along the iteration. We also chose the atan, sin, cos for a smoother jump. The results were great. After the line, we wanted to try to add the fault algorithm circled version. Instead of the line, we added circles on the mesh. Of course, the height and the radius of the circles are random.

With the first version with a constant displacement, the map created was really rough and we could see the "cuts" even with lots of iterations. The iterated displacement solved that problem. The cos and sin functions make bouncy terrains. And the atan function is useful for smoother terrains but with a lot of level. The circle version is great for bossy and tricky terrains. We can also play with the number of iterations and we mixed the different functions to create a myriad of random maps. Thanks to the interface, we could also play with the parameters and apply the algorithm on pre-created maps. Results are shown in Figure 7.



(a) Line function (b) Atan function

(c) Circle function (d) 1-exp(-x^2) function

Figure 7: Results of fault algorithm

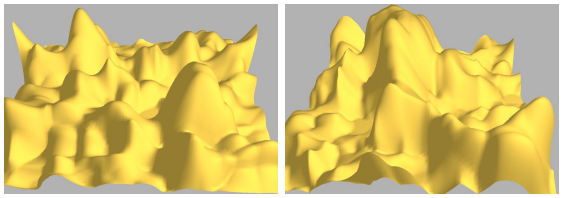
5 Noise algorithm

Afterwards, we implemented a new algorithm for terrain generation, from noise instead of faults. The first idea was to add random noise and of course, the results were bad. So, we implemented Perlin noise to get smooth transitions inside the terrain (see Figure 8).

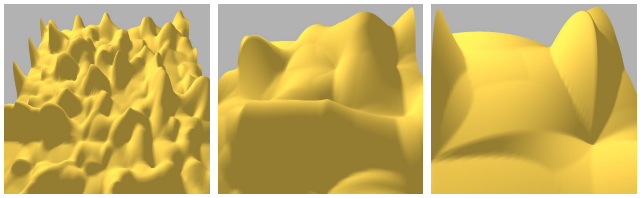
The first step was to create a grid of random gradients of norm one (we just picked vectors on the unit circle) and the size of the grid was the size of our mesh grid + 1 for width and height (so as to have a vertex of our mesh in each cell). Then, we computed the dot products between

gradients of the edges of the cell around the vertex with the offsets. Finally, we performed a bilinear interpolation on the grid with the 4 dot products obtained to obtain the smoothness. To have even smoother and real results, the user can choose the resolution in each dimension (width and height), which just brings closer the vertices before getting the result of the interpolation (we divide x coordinate by the resolution in X and y coordinate by the resolution in Y). It's like changing the size of the cell to apply noise, which is useful to create mountains first with big cells and details with small cells.

To get more flexibility on the terrain generation, we also implemented Fractional Brownian Motion (FBM). This creates an interval between Perlin noise and random noise, creating more real terrains. It works by summing Perlin noises at each octave, with different frequencies and amplitudes, depending on the attenuation and the lacunarity (the user can choose all of these number).



(a) Terrain generated with Perlin noise (b) Terrain generated with FBM



(c) Perlin noise, resolution = 1x1 (d) Perlin noise, resolution = 3x3 (e) Perlin noise, resolution = 7x7

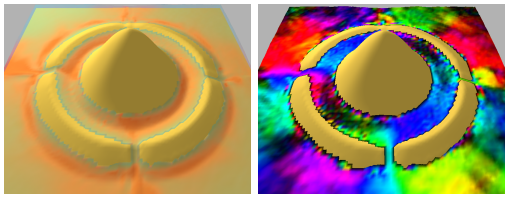
Figure 8: Results of Perlin noise and FBM

6 Implementation on GPU and performances

To improve the performance, we decided to implement the different algorithms in GPU, with the compute shaders of OpenGL. The GPU project allows the visualisation of more information. The user can choose to display the velocity field of the water - represented in HSV color space : the direction giving the hue and the norm giving the value -, and the sediments in the water - the more the water is concentrated in sediments the redder it is.

6.1 Thermal Erosion on GPU

Only the strategy B in 8 connexity was implemented. As explained above the idea with this strategy is to first compute the amount of matter that each point give to their neighbors and then to update the thickness of each point. The GPU implementation follows the same logic, however



(a) Sediment displayed (b) Velocity displayed

Figure 9: Visualisations offered by the GPU project

it has to be adapted to work with the way openGL organized threads which is in workgroups. Therefore, to be able to update the height of each point in a workgroup, it is needed to compute the amount of matter lost by each point in the workgroup plus a border of one, as some neighbors of the point inside the workgroup live in another workgroup. A barrier is needed to ensure the compuations of the flows at each points of the workgroup plus its border, before the updates of the heights.

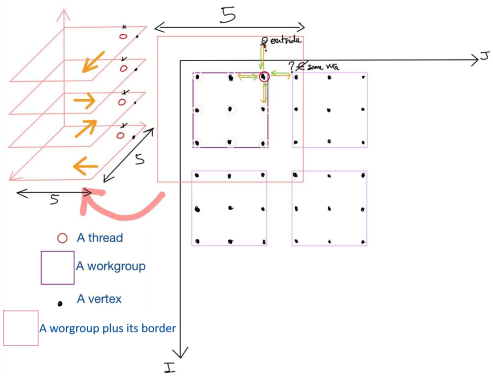


Figure 10: Organization of the threads - Connexity 4

6.2 Hydraulic Erosion on GPU

Hydraulic Erosion is computed thanks to two compute shaders. The first one, takes care, for each point, of computing the water flows, the water velocity, the eventual amount of material eroded and the eventual amount of sediment deposited. This shader is just a GPU adaptation of the algorithm written for the CPU. The adaptation is similar to the one for thermal erosion, as for both there is an idea of flow between points.

Moreover, we observed a strange behavior of sediment being deposed on places without water. The explanation was that even if there were no water in these points, there was still sediment as the rate at which the sediments are leaving the water is quite slow. To solve this unwanted behavior, a simple solution is to deposed all the sediment when there is no longer water at a point.

The second compute shader, takes care of the transportation of the sediments. The article [2] proposed a "backward" implementation of the transportation, meaning that the amount of sediment in a point (i, j) is replaced by the amount of sediment located in the point - not necessary on the grid - pointed by the opposite of the

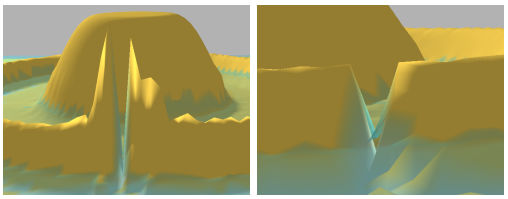
displacement associated to dt : $\overrightarrow{V_{(i,j)}}dt$ with $\overrightarrow{V_{(i,j)}}$ the velocity at the point. If the point is not on the grid a bilinear interpolation is done.

This implementation works fine but we were concerned about a possible loss of sediment, if a point of water was not pointed by any of its neighboring velocities. Hence, we proposed a "forward" implementation of the transportation, meaning that the sediment in a point (i, j) is transported and distributed to its neighbors according to the displacement made during dt at (i, j) : $\overrightarrow{V_{(i,j)}}dt$. The distribution is done between the four points surrounding the one pointed by the displacement (x, y) and accordingly to proximity between the four point and (x, y).

This was implemented on GPU with a different approach from the one using shared variables. Indeed, we decided, to implement it by computing for each point the amount of sediments he receives from each of its neighbors.

For both implementation dt should be small enough that the displacement $\overrightarrow{V_{(i,j)}}dt$ is not bigger than the length of the diagonal of a cell. Otherwise, the displacement would point to a point further than the one ring of point surrounded (i, j), giving an impression of teleportation.

The "backward" strategy gives more realistic results. With the "forward" strategy spikes appear when the water digs a ditch in the sand. Indeed, if the velocity is not perfectly straights along the ditch, some sediments would be deposited on points surrounding the ditch. However, by turning thermal erosion on at the same time, the spikes are eroded and they do not appear. More generally, thermal erosion has a regularization effect on the terrain. A general issue with the "forward" strategy is that sediments can be transported in points without water, which is not realistic.



(a) "Forward" (b) "Backward"

Figure 11: Spikes created or not, depending on strategy

6.3 Performances

The performances were measured on heightmaps represented the same landscape but with different resolutions and on a computer with a Intel i7-10750H and a NVIDIA GeForce RTX 2060. The CPU is no longer usable at a resolution of 500 x 500, the performances dropping at 1 fps, whereas the GPU implementation still gives smooth results at a resolution of 1500 x 1500 (Figure 12).

	Grid Size	Thermal Erosion (ms)	Hydraulic Erosion (ms)
CPU	100x100	10 ± 3	20 ± 2
	500x500	183 ± 6	486 ± 7
GPU	100x100	0.111 ± 0.008	0.122 ± 0.007
	500x500	0.36 ± 0.03	0.47 ± 0.04
	1000x1000	1.54 ± 0.06	2.02 ± 0.08
	1500x1500	3.1 ± 0.2	3.9 ± 0.4

	Computing For Rendering (ms)		Rendering (ms)		FPS
	Terrain	Water	Terrain	Water	
CPU	5.1 ± 0.5	7.1 ± 0.4	0.3 ± 0.1	0.3 ± 0.1	20
	125 ± 4	181 ± 4	2.5 ± 0.3	2.1 ± 0.1	1
GPU	0.041 ± 0.003	0.046 ± 0.003	0.073 ± 0.007	011 ± 0.01	240
	0.097 ± 0.007	0.12 ± 0.02	0.16 ± 0.02	0.16 ± 0.03	240
	0.38 ± 0.01	0.48 ± 0.02	0.48 ± 0.01	0.47 ± 0.02	115
	0.9 ± 0.2	1.0 ± 0.2	1.0 ± 0.2	1.0 ± 0.2	60

Figure 12: Performance table

7 Link to the git repository

https://github.com/Toratom/digitalTerrainModeling

References

[1] Eric Galin, Eric Guérin, Adrien Peytavie, Guillaume Cordonnier, Marie-Paule Cani, et al.. A Review of Digital Terrain Modeling. Computer Graphics Forum, Wiley, 2019, 38 (2), pp.553-577. 10.1111/cgf.13657. hal-02097510

[2] Xing Mei, Philippe Decaudin, Bao-Gang Hu. Fast Hydraulic Erosion Simulation and Visualization on GPU. PG '07 - 15th Pacifi c Conference on Computer Graphics and Applications, Oct 2007, Maui, United States. pp.47-56, 10.1109/PG.2007.15. inria-00402079

[3] Šťava, Ondřej and Beneš, Bedřich and Brisbin, Matthew and Křivánek, Jaroslav. Interactive Terrain Modeling Using Hydraulic Erosion