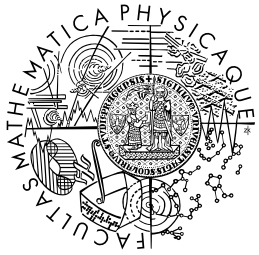


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Dario Lanza

Deriving Suitable Surface Shader And Displacement Map Information From Terrain Erosion Simulations

Department of Software and Computer Science Education

Supervisor of the master thesis:	Doc. Alexander Wilkie, Dr.
Study programme:	Computer Science
Study branch:	Software and Data Engineering

Prague 2020

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date .....

Dedicato a chiunque chi, nel corso degli anni, mi abbia ispirato, attraverso le loro storie e le loro vite, a intraprendere questo viaggio.

I would like to thank everyone who helped me in any way while writing this master's thesis.

I would like to thank my supervisor Alexander Wilkie for his suggestions and his calm and always positive attitude, it really helped me.

I would also like to thank my family and friends for their invaluable support during my studies, and for encouraging me to study abroad. I especially thank my parents, Giorgio and Cristina, for always believing in me and giving me hope for a better future.

Finally, I thank my girlfriend Marzia for supporting me while I was writing this thesis, and for proofreading it for me before submission.

Title: Deriving Suitable Surface Shader And Displacement Map Information From Terrain Erosion Simulations

Author: Dario Lanza

Department: Department of Software and Computer Science Education

Supervisor: Doc. Alexander Wilkie, Dr., Department of Software and Computer Science Education

Abstract: Realistic landscape models are often needed in the film and video game industry to create an immersive and believable world in which to settle a story. However, creating these vast scenes can be a long and complicated task. For this reason, many approaches have been suggested to automatize the process. One of the common techniques used is to simulate the various phenomena that lead to the creation of stunning scenes, such as the Grand Canyon. A downside of this approach is the lack of medium and smaller details, due to the comparatively low resolution of the simulation. What we are proposing in this thesis is to use the data internally processed by the simulation to drive the procedural generation of the medium and smaller details. To prove our point, we showcase the description of two phenomena: wind erosion and sedimentation using this new proposed approach.

Keywords: terrain rendering,erosion simulation,appearance modelling, procedural textures, digital terrain modelling

Contents

Introduction **5**

Context 5

Motivation 5

Research Questions 6

Thesis Outline 6

1 Background **9**

1.1 Shaders 9

1.2 Procedural Textures 9

1.3 Landscape and Procedural Textures 9

1.4 Geology background 10

1.4.1 Water Erosion 10

1.4.2 Wind Erosion 10

1.4.3 Sedimentation 10

1.4.4 Thermal Weathering 11

2 Related Work **13**

2.1 Procedural Generation 13

2.1.1 Subdivision scheme 14

2.1.2 Faulting Algorithm 14

2.1.3 Noise Algorithm 14

2.2 Simulation Modelling 15

2.2.1 Thermal Erosion 16

2.2.2 Hydraulic erosion 17

2.2.3 Wind Erosion 18

3 Framework **19**

3.1 Reading of Inputs 19

3.2 Classification and Processing of vertices 21

3.2.1 AClassifier and AShader classes 21

3.2.2 Shared Data Structure 23

3.3 Output Data 24

3.3.1 Vertex Data Approach 25

3.3.2 Point Cloud 25

3.3.3 Scene file 26

4 Simulation **29**

4.1 Wind Data 30

4.2 Multiple material sedimentation 32

4.2.1 Modified Sediment Transport 32

4.2.2 Modified Thermal Erosion 33

4.2.3 Sediment Data Structure Update 34

4.3 Minor Improvements 35

5 Data Exploit **37**

5.1 Wind Erosion 37

5.1.1 Stack of planes algorithm 38

5.1.2 DDA convolution and LIC algorithms 40

5.1.3 Our implementation 41

5.1.4 Flow shader 45

5.1.5 Parameters choice 46

5.2 Sedimentation layers 47

5.2.1 Nearest Point 48

5.2.2 Nearest point to surface 49

5.2.3 Minimization algorithm 50
5.2.4 Sedimentation Shader 52

6 Implementation **54**

6.1 Technologies 54
6.1.1 C++ 54
6.1.2 RenderMan 54
6.1.3 OpenMesh 54
6.1.4 PointCloudLibrary 54
6.1.5 Project 55
6.1.6 MathToolBox 55
6.1.7 FastNoise 55
6.1.8 Third Party software 55
6.2 Running the project 55

7 Evaluation **56**

7.1 Experiments 56
7.2 Wind Erosion 57
7.3 Sedimentation 62
7.3.1 Simulation 63
7.3.2 Visualization Algorithms 67
7.4 Combination of results 69
7.5 Performance 71
7.5.1 Framework 71

Conclusion **73**

Future Work 75
Wind Erosion 75

Sedimentation Layers 75

Bibliography **77**

List of Figures **80**

List of Tables **86**

Introduction

Context

This thesis is based on two correlated contexts: landscapes and procedural generation. Landscapes are used in movies and video games to create a plausible and immersive environment, sometimes by becoming a key point itself in the narration of the story. To give an example, the film "Avatar" strongly relies on the environment to tell its story. Depending on the artistic style of the game or movie, the landscapes can be less or more realistic, and the more realistic the landscape has to be, the more challenging is its creation. Since this task can be complicated and time-consuming, many approaches have been proposed in existing literature to automate the process.

Two main approaches can be used to generate a synthetic landscape. One method is to simulate the various types of weathering, e.g. erosion caused by glaciers and rivers, that create real eroded landscapes, like the Grand Canyon. The second way is through the use of procedural generation, which in the last two decades has received more and more attention. This also thanks to a constant increase in machine-power.

The increased popularity of software like Substance Designer, allowing the user to use the power of the procedural approach while at the same time allowing them to modify the texture locally, demonstrates that there is a trend toward this approach. Another field that has recently gained popularity is procedural modelling, i.e. the modelling of meshes through the use of an algorithm. The great advantage of using a procedural approach is the ability to obtain similar results, whether they are textures or meshes, through the use of a procedure that can be easily repeated. In this context, we would like to build a framework that takes advantage of this increasing interest in the procedural approach and apply it into the landscape generation context. Note that landscapes can be created using a procedural approach, the novelty of our research lies in the use of data, internally produced by simulation, to drive the procedural textures generation, which will be used later to shade the final mesh.

Motivation

The computational cost of a simulation is directly related to the resolution required by the user. Since the simulation by itself is already complex, the phenomena that are simulated are only the macro-scale ones, leaving the medium and small details up to the user. Most of the time, the output of the simulation is then used as input in the creation of the landscape, and is later modified by the user, who will have the task to correct and improve the geometry and create the adequate shader network for the scene, thus providing in this way the small

and medium details that the simulation was not able to produce.

The motivation for this research is simple: save time. With this framework, we aim to provide artists with some out-of-the-box textures, along with the results of a simulation process. With this approach, the artist will start with a scene that was not only automatically generated by simulation, but also automatically shaded. In some cases, this set-up will be just a starting point for the artist to create stunning scenes. In some other cases, where maybe the scene will be just used as background, the generated output with this approach will be enough without any further adjustment in the shaders network. Whether or not the output will be used as the final scene, we think that the time saved in the creation of the network shader is considerable.

There is also a second aspect that should be considered, this being precision. If the data from the simulation are correct and the algorithms that we use to simulate such data are physically plausible, then the results that we would obtain with this approach are more precise than the ones that an artist would obtain. In some scenarios this is not important, for example in films where a scene just has to be "believable". However, in some other fields such as scientific visualization, this is a more important aspect. With this approach, it would be possible to add more details in the visualization of data taken from space missions on planets in our solar system. Once we have images from the satellites and some extra data, like average wind direction, it would be possible to recreate a more plausible environment, given by the fact that the details are generated through some mathematical model.

Research Questions

The scope of this thesis is to investigate the possibility of using simulation data to automatically shade the geometry modified by the simulation. Instead of limiting us to use a simulation only to define the general shape of the landscape, we want to go further and try to use such simulation for creating details as well. To prove our point we will showcase two examples that we managed to achieve using this approach. During the creation of these examples, we had to answer some other questions like:

- which simulation data are needed to represent a particular phenomenon?
- how can we represent this phenomenon using textures?

We will also give a general description of a framework able to take input data from the simulation, process them and then finally output them.

Thesis Outline

The thesis is subdivided into the following chapters.

Chapter 1 is dedicated to giving the reader some basic background information that we believe is helpful for understanding this thesis.

Chapter 2 provides an overview of the works present in existing literature related to this thesis. In this chapter we also introduce the reader to the two families of simulation models that are widely used: a heightmap approach versus a voxel approach.

Chapter 3 is dedicated to the framework that we developed to prove our claim. In this chapter, we describe a general framework capable of handling different phenomena.

In Chapter 4, we discuss how the data from the simulation are calculated and the edits that had to be done.

Chapter 5 is dedicated to explaining more in detail how the data from the simulation are exploited to create the textures that are to be used at render time.

These last three chapters represent the core of our thesis and are summarized in Figure 1, which we will refer to throughout the entire write-up.

In Chapter 6 we briefly discuss the various technologies utilized for this thesis.

Chapter 7 covers the results obtained.

Finally, we conclude the thesis with a discussion on future works and possible applications.

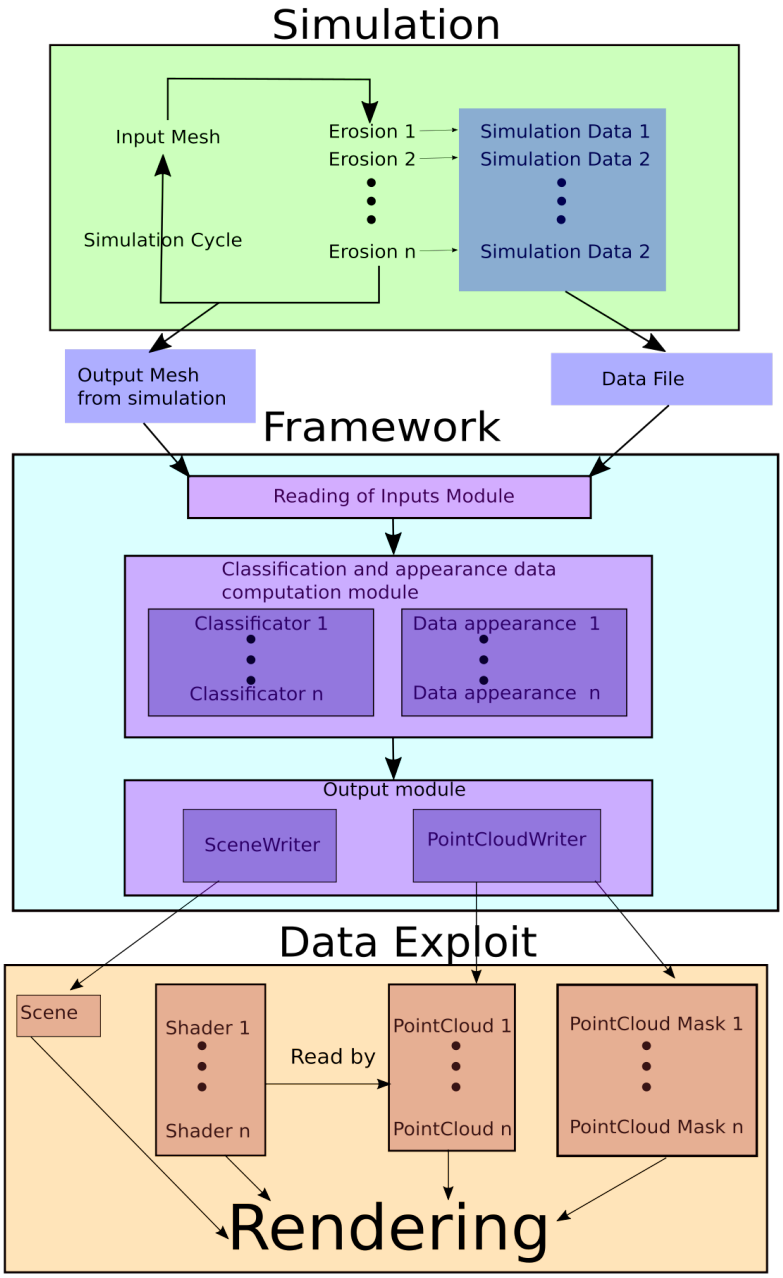


Figure 1: General flow chart of the entire process of the thesis. The three macro-blocks corresponds approximatevely to chapter 3,4,5.

1. Background

This chapter will give the reader some basic knowledge in the computer graphics field as well as the geology field. The reader will be introduced to the procedural textures and their link with landscapes.

1.1 Shaders

In computer graphics a shader is a small piece of code that grants the programmer more flexibility over the rendering pipeline. A shader can be used to achieve many interesting effects, such as post-processing effects, refinement of the mesh, and describing the appearance of a material. In the context of this thesis we will use this word to refer to the latter aspect. For us, a shader is a small piece of code that has the goal of describing the appearance of a point p. It is possible to build networks of shaders, using the output of a shader as input to another one.

1.2 Procedural Textures

A procedural texture is a texture obtained using an algorithm rather than drawn or painted. This approach has the great advantage of creating a texture with infinite resolution, instead of a raster image. Another great benefit of using such types of textures is the possibility of using random seeds that allow the user to create similar, yet at the same time different results. Since the output of these textures is evaluated at render time and more than once for the same spot, it is important to avoid long and complex algorithms that require a long evaluation time. All these elements suggest that procedural textures are usually best suited for the representation of elements with some repetitive pattern, such as woods, marble and bricks.

1.3 Landscape and Procedural Textures

As mentioned previously, procedural textures can be used to model some natural element. An example of this is the creation of landscapes through the use of the Fbm (FractalBrownianMotion). The idea behind this technique is that by adding noise with different frequencies and with different intensities, it is possible to recreate a plausible terrain. Noise functions with a low frequency should have a higher intensity then ones with a higher frequency. In this way, the low frequency noise models big shapes like mountains and valleys, while the high frequency noise generates the smaller details.

Another way to create a landscape using procedural techniques is to use a function, often called the density function, that for any point $P(x, y, z)$ gives one single float as a result. The result of the sign of the density function can be used to understand whether the point P is inside or outside of the material. If the result is positive then the point lies inside the ground, whereas if it is negative than the point is situated in an empty space, which can be air or water. When the result is zero, then P lies on the boundary between the material and the empty space. With this formulation we can implicitly define surfaces, which we can transform into meshes using the Marching Cubes algorithm. With this technique it is possible to generate more interesting landscapes since it naturally supports arches and hangover, due to the fact that the density function depends also on the z variable.

1.4 Geology background

In this section we will briefly explain the weathering phenomena that we will refer to later in the thesis.

1.4.1 Water Erosion

Water erosion is one of the most common causes of erosion. The constant and continuous hydraulic force leads some particles of the terrain to detach from it and be carried away by the liquid. With the passing of time, this simple process can lead to the creation of valleys and canyons.

1.4.2 Wind Erosion

Although similar to water erosion, the eolian erosion is less common and less visible. The principle is the same: particles of the terrain are detached due to the strength of the wind, and they are carried away. This is what causes dunes in deserts to move, but it is also the cause of some stunning landscapes as shown in Figure 1.1

1.4.3 Sedimentation

Sedimentation is a phenomenon related to the water erosion. The sediment present in the water, in fact, will be deposited in zones where the flow is calmer. When this process is re-iterated for a long period of time, the small particles of sediment and the end of the stacks get compacted by the weight of the particles at the top of the stack, thus creating layers by compression. Figure 1.2 depicts the sedimentation phenomenon.

Figure 1.1: National Park, Utah

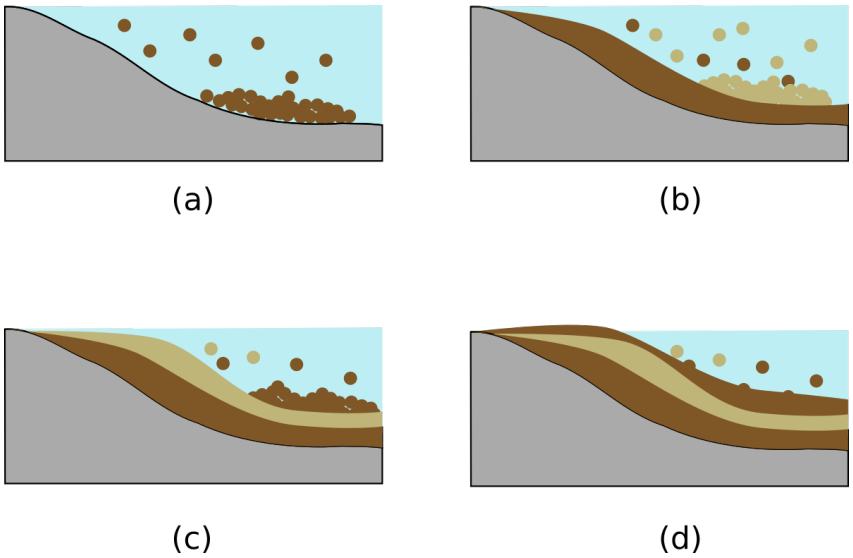


Figure 1.2: Example of a sedimentation process. (a) The particles of one material are transported by the flow and are deposited near the river banks where the flow is calmer. (b) Particles of a second material are now the main material transported by the flow and they are deposited over the first layer. (c) The same process in (b) happens again, leading to the creation of a new layer. (d) The third layer is completed, and a new layer is going to begin again.

1.4.4 Thermal Weathering

Thermal weathering, also known as thermal shock, is a phenomenon that leads to the breakage of bigger stones into smaller ones. During the day, the water, under the form of humidity, is able to filter inside the rock. At night, due to the lower temperature, the water turns into ice. Since the ice occupies a larger volume than water in its liquid state, the bigger volumes of the ice put a small pressure inside the rock. This small pressure creates a flaw inside the stone. After many

times the various flaws inside a rock create a network of flaws, this until one flaw breaks the rock in one or more pieces.

2. Related Work

In this chapter, we will mention and describe existing research in literature related to this thesis. For a complete and more exhaustive description in the field of terrain generation, we refer the reader to [15].

To store data relative to a landscape, usually one of two approaches is used: the elevation model or the volumetric model.

The elevation approach implicitly describes the landscape as a 2D function f in which the height z at a point of coordinate x, y is the result of the evaluation of $f(x, y)$. Most of the time, this approach translates into using a grid in which at each cell the height of the terrain at that position is saved. Then interpolation among the neighbours is used to approximate the real value of $f(x, y)$. This method has the advantage of being light in memory and fast to compute, and since it defines the mesh as a 2D function, is well suited for the application of textures and procedural approaches in 2D. A downside of this model is that it is not able to represent overhangs and caves. With a 2D function, we cannot obtain two values for the same x, y coordinates, which we would need to describe overhangs and caves.

The volumetric procedure overcomes this issue by subdividing the space of the scene into a 3D grid of voxels, such that in each voxel useful data can be stored. This technique is generally harder to use since it is weightier in memory and the surface reconstruction is also not a trivial task. Algorithms of surface reconstruction can be invoked to obtain the surface of the mesh, the most used one being the marching cubes algorithm. Although this representation shows these disadvantages, it is more general and allows the creation of overhangs and caves, in fact, the surface is reconstructed through the use of a 3D function.

There is a third approach that can be used: a hybrid approach proposed by Peytavie et al. [28], used to model and sculpt arches, caves and overhangs. The key idea is to have a discrete representation of the terrain, and an implicit representation of the surfaces, which is calculated as a convolution of the discrete model. The discrete model is a two-dimensional grid that stores material stacks. Stacks of material contain the thickness and the corresponding type of material. There are five types of material: air, water, bedrock, sand and rocks. By allowing the air to be a material type, this model naturally allows the representation of overhangs, arches and caves.

2.1 Procedural Generation

This family of algorithms relies on the fact that some landform shows some fractal properties, i.e. they are self-similar at different scales. This has been noticed by Mandelbrot [19], who proposed fractals as a possible way to model coasts.

This self-similarity can be exploited to define a general class of algorithms that iteratively apply a set of rules at each step of the algorithm.

2.1.1 Subdivision scheme

The class of algorithms known as subdivision schemes uses a subdivision algorithm to refine the mesh, such that after every refinement iteration, the newly generated points are randomly displaced along with their height. The strength on which this displacement is applied diminishes with the number of subdivision steps, i.e. the first refinement iterations will be able to influence with more strength the height of the vertices. Meanwhile, the last subdivision steps will have less influence over the displacement of the vertices, however, they will influence more vertices than the first iterations.

2.1.2 Faulting Algorithm

In [20], Mandelbrot introduces the so-called *faulting algorithm*, which will be later developed by [14]. Starting from an input terrain, the algorithm introduces details by adding randomly directed faults represented by a line: points above or below the line will be raised or lowered according to a distance function from the line. By re-iterating this process many times, some interesting results can be achieved. This algorithm can be extended to use other curves: recently in [34], Warszawski et al., made use of the poison faulting algorithm to create mesa and butte terrain. The idea is to use circles instead of lines to divide the space: points that lie inside a circle are raised according to a distance function from the faulted edge.

2.1.3 Noise Algorithm

The most common way to reproduce a general terrain is through the use of a technique based on the evaluation of a noise function n at point p. The results of $n(p)$ usually ranges from -1 to +1. In [14], Ebert et al. noticed that by adding several noises at different scales and amplitudes, it is possible to create a function that plausibly reproduces the terrain. This is very similar to the subdivision scheme explained before, and in fact, both of them are computing a fractal Brownian motion (fBm). However, the two methods differ because the first is refining and moving vertices, while the latter is displacing shading points. In order to have a coherent terrain, the noise function must be smooth: i.e. there must exist some spatial correlation between noise values. For this reason, most of the time a Perlin's noise function is used.

Using a smooth function does not allow the creation of sharp mountains and ridges, but this can be overcome using the so-called ridged noise introduced in [14]. The ridged noise function can be expressed as: $r(p) = 1 - |n(p)|$. Another

downside related to the use of the classic noise is the fact that every zone of the landscape present more or less the same pattern, which gives an unrealistic look to the scene. One solution to this problem was presented in [14]. In this book, the authors describe a new algorithm called *multi-fractal* for its property of showing different fractals. The modification proposed was to weigh with a function f the various intermediate results of the fbm.

Warping is another technique used to add some other variation to the landscape: by perturbing with a low-frequency noise the initial position of point p before evaluating $n(p)$, it is possible to roughly simulate some wind erosion. It is also possible to use fractals in three-dimension, using the marching cubes algorithm to visualize them, creating some interesting landscapes.

In conclusion, procedural generation algorithms are fast to compute but they tend to use a phenomenological approach, i.e. they try to reproduce an environment just by noticing some visually recurrent path, ignoring the various causes that lead to such paths. In many cases, some specific landscape elements cannot be easily modelled with a procedural approach, like sedimentation in zones where the water flows slowly. This is not always an issue: since they're not physically based it's possible to use them to create some alien or fantastic landscape, like the one in Figure 2.1, which has been created using a 3D fractal.

Figure 2.1: Image from the film: "Guardians of the Galaxy". The alien structure in the picture has been created using 3d fractals.

2.2 Simulation Modelling

The other widely used technique to generate a landscape is to run a simulation that approximates the various phenomena involved in landscapes creation. The next three subsections will be dedicated to a fast review of the various erosion processes that are usually simulated in literature.

2.2.1 Thermal Erosion

Thermal erosion was first introduced in [24]. The term refers to the thermal weathering that causes the breakage of bigger rocks into smaller stones, but in the end, most of the time it is used to define all those phenomena that lead to the creation of smaller stones from bigger ones and their slippage. The key idea this erosion is based on is that a stack of granular material tends to distribute, forming a certain angle known as the "talus angle" or "angle of repose". This angle is usually reported to vary from 30-45 degrees depending on the size of the granular objects. Starting from this idea it is possible to devise an algorithm. Let us define $s(x, y)$ as the slope of the terrain at x, y and α as the talus angle. At each step we evaluate $s(x, y) < tan(\alpha)$. If the equation is satisfied then nothing happens, because the slope of the terrain is not steeper than the maximum value allowed by $tan(\alpha)$. If the equation is false then the material is redistributed across the neighbour cells by a value proportional to $tan(\alpha) - s(x, y)$. Thermal erosion is, hence, well suited to reproduce taluses of mesas, as the authors in [7] did.

In [21], the authors propose an algorithm to reproduce thermal erosion in a parallel way as a relaxation problem implemented in GPU.

Other papers use a similar concept for different purposes: in [16] a concept similar to the angle of repose, called "sliding angle", is used. In this article, the authors propose a new method to visually simulate rock with joints. The idea is to create a voxelized box and randomly distribute joints to create empty spaces between voxels of material. Then a random number of rock voxels is selected and an analysis of the force that is applied at that point is done. In this analysis, the sliding angle, a concept similar to the angle of repose, is involved to reproduce the friction. If a voxel of rock is moved then a similar computation is done for nearby voxels. In this way, Ito et al. managed to create some visually interesting rock formations.

Another example of the thermal erosion algorithm can be found in the above-cited [13]. In it, the authors propose a framework that allows the user to modify the landscape using some special sculpting tools that emulate the effect of erosion. By sculpting the bedrock, i.e. by removing or adding rock material in an area of radius r centred in p, sand and rocks that were over it are affected. If some bedrock that was part of an overhang is removed, then smaller rocks should be created underneath it to simulate the erosion processes that were simulated by the brush stroke. If a pit is created, then sand and rocks nearby should naturally fall inside it. For this reason, after every iteration, a stabilization algorithm derived from the thermal erosion is internally invoked inside the discrete model to make sure that both rocks and sand materials are redistributed naturally. After the discrete model is updated, the implicit surface is finally recalculated.

2.2.2 Hydraulic erosion

This subsection is dedicated to the hydraulic erosion. The simulation of this type of erosion is what the majority of papers focus on. Many approaches have been proposed during the years, starting from Musgrave et al. in [23]. In this article, they devise a general algorithm that would have eroded sediment from some position and then released them in another place. Those ideas were later improved and expanded in [26],[9],[8], and [25] .
Later, an algorithm known as the "virtual pipes" model would be developed and implemented in [21] to simulate the flow of the water in a parallel way. Finally, in [30], the authors grouped all the research done so far into one unified framework.

Many of these models rely on a simplification of the Navier-Stokes equation, known as the shallow water, to allow an easier but still physically-based water description. The assumption of this model is that the strength of horizontal flow is much greater than the vertical one, which is well-suited to represent the fast flow of a river, but less for lakes or seas where the internal flows can play a non-negligible role in the movement of the water. Furthermore, all these models present the general idea that the flow of water causes the release of small particles of terrain under the form of sediment. The sediment present in the water is then transported and released in some other part of the scene where the flow is calmer. This process is described through the idea that the quantity of water present in a cell $w(x, y)$ is able to transport a certain quantity of sediment. In [21] this is called *sediment transport capacity* and defined as

$$S_k^m(x, y) = ||v(x, y)||C_k sin(\alpha(x, y))$$

where $v(x, y)$ is the water velocity, C_k is the sediment constant capacity and $\alpha(x, y)$ is the tilt angle of the terrain. This equation gives a maximum of the sediment that the water can transport for each cell. The value of S_k^m is then compared, at each iteration, with the sediment present at each cell $S^a(x, y)$. If $S^a(x, y) > S_k^m$ then the water is saturated with sediment, hence a fraction of it will be released and will become part of the bedrock. If this is not true, then it means that there is still room for some suspended sediment, thus, some part of the terrain will be eroded and it will increase the value of $S^a(x, y)$. After this step has been applied to each cell, the transport of sediment will be calculated. One of the basic solutions is to use a Lagrangian advection scheme, which has been further improved in [30] through the use of a semi-Lagrangian MacCormack method, described in [29], for the advection of particles. As the authors in [29] demonstrated, the use of the semi-Lagrangian MacCormack scheme is a good compromise between more precise and stable results and the number of steps needed to calculate these results.

The hydraulic erosion is also used in [13] as one of the possible phenomena that can happen. In it, Cordonnier et al. present a new approach for modelling landscapes which combines the classical erosion simulation with an eco-system one. The result is a simulation driven by the influence of the eco-system over the terrain. The landscape is divided into tiles, and in each tile, a discrete layer

representation is used to track the amount of sediment, vegetation and rocks present. The simulation is driven by events that can randomly happen on each tile and that can propagate to neighbour cells. Those events are :

- water erosion
- slides
- fire
- lighting

The idea is that each event consumes either vegetation or bedrock, and at the same time the quantity of vegetation and rocks can dampen some events. For example, the quantity of rocks that fall from a slide is reduced by the quantity of vegetation present, at the same time the event destroys part of the vegetation as well.

2.2.3 Wind Erosion

Although most literature is focused on the thermal and hydraulic simulation, a small branch of the literature has been dedicated to that of wind, or Eolian, erosion. In [6] the authors try to replicate the weathering that leads to the creation of the goblin formations in the desert of Goblin Valley State Park, Utah. The algorithm they describe is based on the geological processes explained in [22]. Starting from this, Beardal et al. pointed out that weathering at each point was strongly influenced by two parameters: surface curvature and hardness of the point. To replicate a similar scenario to the one described by [22], Beardal et al. developed an algorithm to replicate such weathering: let the user define the rough shape of the rock to be weathered then voxelize the object, and for each voxel compute, over a certain radius, the air-rock ratio. At each step, the algorithm would have removed some material inside the voxel, based on the air-rock ratio and on the hardness of that particular voxel. When the amount of material in a voxel reaches zero, the voxel will become an air voxel, exposing the voxel underneath it to the air. Later in [18], this idea was further expanded to allow cavernous weathering: using the same algorithm for estimating the surface curvature, the authors developed a generalization of the algorithm in [6]. With this approach, the authors devised an algorithm that would create smoother surfaces when the curvature is high, thus replicating the previous work of Beardal et al. When the curvature was negative, the surface would tend to create holes, replicating in this way the cavernous weathering.

In [32] Tychonievich and Jones propose to use a Delaunay triangularization to more efficiently divide the space. With this approach the authors managed to run both a hydraulic as well as an eolian erosion at the same time. The Delaunay triangularization allowed the author to easily approximate the surface curvature using the mean curvature of each vertex. With this hybrid approach the authors managed to obtain finer results than the one proposed in [6].

3. Framework

In this chapter we will describe the framework that we developed to prove that it is possible, and useful, to use simulation data to derive surface shaders.

During the creation of this structure, our goal was to have a minimal working system capable of handling the phenomena that we wanted to showcase and, at the same time, general enough so it can be easily expanded in the future.

As can be seen in Figure 1, our framework can be broken down into three general parts:

- reading of the simulation results, in which the two input files are read.
- classification and processing of vertices.
- writing of the final output, where a file of the scene to be rendered is created and two pointclouds for each simulated phenomena are generated. One pointcloud will be dedicated to storing the data computed in the "classification and processing of vertices" step. The second pointcloud, which will be derived automatically from the first, will be used as a mask. In this way we can quickly check, using the mask pointcloud, if the shaded point p is associated with the phenomenon that we are simulating and should read the data stored in the first map.

The above list outlines the abstract idea of our framework. We would now like to point out some minor differences between this idea and the actual implementation.

In our implementation, to solve the first point in the above list, we relied on the OpenMesh I/O module for the reading of the mesh, hence the format of the input mesh should be compatible with the ones allowed by OpenMesh. In our case, we used the widely used Obj format, to simply store vertices and faces.

The last module of our framework is probably the most abstract and generic: it has the task to create a scene file that can be rendered through some render engine and create a set of pointclouds that can be read. As we will explain in more detail later in this chapter, the system that we developed generates a scene file only for a specific render engine: RenderMan. The creation of the pointclouds is also aimed to work with RenderMan. Note from Figure 1 that during the rendering phase a list of shaders is expected. These shaders have the task of reading the pointclouds generated by our framework and use their data to compute the final appearance for the point p where they have been called.

3.1 Reading of Inputs

In this section, we will describe what are the expected inputs and how they will be parsed. The general idea is that we want to have, for each vertex in the mesh, some useful data from the simulation. These useful data can be anything, this

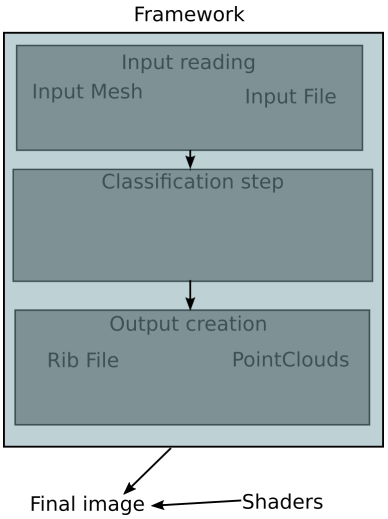


Figure 3.1: General flow chart of our framework

because we are defining a rather general framework that can be used for various situations. Therefore, in theory, we do not want to put any limit on the type of data that we want to receive as input. However, for now we are only supporting floats type. Even with this limitation, we believe that most of the situations can still be handled.

Our framework requires two files as input: a mesh file, in which vertices and faces of the mesh are defined, and a data file that contains the data obtained during the simulation process that led to the vertex being in the position in the end location that we are rendering it at. This file will contain n lines where n is the number of vertices in the mesh. In each line it will be possible to put all the data associated with only one vertex. We expect that the i-th line is associated with the i-th vertex in the description of the mesh. Once more, the general idea of the framework does not require any particular mesh type (trimesh or quadmesh), and in theory, every format that is able to encode a mesh should be sufficient. In practice though, we only tried our framework with the widely used Obj format. Each input data that the user wants to pass to the framework must be composed of a name and a floating value. In order to have a cleaner framework interface, each variable name is associated with a case of the enum class *SimulationDataEnum*.

This association is done using a map that will link the name of the input with the proper enum case. For this reason, the name of each input must be known by the framework a priori. If the user wants to add another input name that might be later used in the other modules, then it must create a new enumerator case that represents this new input data and associates the new case with its name. It is also possible to pass three floats, i.e. a vector, under the same variable name by following the name with a space and the character v. Lists are also supported: after the variable's name, the character l must be followed by the list's size and then the data.

A warning will be raised in the following conditions:

- If the name of the variable is unknown,
- If the size of the list does not match the number of received inputs,
- If less or more than three floats are passed to a vector
- If a name of a variable is declared but no data are given after this declaration

For now only four variables are used in our prototype and their names are :

- flow_normal, a vector.
- sediment_value, a list.
- material_stacks_height, a list.
- initial_sedimentation_point, a vector.

If no useful data has been elaborated for the i-th vertex, then the special token "void" is expected. This special sign is used to maintain the list aligned with the mesh file.

3.2 Classification and Processing of vertices

We will now discuss the main bulk of the project: the classification and processing of vertices. Our approach starts from the idea that a specific phenomenon happens only if some specific data are present: for example grass should only grow if there is water nearby. However, many interesting processes that take place in nature are due to more than one cause. For this reason, to understand which vertex shows a specific phenomenon is a non-trivial task. Furthermore, the most visually interesting among these processes are also the most complex to describe through a mathematical point of view. Therefore it might be a good idea to move the computational part to describe these phenomena before the rendering starts, and use only the results during the rendering. Once the correct vertices are saved and the data elaborated, we need to define a scheme that acts as an interface between the data elaborated and the output format in which we want to save the data. In the next subsection we will describe more in detail the two classes that handle these three cited problems:

- classification of vertices
- computation of appearance attribute
- saving of data

3.2.1 AClassifier and AShader classes

In this paragraph, we introduce the two main actors used in this process of classification, computation and savings of appearance attribute. First, we will describe the general approach for what they have been created for, and later we will briefly explain how their derived classes are used in our examples.

AClassifier is an abstract class that is in charge of the task of selecting the vertices interested by the particular phenomenon that we would like to represent and then processing the appearing data. Its output is a list of points, a single value called the *minimum density*, and a list of *appearance data*. These appearance data are the specific data needed by the associated shader to represent the phenomenon. Those data will be used at render time to represent the phenomenon. For this reason, the abstract class wrapping these appearance data is called *AShader*.

AShader is an abstract class that has the goal of storing the data, relative to one vertex, needed by the shader during the rendering. Furthermore, it also stores the confidence interval, a value between 0 and 1, used during the selection of points from the classifier. The AShader class also has the task of providing to the framework some useful information that will be later used in the creation of the scene file, for example the name of the shader and the list of its parameters that might be needed during the scene creation. Note that these parameters are somewhat global for the shader, i.e. every instantiation of the shader at render time will always have the same parameters. On the other hand, the data stored will change for every vertex. This dualism between data and parameter is the same present in OpenGL between uniform and varying data. Finally, the last task of the *AShader* abstract class is to give information about the data to the *PointCloudWriter* class, that is which type they are and in which order they will be saved in the point cloud. These two abstract classes will be derived for each phenomenon that we want to emulate. Each classifier will select a subset of the vertices of the mesh, process them, and then give the following three outputs:

- a scalar, called minimum density.
- a list of points.
- a list of *AShader* derived classes.

In particular, the *AShader* class will be used to store the data elaborated in the classification phase and the classifier's confidence in the data. The list of points that will be the result of the classifier does not necessarily have to match the points selected at the beginning of the classification phase: since the goal of this application is to create details, it might be very likely that some intermediate points would be needed to reach the desired density of details.

For this reason, *AClassifier* can, during its life cycle, create intermediate points and output them, provided that to each point the *AShader* class is associated. The framework also expects that the two lists are aligned: the i-th point will be associated with the i-th element in the data list.

All the classifiers that will be used are put in a list of *AClassifier*, then for each element in the list the abstract method *ClassifyVertices* will be called. As we will see later in this chapter, the order in which the classifiers are called matters.

In our implementation, since we want to simulate the wind erosion and the sedimentation, we have two classes that are derived from the *AClassifier* class - *FlowClassifier* and *SedimentationClassifier* - and two other classes derived from the *AShader* class - *FlowShaderParameters* and *SedimentationShaderParame-*

ters. The first two classes will have the goal of selecting the appropriate vertices and process to the associated data; the result will be stored in the two other classes *FlowShaderParameters* and *SedimentationShaderParameters*. These two last classes will be used to generate the point clouds needed by the associated patterns at render time. They will also be used in automatically creating the scene file. In the next section, we will explain more in detail how this is achieved.

3.2.2 Shared Data Structure

In this subsection, we will describe a shared data structure that can be used between classifiers. We thought that in some cases it would be useful to allow the possibility that some classifier would use the output of other classifiers. For example, if a classifier can detect river beds, this knowledge, inherent to the exact position of where river beds vertices are located, can be shared with other classifiers that work with river beds vertices as well, without any redundant work. In this way, classifiers written by some users can be used by others, similarly as some classes can use other classes in an object-oriented environment. The key idea is simple: store all the points outputted by each classifier in a kd-tree that can be accessed by any classifier. We will now proceed to explain the process more in detail.

Recall that every classifier will output a list of points and a separate list of data. The points are stored in a separate list from the data because they will be used to form a kd-tree, in this way we decouple the data from their position in space, leading to a lighter and faster data structure. In particular, the list of points will be added, through the method *SharedDataUpdate*, in a shared kd-tree that will be possible to pass during the classification phase of each *AClassifier*, derived class. By imposing that each vertex in the list is associated with one element in the list of data, it will be possible to recover the data associated with each point in the kd-tree using the same index. Since a classifier can create intermediate points and output them, there is an issue that should be handled: more classifiers might add many points in the same small space, slowing the kd-tree search and at the same time consuming memory space. To avoid this, we impose that every classifier also has to return the minimal distance that must exist between two points. If two points are found below this threshold, then they will be considered as a single point and will be merged together. This parameter is called the minimum density.

The update of this data structure is the following: after a classifier finishes its computation, the list of points created as output by the classifier will be queried over the shared kd-tree. If the distance between the point in the list and the closest point in the kd tree is superior to the minimum density value, then the point is added to a special list of points that will be added later to the data structure, otherwise it is rejected. Note that it would be better to add the point to the kd-tree at every iteration, since some points inside the list outputted by the classifier might have a distance below such threshold. However, this would require the re-evaluation of the entire data structure, slowing the entire process. This could be alleviated if we would have used some data structure similar to the

one described in [10]. Note that since we are allowing to merge two points, it might happen that a single point can store data used to represent more than one phenomena: for example, some classifier can handle the colour of a region, while some other classifier can change the displacement. Therefore the list associated with the shared kd-tree is a list of lists: each point inside the kd-tree will have a list of shaders data associated to it.

The other classifiers will be able to use such a kd-tree and lists of data for their purposes, provided that the user knows how to handle the data stored in the other *AShader* classes. After each classifier has been called, the framework enters its last module.

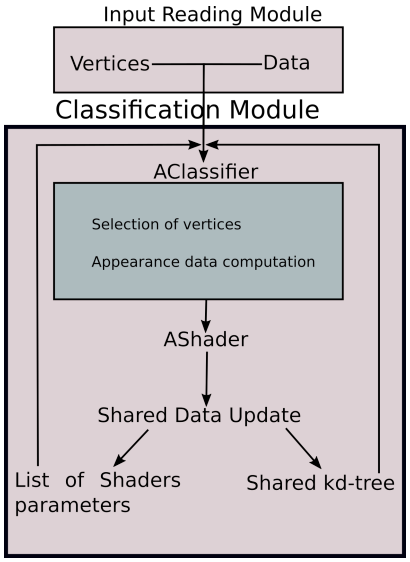


Figure 3.2: Flow chart of the classification module. Vertices and data are read, then each classifier uses these inputs and the shared data to select the appropriate vertices. The data computed that are stored through the abstract class *AShader*. The outputs of the classifications phase are then used to update the shared data. The cycle goes on until all the classifiers are used.

3.3 Output Data

After the classification step is completed, our prototype framework outputs a series of files: two pointclouds, for each classifier, and a file that contains the scene. As stated in the general description of this chapter, we are expecting that each classifier is associated with a shader that will be present during the rendering. The shader is in charge of the task of reading the pointcloud with the appearance data computed by the associated classifier.

In the next sections, we will explain two different strategies that we tried in order to pass the data from the classification module to the shaders at render time.

3.3.1 Vertex Data Approach

One of the first approaches that we tried was to save the data elaborated in the previous step in each vertex. We explored this possibility because it came natural to think that all the data received at each vertex should have been also saved in the same place. We also believed that it would have been faster at render time, since all the information was stored in points, avoiding the lookup in a 3d texture. However, in our implementation that uses RenderMan to generate the final images, this approach is inefficient for two reasons: firstly, it was memory consuming, and secondly, the data would have always been interpolated.

The approach is memory inefficient because in RenderMan, the data type must be the same for every vertex. This means that if a shader, s_1, needs to use a vector as input, and another shader, s_2, needs to use one vector and two floats, then every vertex in the scene must have saved a vertex and two floats, even if the vertex would have been selected by the classifier associated with s_1. This behaviour cannot easily scale with an increasing number of shaders. The second reason that led us to discard this approach was due to interpolation: the data present in each vertex are automatically interpolated among secondary points generated after a refinement step. Although sometimes this might be desirable, there are certain situations in which it is not. For example, if the parameter that we want to pass to the shader is an index, then interpolating can be dangerous: if the interpolation is between two numbers that are not sequential then the interpolation might create some middle value that was not present in the first instance. For these two reasons, we decided to use a point cloud approach, in which all the data needed by one shader at render time will be saved to a dedicated point cloud. In this way the memory use is more efficient: each point present in the point cloud has data that are used only by that shader, instead of carrying data that might be used by other shaders. Furthermore, it is now possible to decide whether or not to interpolate, by choosing how the interpolation is done in the function call that is used to read the point cloud. As a side effect, there is also the possibility to convert one point cloud format to another.

3.3.2 Point Cloud

In this subsection, we will explain the point cloud approach that we used to save the data computed in the classification step. The key idea is to create two pointclouds: one dedicated to storing the data, and another which is used as a mask to blend the result of other classifiers.

As explained before, after all the processing has been done, the shared kd-tree and the associated list of data will be used to create two pointclouds for each classifier that has outputted more than one point. Note that some classifiers might not have produced any output because the scene doesn't require it: for example if the landscape is a pluvial forest we would expect that a classifier with the goal of simulating the desert would not select any point in the scene.

We are expecting that at render time, each classifier has an associated shader that handles the data stored in the appropriate *AShader* derived class. In our case, for example, the shader *FlowShader* handles the data stored in the class *FlowShader-Parameters*, derived from the *AShader* class. Those data are generated through the class *FlowClassifier*, derived, once more, from *AClassifier*. These shaders are called during the rendering of the scene and have the purpose of reading the data stored in the dedicated point cloud, if necessary do some further processing, and then finally output the result. However, a generic shaded point p doesn't know to which shader it should be associated. One idea to obtain this information would be to query each point cloud: if p is close enough to a point present in the i-th point-cloud, then p will fetch the data from it and the i-th shader will be used to shade the vertex.

The point cloud maps associated with each classifier can be large if the data needed for the shader are numerous. Therefore it can be demanding loading a point-cloud in memory just to understand that p is not close enough to any point inside it. To solve this issue we propose to create, for each classifier, another point-cloud that acts as a mask. At first, we thought to create such a mask simply by using, for each classifier, all the points outputted by it and store their position in the point cloud.
During the evaluation of the shader, the point p to be shaded is queried over the point clouds. If p would have been found in a certain radius, r, to some point belonging to the i-th point cloud, then p would have been associated with the i-th shader. This procedure has the disadvantage of creating a harsh mask: the points are always either outside or inside the point clouds without the possibility to create a smooth transition between where a material ends and where another one begins. Using the confidence value stored in the *AShader* class solved this problem: with it, it is possible to create a point cloud map that acts as a mask, furthermore since the confidence lies between zero and one we can directly use it to grant a fading effect. Therefore for each class derived from the *AShader* class two pointclouds are created: a mask pointcloud with only one float value between 0 and 1, and another pointcloud with the data needed for each associated shader at render time. These two pointclouds will be created through the use of the abstract class *PointCloudWriter*. In our project, we only derived one class from it, the *RIBPointCloudWriter*, which is a class with the purpose of writing a PointCloud for the Renderman environment.

3.3.3 Scene file

As already stated before, the framework also automatically generates a scene file. This is imposed through the use of the abstract class *SceneFileWriter*.

Since this is a task that can vary substantially depending on which render engine is used, it is left rather general. The most important part of the scene creation is the generation of a network of shaders that uses the outputs of the classifiers. This is the reason why the scene creation should be handled by the framework: since its duty is to ensure that the generation of the shader network reflects the

order of the classifiers. Otherwise we would output only the pointclouds with the data for the shaders without knowing how to combine those shaders.

In this project, the class that has this task is the *RIBFileWriter* class, derived from *SceneFileWriter*. The rest of this subsection will be dedicated to the description of the creation of the RIB file, and the procedure used to generate the shader network.

The scene generated from the *RIBFileWriter* class is rather simple: with just the input mesh, a light in a fixed position, and some default parameters that can be easily overloaded at render time like number of samples or resolution. During the development phase, we realized that it would have been useful to have the possibility to change where the RenderMan's camera was facing. For this reason, we took part of the code used to simulate the camera in the simulation project and adapted it for our purposes. The result is a preview window that allows moving through the scene. When the navigation is finished the user can exit from it and the camera projection used to visualize the scene will be written down in the RIB file, i.e. the rib scene will be, approximately, rendered from that same view. In this way, during the developing phase we avoided the overhead time due to opening some other 3rd party program like Maya or Blender, load the RIB file, visualize the scene, move the camera, and then render the scene.

Our framework automatically writes a basic material network that is applied to the mesh. The appearance of the material is influenced by the various phenomena that were selected in the previous phase. For now, only the base color and the displacement value can influence the appearance of the material. In future it might be interesting to modify some other parameters, for example, the reflectance color and roughness or some other parameter related to the subsurface section.

Each classifier has a pointer to its related shader, and during this final phase these shaders pointers are collected and stored in a list, creating the "Shaders_list". Each element in the list represents a shader that will be used at render time, therefore the framework will proceed to write the name of the appropriate pattern in the RIB file.

After all the elements have been "instantiated" in the RIB file the framework starts to build a basic material tree using the elements in "Shaders_list".

The idea is to create two trees of shaders that are used to influence the two aforementioned inputs in the Material node: the "diffuseColor" and the "dispScalar".

In the RenderMan framework, the "mix" pattern is a node used to combine two colours using the third value to determine the weight of such a combination.

The generation of the tree for the "diffuseColor" section is the following: the first element in the "Shaders_list" is combined, through the use of a "mix" pattern, with a shader that has a constant color. As explained before, each Shader has a Mask pointcloud associated with it. Through the use of a pattern that we created, called "MaskReader", the mask associated with the first element in the list is read

and used to weight the blending of the two shaders. At the second iteration, the second element, $Shader_2$, in the list is selected and mixed with the result of the previous iteration, again using the pointcloud mask associated with $Shader_2$ as weight. This is done until the last element of the list has been processed. After this, the last result of the mix operation is used as input in the "DiffuseColor" section of the base material. Note how the order in the "Shaders_list" influences the final output: since the shaders are essentially overlayed one on top of each other, the topmost layer will have the highest priority. This means that if the same vertex v has been selected by two classifiers, c_1 and c_2, and the data elaborated by c_2 comes after the data elaborated in c_1 in the "Shaders_list", then at render time the color at point p will be obtained by consulting the shader associated with c_2.

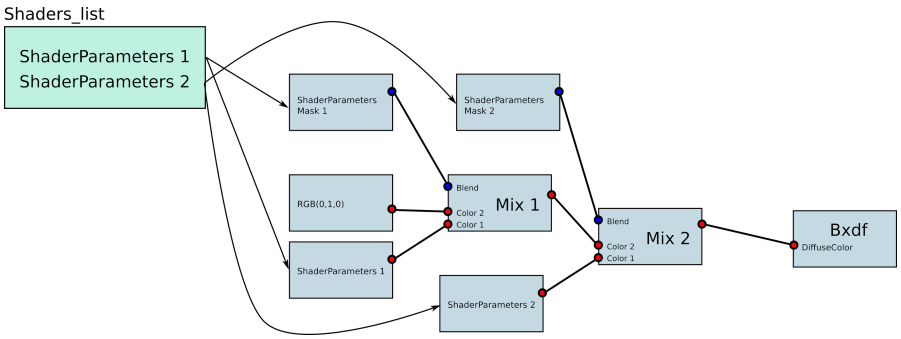


Figure 3.3: Example of a shader network generated automatically. The shaders presented in the list are used to instantiate two blocks: one for the shader and one for the mask. The shaders are then combined using the mask to weigh the blending between the various nodes. This is done until the last element in the list is added to the network. A similar scheme is used to calculate the displacement height.

A similar process to the one described for the color is done for the displacement section but with a difference: since the value that is used as input in the "DisplScalar" section is a float, it is now possible to blend the result of different shaders using another type of operator, the "Add" operator, which, as the name suggests, simply adds the values of two nodes. With the "Add" pattern it is now possible to combine the result of different shaders instead of just overlaying them. This can sometimes be desired while some other times not, so for this reason it is possible to set the *BlendMode* flag inside the derived classes to choose whether to add the value or not. The value stored in the *BlendMode* will only be used to determine how to combine the two shaders only in the tree dedicated to the displacement. The framework also assumes that each shader is capable of creating two outputs: a float and a color. The float must be named *resultF* while the color must be called *resultRGB*. If this naming convention is not respected, the render will fail.

4. Simulation

This chapter will be dedicated to the description of the "Simulation" module that can be seen in figure 1.

Our framework needs two input files to work: a file containing the mesh and one with data relative to each vertex in the mesh. To obtain these two files we had to develop a system able to simulate and output data useful for our framework. Implementing a hydraulic erosion simulation by our own would have required a tremendous amount of time and would have been out of the scope of this thesis, therefore we looked for some already implemented project on Github. We chose to use the one at [5] for its simplicity and clearness. The project implements the transport sediment model described in [21, 17, 31]. Our goal here was to demonstrate that it is a valid idea to link simulation processes with shader-based appearance. Therefore we did not strive for perfect realism, but we focused more on phenomena that would have been otherwise hard to model.

As already stated the two phenomena that we wanted to model were eolian erosion and sedimentation. We selected these two phenomena because they create some visually interesting pattern in nature and at the same time they are not so trivial to represent. More specifically it is hard to model the wind erosion in a coherent and physically plausible way: clearly, a skilled artist can reproduce the macro geometry caused by the flow of the wind and of the water, but creating the smaller details can be really challenging. As far as we know there are two options, one of these being that the artist can rely on stretching some noise texture along a fixed direction, but in this case the direction of the flow is fixed and would not change throughout the scene. The other way is to sculpt or draw the details by hand, which can be a long and tedious task if it needs to be done for a big scene,for example a video-game map that can be explored by the player. Moreover, he or she would have to assume the direction of where the eroding agent was flowing, which sometimes can lead to incorrect results. With the first approach the texture is fixed along one direction, giving a slightly unrealistic look to the scene, while in the other a tremendous amount of time is used. Both cases, however, would not be able to ensure correctness.

We took the sedimentation phenomenon for a similar reason: it is possible to represent sedimentation layers with a procedural approach and the results that can be obtained with this technique are impressive, but only for local areas. Representing the sedimentation layers of an entire scene, maintaining at the same time the same local quality, can be rather demanding in terms of computation, because more and more functions are needed to be evaluated at render time. Furthermore there is no correlation between the macro-geometry of the mesh and the micro-geometry describe by the shader. For this reason we think that running a simulation able to reproduce the sedimentation process could lead to a better disposition of the sediment on the large scale. Once more this project is not striving for perfect realism but is more oriented so solutions that can be used as base for the creative process.

The rest of this chapter will be dedicated to the description of the modifications that had to be done to the initial project to collect and calculate data for our framework.

4.1 Wind Data

In this section, we will describe the edits that had to be done to collect useful data for our air erosion algorithm.

Starting from the idea that air, like any other gas, can be described as a fluid, we took the algorithm that was governing the flow of the water and we used a modified version of it to define the movement of air.

The shallow water model that was used in the water simulation was still a valid option for us, because in the scenario that we are interested in, the main direction of where the wind flows is the horizontal one.

At the same time, the flow of the air is also influenced by the topology of the terrain: for example in slot canyons, i.e. canyons that are much taller than wider, the wind is channelled through the entire canyon. Therefore we ideally wanted a simulation that could have been influenced by the beneath topology and at the same time been able to set a favourite flowing direction for the wind.

Our first idea was to apply the same algorithm present in the project to calculate the flow of the water and use it directly to derive the flow of the air as well. However, applying directly the water simulation to model the movement of the wind left us unsatisfied: the flow of the air, as the water, was completely driven by the topology of the underlying mesh.

In particular, we were interested in a scenario with the presence of canyons: we wanted that the air flew through the canyon, influenced by its geometry. However with this approach the air was simulating the behaviour of a river at the bottom of the canyon, leaving the vertices at the top without any useful information.

For this reason, we added some changes to the code of air simulation. In particular, referring to [9], the movement of the water is usually governed by the following equation:

$$\Delta P_{i,j}(x, y) = \rho g \Delta h_{i,j}(x, y).$$

where $\Delta P_{i,j}(x, y)$ is the pressure difference between cell x, y and its neighbour cell with coordinates $x + i, y + j$. ρ is the water density and g is the acceleration gravity. Finally, $\Delta h_{i,j}(x, y)$ is the difference in height between cell x, y and its neighbour cell $x + i, y + j$. We simply add in this equation an external force f leading to:

$$\Delta P_{i,j}(x, y) = \rho g \Delta h_{i,j}(x, y) + f.$$

where f represents the flow of the wind and its orientation and strength is constant over all the scene at a single frame. The force f is capable of influencing the pressure of the media over a preferred direction, hence favouring the flow of the media in some places that would not have been reached otherwise. At the same time, if the strength of f is comparable with the same order of power with the rest of the equation, the flow is not completely driven by it.

The direction of this force is then randomly changed during the simulation to reproduce the changing wind directions.

With this simple modification, we were able to collect data with enough coherency to be used by our framework to simulate our approximation of the wind erosion. At each iteration, in fact, the flow of the wind, calculated through the simulation of the air, is stored for each vertex position. Then when the simulation ends the average flow vector is calculated, normalized, and written in the file where all the data are saved. Note that the direction of the flow of the wind calculated at each vertex v is not the same as the external force f: if for example, the vertex v lies at the bottom of a canyon, and the air has a direction direction f, then the flow of the wind computed at v will probably follow the direction of the canyon and not the one imposed by f.

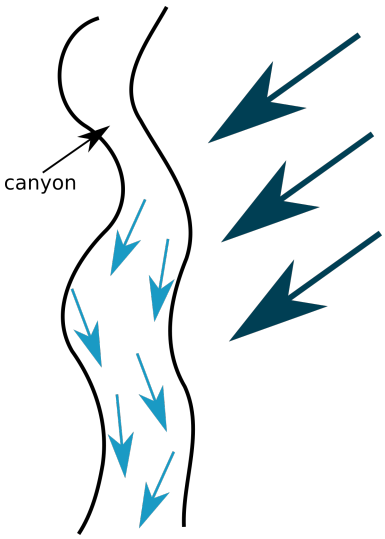


Figure 4.1: The big dark-blue arrows represent the direction of the wind imposed by f. The light-blue arrows represent the local flow of the wind influenced also by the geometry of the canyon.

The introduction of the force f proposed here has been taken from the description of the Shallow Water model provided in [12]. The solution that we proposed here is just an approximation of the wind simulation, but nevertheless, it managed to produce data with enough coherency to be used by our framework. After all, the goal of this edit was to collect data: if some other more sophisticated method is used to calculate such data, the final result will surely benefit from it. However, the approach we are proposing to use data from which derives surface shaders is valid in both cases.

4.2 Multiple material sedimentation

Our second challenge was to compute data that might have been useful for the second phenomenon that we wanted to model: a stack of sedimentation layers. One of the common approaches is to let the user define, through some GUI tool, a curve that describes the disposition of the various levels of sedimentation. After this step the simulation is started, and then the data provided by the user are just used to calculate the hardness of the rock. However, in this way we would have denied the erosion and the successive transport of sediments involved in the simulation. Therefore we went further and tried to elaborate an algorithm able to simulate the transport, and then the successive deposition, of multiple materials at the same time, instead of using the hardness data just to influence how much sediment should be eroded at a specific point.

To do so we had to edit the sediment transport and the thermal erosion algorithms. We also had to add two stacks of float to every vertex in the simulation, to be able to store the new data calculated through these two proposed methods. The first stack will store the various material ids that came in succession, creating a history of the various sediment layers. The second stack will store the height at which those sediment layers started. With these two stacks, we are able to maintain a history of the various sediment layers and from where they started during the simulation. We sided these two stacks with a pair of items that will be used to update the data structure. We will later describe how this pair is used.

We will now explain in details the new algorithms that we developed for this scenario and how the data are updated during the simulation.

4.2.1 Modified Sediment Transport

In this subsection, we will describe the modified sediment transport used to collect data for our framework. The problem that we are facing can be described in this way: in the simulation there is more than one type of sediment. Each type of sediment is represented by a unique material id. Our goal is to create a model able to transport these various type of sediments.

One of the common ways to simulate sediment transport is to use a Lagrangian approach. With this method, the property that a particle will show at time t is determined by looking at the property of the same particle at time $t-1$. In the usual implementation, this is used to calculate the quantity of sediment transported by a particle at time t. The position of the particle, p, can be approximated with $p(t-1) = p - v * dt$. Where v is the vector of the flow and dt is the discrete-time step. Since $p(t-1)$ will probably not lie exactly in one of the points of the simulation grid where the data of the simulation are stored, the quantity transported by $p(t-1)$ will be known through interpolation of the data present at the four corners of the cell where $p(t-1)$ will be. With this algorithm it is easy to know how much sediment is transported by a single particle, but it is not immediate to know exactly which material id has been transported. If for

example two corners of the cell have id_1 and the remaining two have id_2, which material id should be selected for the transport?. The approach that we used was to calculate how much a particular material id present at one of the corners of the cell contributes to the total sediment transported by the particle at time $t-1$. This is done by iterating over each corner in the cell, calculating the distance between p, the particle position at time $t-1$ and c_i, the i-th corner of the cell. The inverse distance is used to calculate how much the material id present at c_i influences the total sediment transported. This is done for each corner of the cell, then the contribution from corners with the same material id is summed up. After this step it would be already possible to select the material id that gave the highest contribution to the point, however in this way we would have partially denied the presence of the other material ids, therefore we went further and used a probabilistic approach: we used the contribution of the various materials id to create a small pdf, then select the final id to be transported by using a random number. With this approach, we were able to simulate the sediment transport of multiple materials, with the Lagrangian approach.

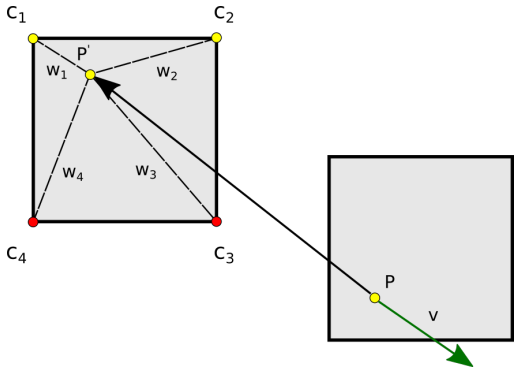


Figure 4.2: Modified sediment transport. In this example the material id at point p is selected by looking at p'. The position of p' is known by inverting the direction of v, the velocity of the fluid at p, and multiplying it by dt. Once p' is known, we calculate the weights w_1, w_2, w_3, w_4. Each weight is equal to the inverse distance between the corner and the point P'. Then we sum w_1 with w_2 and w_3 with w_4 since they have the same color id. Then we select a random number x between 0 and 1, if $x < w_1 + w_2$ then P will transfer the yellow material, otherwise the red. In this example the yellow material has been selected.

4.2.2 Modified Thermal Erosion

In order to simulate the Thermal Erosion, we developed a similar algorithm to the one described previously.

Usually, to represent thermal erosion a simple smoothing operation is applied for every point in the grid. In the implementation of the hydraulic erosion that we took, this is implemented through averaging of points. Once more this solution can handle the problem of how much sediment is moved from a point to another, but not which type of sediment will be present at the point after the transport

phase. We developed an algorithm similar to the one proposed above: we select the four closest cells and the central cell c for a total of 5 cells. Then for every cell, we select how much it contributes to the choice of the material id to be transported. This time the contribution is calculated by taking the height of the cell in question, and in this way the material id of higher cells, with respect to the selected cells, should have a higher chance of being selected. If a cell has a height less than h, then its contribution will not be used. In this step, in fact, we are trying to reproduce the sediment settling, that means a redistribution of material from a point to its lower neighbours. For this reason, the cell shorter than h will not affect the choice of the material id. If some cells have the same material id then their value will be summed up together. The contribution of every material id is used to build a pdf and select the material id to be transported through some random number between 0 and 1.

4.2.3 Sediment Data Structure Update

This subsection is dedicated to the description of the operations required to update and maintain coherently the dedicated data structure. This will essentially involve the removal and addition of sedimentation layers.

At every step of the simulation, each vertex $v(x, y)$ can either be subject to a sedimentation process, an erosion process or to no process at all. Let us define as z the height of the last vertex saved in the list of sedimentation points and z' as the new height calculated by the simulation at the coordinates x, y. We also define as id the material id of the last element saved in the list of sedimentation history and id', the material id that has just been calculated for the position x, y.

We can now explain more in detail how the pair of items that is saved along the two stacks is used in the data structure update. The first item of the pair, $ActiveMaterialId$, is used to store the material id of the last sediment that has been deposited on that point. The second element,$MaterialIdCounter$, is a counter which counts how many times the material id, stored in the first position, has been deposited on p.

Before adding a new layer of sediments some tests are run: firstly, we check $z < z' + t$. Note that this does not necessarily coincide with a sedimentation event in the simulation: this depends if the sediment deposited on the ground is enough to overcome the threshold t.

Second, lywe check that $id \neq id'$, or rather if the material id that has been deposited is different from the material id at the top of the stack.

If both conditions are true then we increase the value of $MaterialIdCounter$ by one, otherwise, we decrease it. If the value of $MaterialIdCounter$ is more than a certain threshold, $MaterialIdTreshold$, then z' is saved and added to the list of intermediate sedimentation points and id' is added to the list of sediments history, thus associating the height z' with the start of the sediment stack id'. If the value of $MaterialIdCounter$ reaches zero then the system assumes that

the material id stored in $ActiveMaterialId$ was a noise value and hence must be discarded. Therefore the new material id, id' overwrites the value stored in $ActiveMaterialId$.

A layer is removed whenever $z < z'$, i.e. the height z', is lower than the last height saved in the list of intermediate sedimentation points, z. This means that the ground is now at a height lower than the last point with material id id', thus the last layer is cancelled by removing the first head in both stacks.

Note that the last element in the sedimentation history is used as the material id for the position x, y. This means that if at the end of the update of the data structure a new layer with material id id_{new} is added, then at the next iteration, when the sediment transport algorithm will be called, id_{new} will be used as material id for the position x, y.

4.3 Minor Improvements

Some other minor improvements have been done: in this last section, we will briefly explain them.

First, we implemented the possibility to save the simulation mesh and internal data into two files: one for the mesh itself and another for the data.

We also developed some visual debugging tool that proved to be useful: one to visualize the airflow and another one to visualize the spread of the various sediment stacks.

Lastly, we also update the entire project to use a more recent release of OpenGL.

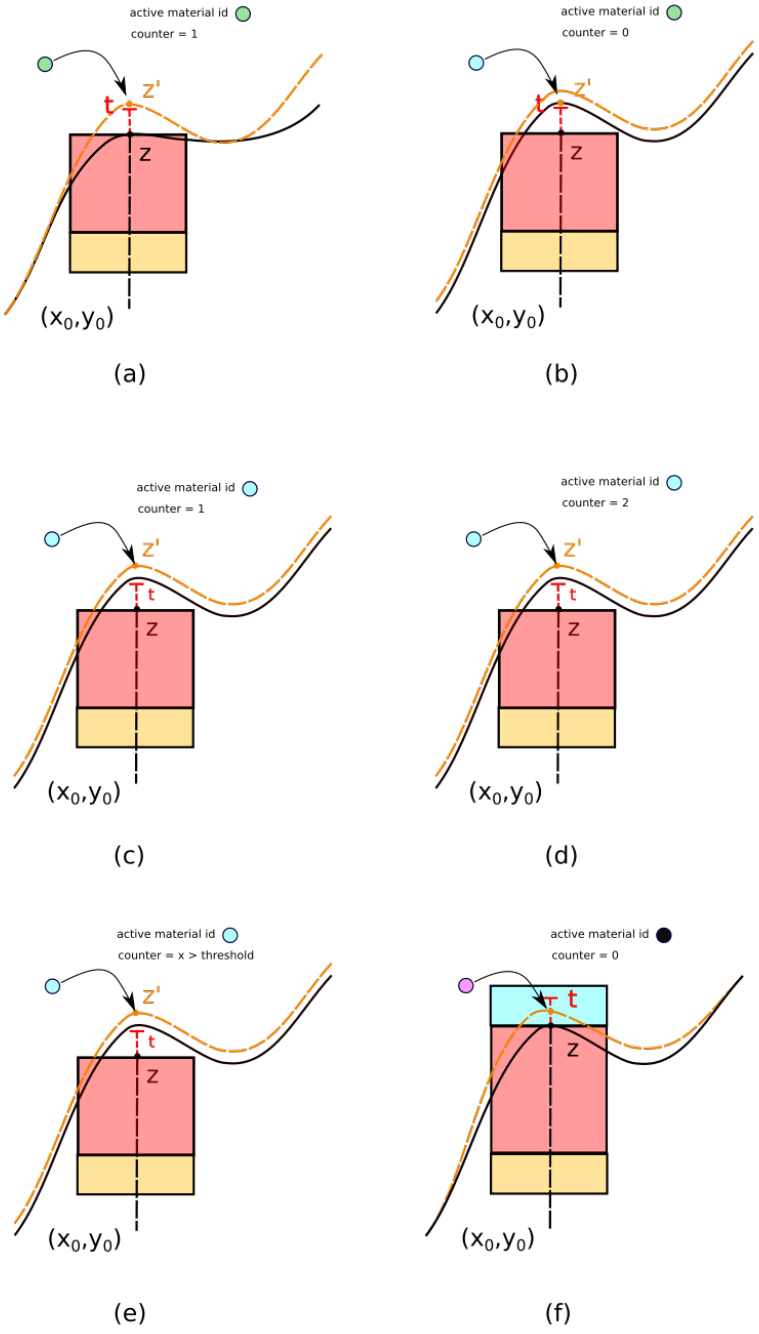


Figure 4.3: Example of a creation of a new sedimentation layer. (a) The green sediment deposits at x_0, y_0. Since $z' < z + t$ the green sediment is stored and its counter is increased by 1. (b) Another sediment comes, but it has a different material id, therefore the counter is decreased by 1. (c) Another particle come to position x_0, y_0, the counter was at zero so the new sediment is accepted as the new active sediment. (d) The counter is increased by 1 because a particle with the same id has arrived. (e) The situation in d is replicated until the counter is superior to a certain threshold. (f) A new layer is created with the id of the previous active layer, the counter is set to zero and the active id is set to a void value, ready to repeat the process for the new incoming particle.

5. Data Exploit

To show that our framework is valid we will use it to model two phenomena: wind erosion and sedimentation. In this chapter, we will describe more in detail how the data from the simulation are employed in our framework. For each phenomenon, we will discuss which data is expected from the simulation, and the various approaches used to process them. Finally, we will also explain how the data elaborated in our framework are used, through the use of the shaders, at render time.

5.1 Wind Erosion

In this section, we will describe the process involved in the representation of our first phenomenon: wind erosion. Although the shader that we developed is aimed to reproduce the stripes due to wind erosion, this shader could be used more generally to represent all the phenomena driven by the erosion of some fluid agent: for example, also some karst phenomena that are driven by the erosion of the water could be represented in this way.

Figure 5.1: (a) Parco Nazionale di Belluno ,(b) Triestinen Karst Both images are examples of a karst phenomenon knows as Rillenkarren. Karst phenomena happens when acidic rains corrode limestone or dolomite bedrock.

To simulate this type of phenomena, the shader is expecting only one data from the simulation: the normalized direction v of where the weathering agent was flowing at point p during the simulation. From now onwards, we will refer to this data using the term *normalized flow*.

5.1.1 Stack of planes algorithm

Our first approach was to reproduce the stripes caused by wind erosion through some continuous function $f(x, y, z)$. The core idea was to create stacks of lines that should have followed the flow direction at each shaded point p. This led to three sub-problems that had to be solved:

- create a line at p that would have followed the flow normal v
- create a stack of such curves with the possibility of increasing the density of the stack (i.e. having more lines in the same amount of space to create additional details).
- make sure that these stacks of curves are continuous between each other, i.e. the stacks of line produced at point p with flow normal$v$ would have been continued by the neighbour point p_1 with flow normal v_1.

We will now quickly explain how we tried to solve these issues.

To address the first problem, it is sufficient to note that to form a line in a 3D space, two intersecting planes are needed. In our approach, we used the plane defined by the mesh normal, n, at point p and the plane defined by $v_\perp$, the vector orthogonal to the flow normal v. In this way, the curve would have followed the direction imposed by v.

To solve the second issue we had to modify the first algorithm in this way: for each point to be shaded p, we find p', the closest point to p in the pointcloud map generated by our framework, and through p' we can approximate the value of v. Then we calculate the Dot product between p over v, thus giving us the projection of p along the direction v. The result is then put inside a *sin* function. By looking at the sign of the *sin* function, we know if p is part of the stack of lines defined by the vector v at position p'. By increasing the frequency inside the *sin* function we can add more details.

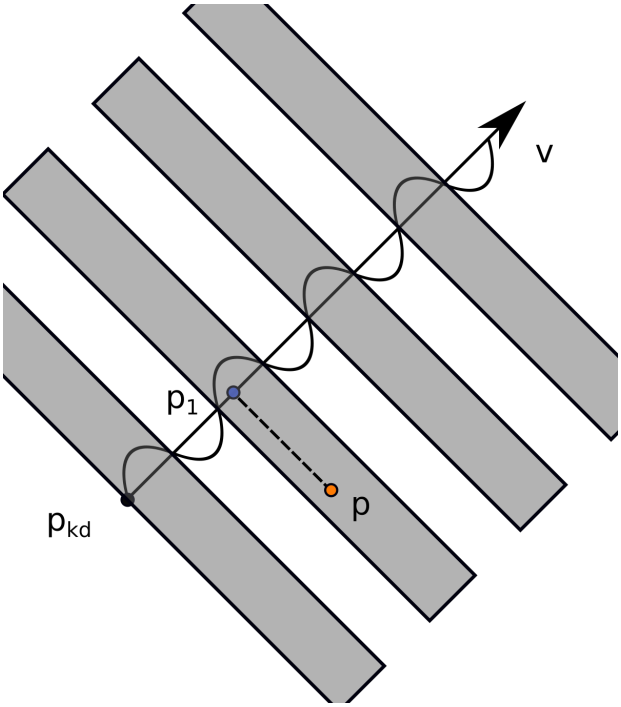


Figure 5.2: Example of stack of planes algorithm. The point to be shaded p is projected along the direction of v_{orth} finding p_1. The sin function is used to define whether or not the projected point is inside or outside the stack of planes. In this case, $sin(p_1)$ is positive, hence the point p is selected to be part of the stack of lines that have direction v.

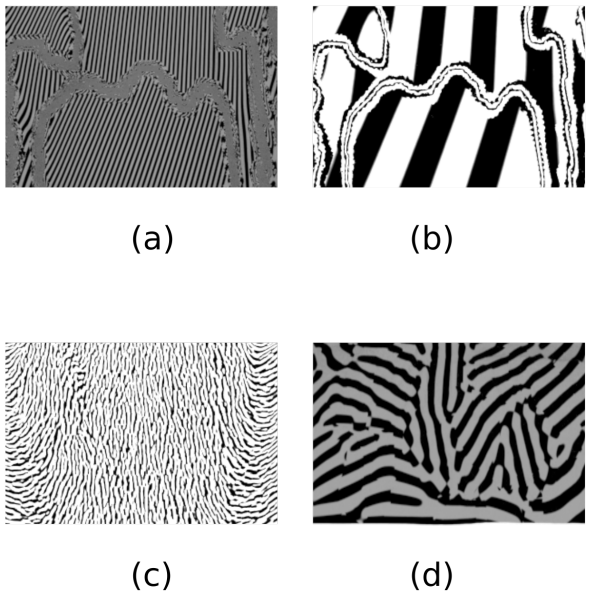


Figure 5.3: (a) Errors caused by interpolating the value of v , (b) Same situations of (a) but with a lower frequency.(c) Result with the cellular automata approach, the output is better but when we decrease the scale of the details we obtain (d). As it is possible to see in figure (d) there is a general sense of flow, but the lines are not always smooth and it is possible to see the interface between cells.

We tried to solve the third issue, i.e. making a continuous 3d function, by interpolating the value of the vector v. Unfortunately, the results were disappointing and insufficient, showing some unpleasant artifacts.

Another solution that we tried was to use a cellular automata approach to avoid the interpolation of the flow vector: we divided the space into regions and in each region, the flow direction would have stayed constant. In this way, we avoided the artifacts related to the interpolation, however, we had some other problem with the continuity of the function. Some of the results can be seen in Figure 5.2. After many trials, these approaches were abandoned in favor of a LIC solution. This algorithm had the advantage of being simple, easy to compute, and entirely procedural without any excessive cost in terms of space, however this is not true for the solution that we are going to explain in the next sections.

5.1.2 DDA convolution and LIC algorithms

In this subsection, we will quickly introduce the two common ways used in scientific visualization to describe a flow field.

One of the earliest algorithms used to visualize flow fields was the DDA convolution [33],[27]. DDA convolution works as follows: for each pixel p it applies a convolution operator between a noise image (i.e. an image with just noise) and a line of which the direction is determined by the vector flow v present at the point p. DDA algorithms can give a good approximation of the flow field only if the variations in the flow field are bigger than the kernel size used for the convolution. This limitation would be later solved by Cabral and Leedom in [11]. In this article, the two authors propose to visualize flow fields by doing an integral convolution over the curve defined by the flow field itself. In other words, each pixel $p(x, y)$ in the flow field is convoluted along a flow field line l. It's possible to decompose the line l into two parts: one before the point p which we will call l_- and one after it (l_+). Due to the linearity of convolution, the convolution of l can be decomposed as the sum of the convolution of l_- and l_+ as well. Since every point p has an associated flow vector f it is possible to approximate the line l_+ by moving at each iteration the kernel convolution center in the direction of f. If repeating the same process but using the opposite of f, starting again from p, we can compute the convolution of l_-.

One downside of this approach is that the result depends on the kernel's length l: if l is small, the convolution will be limited to a small area and hence the result will be mostly dominated by the noise. Unfortunately, there is not a clear rule on how to automatically define l. Using long kernels also has a hidden downside: the box kernel that is commonly used simply averages the value obtained in every iteration. This means that using a wide convolution radius tends to create a monotone image with values around the noise average. This issue can be solved using some contrast enhancement techniques. In order to obtain cleaner images, it is common to use a contrast stretch and edge enhancement filters over the output image. The result of these two filters is then used for another LIC pass.

We will use this entire process to simulate the erosion caused by the flow of the wind.

5.1.3 Our implementation

In this subsection, we will describe our implementation of the LIC algorithm fit for our needs.

The key idea is the one discussed above: for each point that has been selected by the classifier we compute the line integral convolution and we apply a contrast enhancement filter. Then we use this result as input for a second LIC pass and apply a contrast enhancement filter again. At the end of these steps, most of the vertices will be associated with a value between 0 and 1, the output of the LIC algorithm. This output is then stored, among the flow normal, in the class *FlowShaderParameters*. We will later explain how these four floats (three from the vector and one from the output result) will be used by the shader at render time.

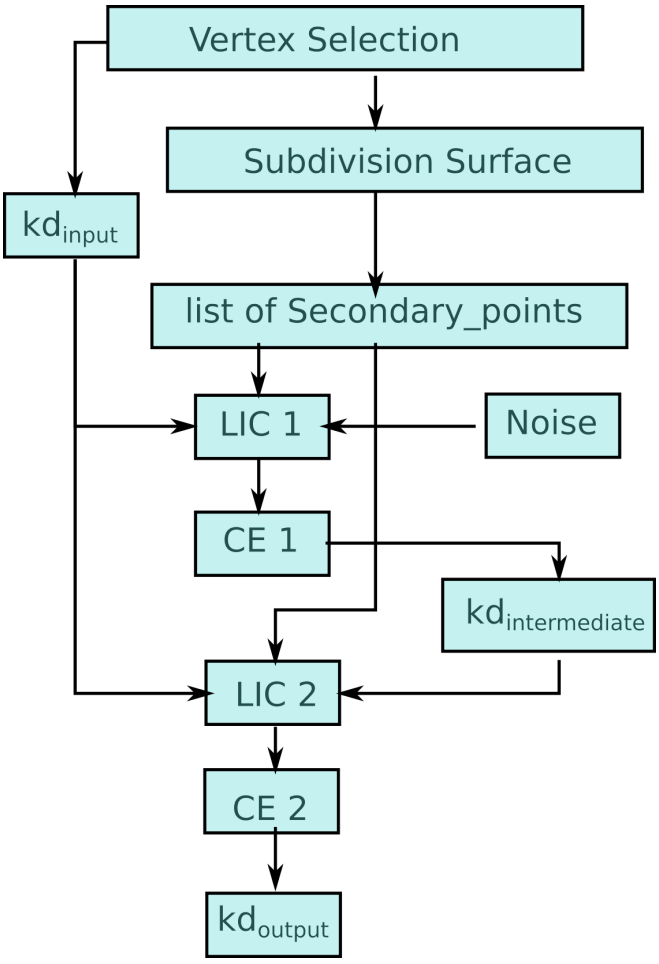


Figure 5.4: General diagram of our implementation of the LIC for our purposes.

In theory, it would have been possible to implement the LIC algorithm in a

shader, however, this would have caused two major issues. First of all the Line Integral Convolution is slow - as estimated by the author in [11], it is a power order slower than the DDA convolution - hence it is not well suited for render time where multiple re-evaluations of the same point might be needed. Furthermore the convolution using a kernel box, for long kernel's length, tends to create an output value that is close to the average of the noise. This creates flattened images with few details. Therefore some contrast enhancement is required even just to show the results, however, most of the contrast enhancement techniques usually require some knowledge of the value of the neighbours, or at least some general knowledge of the image. Furthermore, we wanted to improve the first LIC pass by applying a second pass. To do so every vertex would have been needed to store already the output value from the former step. Once more, this is not feasible at render time. For these two reasons the algorithm that we are going to explain will be computed during the classification and evaluation of vertices phase in our framework.

For this algorithm we will use these data structures:

- a list of vertices $input\_points$, to store the input vertices that will be used as a starting point for the convolution.
- a kd-tree, kd_{input}, to store the position of the points in the coarse mesh that will be used to retrieve the data from the simulation.
- another kd-tree, $kd_{intermediate}$ that will be used to save the result from the first LIC pass.
- a list of points that is the expected output from the classifier
- a list of data that will store the result of this algorithm.

Before starting with our algorithm some pre-processing steps are required:

- selection of points interested by this phenomenon
- creation of secondary points from the coarse mesh

The first point is the goal of the classification phase: select vertices that have data relative to the phenomenon that we want to model. In this case, the data that we are interested in is the wind flow normal. If any vertex in the coarse mesh has a value associated with it, then it is selected.

The wind erosion that we are trying to simulate usually happens on the vertical face of stones, therefore the selected vertices in the previous step are again tested to see if their normal is horizontal, i.e.if they are contributing to creating a horizontal face. Finally, the vertices that also passed this second test are saved in kd_{input}.

The second task that must be accomplished before starting corresponds to the necessity of creating more details than the one present in the coarse mesh. Therefore we need to create more points where to store the data. The position of these new vertices should be as efficient as possible, i.e. the point that we are creating should be used at render time, so we want to avoid wasting memory and time for points that might be inside the mesh or too far away to be selected by any shading

point at render time. Therefore for the creation of the intermediate points we rely on a subdivision scheme, that should create points close enough to the surface. In our implementation, we used the Catmull-Clark method. These extra points will be then inserted in the list of *input_points* along with the points present in the coarse mesh. The vertices present in the list will be used as a starting point for the convolution in the successive steps.

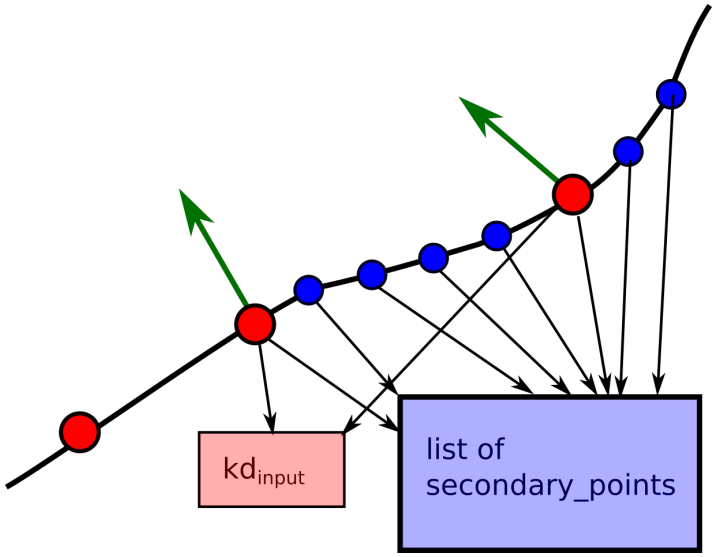


Figure 5.5: Visual representation of how the data are structured for our algorithm. The red points are the vertices present in the coarse mesh. The dark green arrows represent the flow normal and the blue points the secondary points. Only the red vertices with a flow normal are inserted in the kd-tree and in the input_list. The secondary points are generated only between points that will be put in the kd_{input}. The red points that are inside the kd_{input} are also put in the list of secondary_points.

After these two above mentioned tasks have been accomplished, it's possible to start with the main algorithm. We first iterate over the points present in *input_points* and for each of them, we will calculate the Line Integral Convolution. To do so we need a point p and a direction, the flow normal, v. Since p is one of the elements present in the list *input_points*, it will very likely be an interpolated point. Interpolated points, however, do not have associated with them a value v, since v is associated only with the points present in the coarse mesh. To find v we will simply query kd_{input}, find the closest point to p inside the tree, and use its flow vector to approximate the value of the flow vector at point p. Once p and v are known it is possible to begin the convolution. Note that if p is a point that was already present in the coarse mesh, we can immediately know the value of v.

The process done at each iteration of the convolution can be broken down in three steps:

- evaluation of the noise function at p
- evaluation of the successive point p_1, through v.

- evaluation of the successive value of v_1 at point p_1.

The first step is rather immediate, since it requires evaluating a 3D noise function at p. Note that the frequency of such a noise function influences the details scale in the final result. The faster the noise function changes, then the smaller the details in the final output will be.

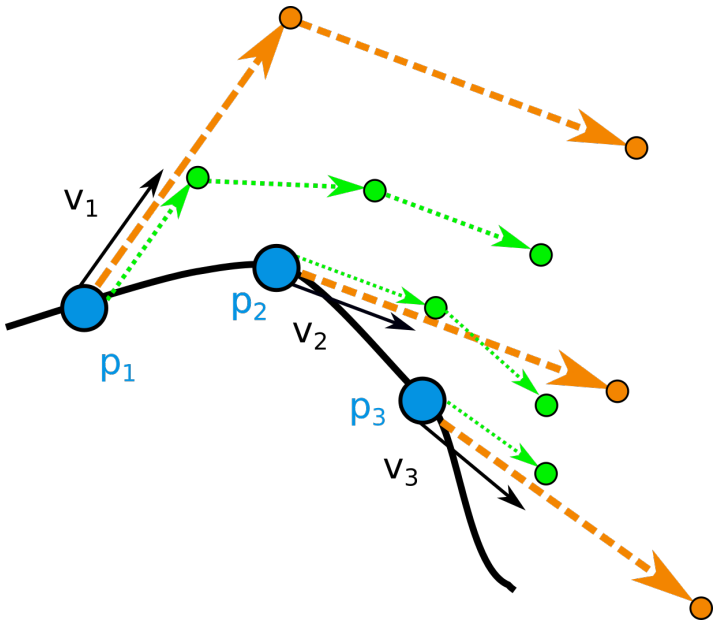


Figure 5.6: Note how the vertices found in the path with a smaller step size (green) are spatially closer together with respect to the ones found with a larger step size (orange). Since the green points are closer together, the evaluation of the noise function at those points will differ a bit. This means that the result of the convolution at point p_1 will be close to the result obtained at point p_2. This is because the values involved in the convolution are similar. If we used the orange paths, the discrepancy in the results will be bigger. Note also how the frequency of the noise texture plays a relevant role: if the frequency is low, the result of the convolution at point p_1 and p_2 using the orange path will be very similar because the evaluation at the points in the orange path will be very similar since the noise function will change slowly.

After this, it's necessary to find the next point where to evaluate the noise function, which is done by approximating $p_1 = p + v * dt$, where v is the flow vector associated with point p and dt is the step size. Choosing the appropriate value of dt is critical for a good output. If the value is too big then there is a high chance that the resulting texture will be incoherent: see Figure 5.6. However, if the value is too small then many steps will be used without adding too much to the final result, this way increasing the time needed to calculate the appearance data.

The last step consists in finding the next value of v, v_1. Once more we have a point, p_1, but we do not have associated with it a flow normal v_1. For this reason, we introduce this third step, in which we interpolate the value of the flow

vector v_1 at point p_1. The interpolation is a simple average of the flow vector of the k-closest vertices over the coarse mesh, that were previously saved in the kd-tree. Another solution to this problem that we tried was to generate the successive point p_1, then select the closest vertex to p_1 inside kd_{input} and then use the closest vertex as the next point. This solution would have been easier because no interpolation of v_1 would have been needed since the closest vertex to p_1 in the kd-tree would have surely had a value v_1 associated with it. However, this solution has the disadvantage that the final result would have been strongly influenced by the topology of the mesh.

Once this last step has been done, the algorithm is ready to repeat all the steps explained before on the new point p_1, this until the ending criteria for the convolution is met: $path\_length \geq l$, i.e. the length of the path is greater than the kernel box length. When this condition is satisfied, the convolution stops and the same procedure is repeated starting from the same initial point but using the opposite direction v. Recall in fact that the Line Integral Convolution is done in two steps: first the convolution of l_+ and then that of l_-. This process is done for each vertex selected in the pre-processing steps. The results of each iteration are then saved in another kd-tree map.

As already stated, after the first passage a contrast enhancement filtering is done over the result. Since most of the computed values are close to a particular value, in this case, 0.5, we used the contrast stretching algorithm. By cutting 2-5 % of the values, we found that it is possible to visualize some good results.

The output of this passage is passed as input for a new Line Integral Convolution. For this second pass, the steps to be executed are almost the same, except for the first one. Instead of evaluating a 3D noise function over the point p, we need to know the value of the first LIC step at point p. Unless p is the starting point from the convolution, there will not be any value saved at position p in the point-cloud $kd_{intermediate}$. Therefore we take the k-closest neighbour to p in $kd_{intermediate}$ and we evaluate the value of the LIC algorithm by averaging the k-closest neighbours.

After this second LIC passage, the result is once more subject to some contrast stretching filtering. Finally, the position p is saved in the output list of points and the result from the second contrast stretching is associated with the position p.

Once this last step is done the FlowClassifier will output the list of points, the list of associated values, and the minimum density value.

5.1.4 Flow shader

As explained before, the LIC algorithm is computed before the rendering and the results are stored as a pointcloud map that will be read by a specific shader at render time. In this subsection, we will analyze deeper how this shader will use the results from the processing step to shade every point.

Each vertex in the point cloud map has 4 floats: one is the actual output calculated through the previously described algorithm, and the remaining 3 floats are used to store the flow vector associated to that point. Using the flow vector it is possible to filter the texture to avoid aliasing artifacts: usually to filter a point in 3 dimensions, 8 points are used (to select the 4 closest cubes to p). However, if we know in which direction the function will change, the problem can be simplified to just sampling along the direction of changing. For this reason, we pass through the point cloud map the flow direction, from which it is possible to know where the change will be: along the flow direction and orthogonal to it, (we can ignore the tangential direction since it will be either inside or outside of the material). This gives us the possibility to filter our texture using 4 samples instead of 8. The sampling is done by querying the kd-tree in a new point $p_i = p + delta \cdot v$, where p is the point to be shaded, v is the direction in which we want to sample and $delta$ is a constant that represents the differential increase.
Having the flow vector also gives us the possibility to create and add very fine details using the DDA convolution, which has the advantage of being fast to compute, although it is imprecise. We noticed that for a small convolution radius both DDA and LIC lead to almost the same results, which can be explained by the fact that for small portions of space, a function can be approximated with a straight line. From this approximation we can see that both DDA and LIC algorithms will gives us the same result, because the flow field is now a straight line. Therefore, for small details the DDA can approximate the flow field almost as well as the LIC, but with the advantage of being faster.

Finally, the DDA contribution is summed up to the filtered output of the LIC. The final result is then used to displace the points. Since the result can be monotonous, it might be interesting to add some Fbm into the final result. Please consult Chapter 7 to see the result of these three approaches.

5.1.5 Parameters choice

One major drawback of using this approach is that the user has to manually set some parameters for each scene: first of all the frequency of the noise function used during the first LIC pass.

As already explained before, this parameter influences the frequency and scale of the detail in the final output.

The number of subdivision steps, which defines how many secondary points are created starting from the coarse mesh, should be directly influenced by the frequency of the detail. This is because more and more points will be required to catch the high-frequency details generated using the noise function.

Another important parameter that must be accurately set is the dt on which each step in the convolution is done. As already explained, before a large value of dt will lead to an inconsistent texture, at the same time values too small will lead to very long computation time. We believe that setting dt equal to the average

distance between the vertices present in the list input_points can be a valid choice, although we did not try it.

All these three parameters are related to each other: a high-frequency noise will create many details, hence more and more subdivision steps are needed to support such request of details. At the same time, increasing the number of vertices decreases the average distance between vertices, which affects the choice of dt.

One last parameter that is left free is K, the number of points selected for the interpolation of the flow normal. We found that a small number between 3 and 9 usually works fine. Future improvements for this approach will be discussed in the dedicated chapter.

5.2 Sedimentation layers

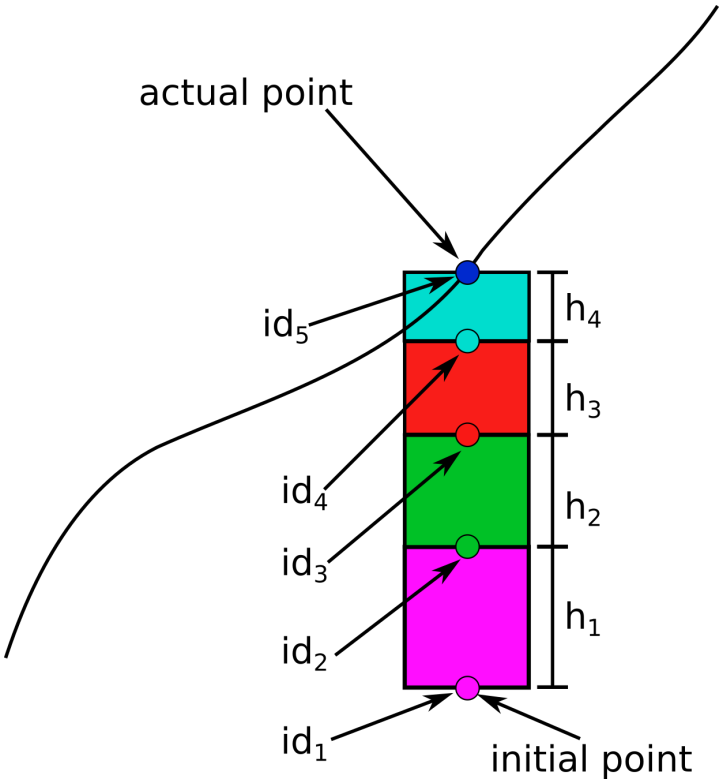
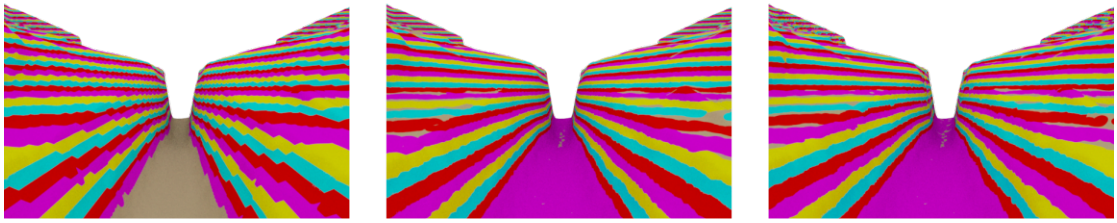


Figure 5.7: Visual representation of the data structure for the sedimentation phenomenon.

In this section, we will discuss the other shader that we developed, the sedimentation shader, which aims to reproduce the sedimentation structure, using the data provided by the simulation. To represent this phenomenon, the framework expects:

- an initial point, which represents the initial point from where the sedimen-



(a) (b) (c)

Figure 5.8: Results on the mockup data with the three visualization algorithms (a) NP (Nearest Point). (b) NPTS (Nearest Point To Surface). (c) RBF approach

tation started
- a list of material indices indicating the various materials that compose the material stack
- a list that stores the thickness of each layer

For each vertex present in the mesh, we use this data to create a list of points that represent the various steps of the sedimentation process. The creation of those intermediate points is rather simple: we start from the initial point and then we add to its height the thickness of each layer. Check Figure 5.7.

This list of intermediate sedimentation points will be used later in all the strategies that we are going to describe. Note that if the landscape is relatively flat and smooth, only the points at the top of the list of intermediate sedimentation points are needed to render the scene. In places where there are sudden changes in the landscape (e.g cliffs), it will be possible to see the intermediate sedimentation points.

With this data, we tried three different solutions to represent the sedimentation effect.

Unfortunately, we realized very soon that the data from the simulation were noisy. For this reason we decided to first use some mock-up data, to test and compare the various visualization algorithms in a known environment, and then apply these algorithms on the real case. This made the process of debugging easier.

We will now proceed to analyze the various algorithms that we developed to visualize the sedimentation data.

5.2.1 Nearest Point

The first strategy that we tried was to insert every point in the list of intermediate sedimentation points inside the pointcloud that would be read by the shader at render time. As it is possible to see in Figure 5.8, this solution tends to create a "blocky" texture.

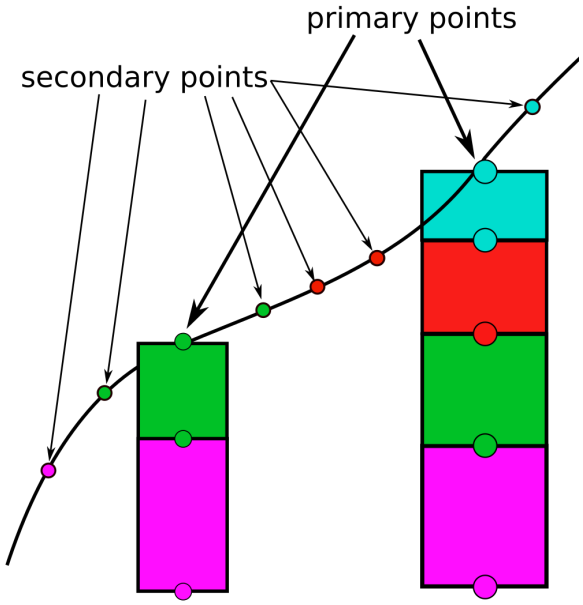


Figure 5.9: Visual representation of the Nearest point to surface algorithm. Secondary points are created around the surface. The material id of the secondary points is taken from the closest intermediate sedimentation point stored in each vertex of the input mesh. The points are then smoothed, but this is not represent in the picture

5.2.2 Nearest point to surface

The second algorithm that we would like to discuss was devised to suppress the "blocky" look present in the first approach.

The workaround that we found for this problem was to smoothen the results obtained with the previous technique. Therefore, we created intermediate secondary points, as we previously did with the flow shader. We iterate over the secondary points and for each of them, we select the id of the closest point in the list of intermediate sedimentation points. Then we run a smooth operator, implemented through averaging of points, to these secondary points. The results can be viewed in Figure 5.8-b.

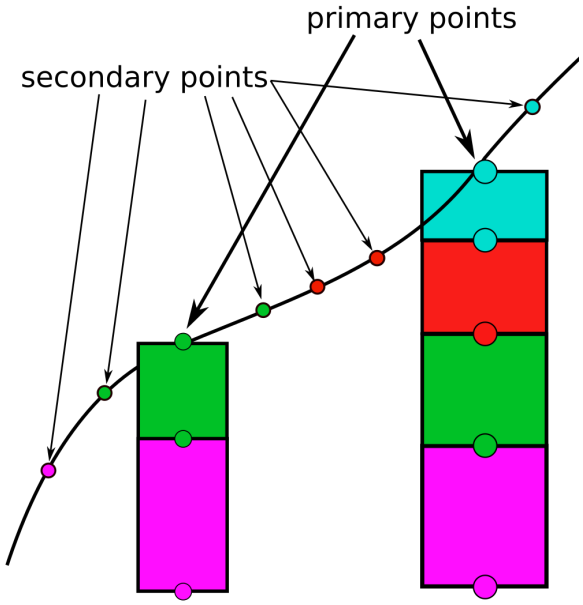


Figure 5.10: Visual representation of the Nearest point to surface algorithm. Secondary points are created around the surface. The material id of the secondary points is taken from the closest intermediate sedimentation point stored in each vertex of the input mesh. The points are then smoothed, but this is not represented in the picture

5.2.3 Minimization algorithm

The above mentioned processes managed to obtain good results with the mockup data, however, they were not able to suppress the noise in the input data. For this reason, we tried to develop another solution by redefining our goal as a minimization problem.

More specifically, we have a point to be shaded, p, and we want to associate a material id to it. Since we know how many material ids are in the scene, we would like to pose our problem as finding the material id, id, that minimizes the value of a utility function f at point p.

To build f we start by noticing that sediment layers, most of the time, can be enclosed by two polynomial function. One delimiting the upper part and one the lower. Therefore we define two polynomial functions, $MinLayer_{id}(x, y)$ and $MaxLayer_{id}(x, y)$, with $MinLayer_{id}(x, y) \leq MaxLayer_{id}(x, y)$. With this model we can claim that p will have id if $MinLayer_{id}(x, y) < p.z < MaxLayer_{id}(x, y)$, i.e. p will be associated with id if the height of point p lies between the two extreme functions $MinLayer_{id}(x, y)$ and $MaxLayer_{id}(x, y)$. With this criterion we should be able to associate p to the right id, although in our case, more than

one material id was able to verify this inequality. For this reason we defined our utility function f as:

$$f(id, p) = |MinLayer_{id}(x, y) - p.z| + |MaxLayer_{id}(x, y) - p.z|$$

This process is based on representing the value of the two functions $MinLayer_{id}$ and $MaxLayer_{id}$ at x, y. To find these two values we relied on interpolating the data using the RadialBasis method.

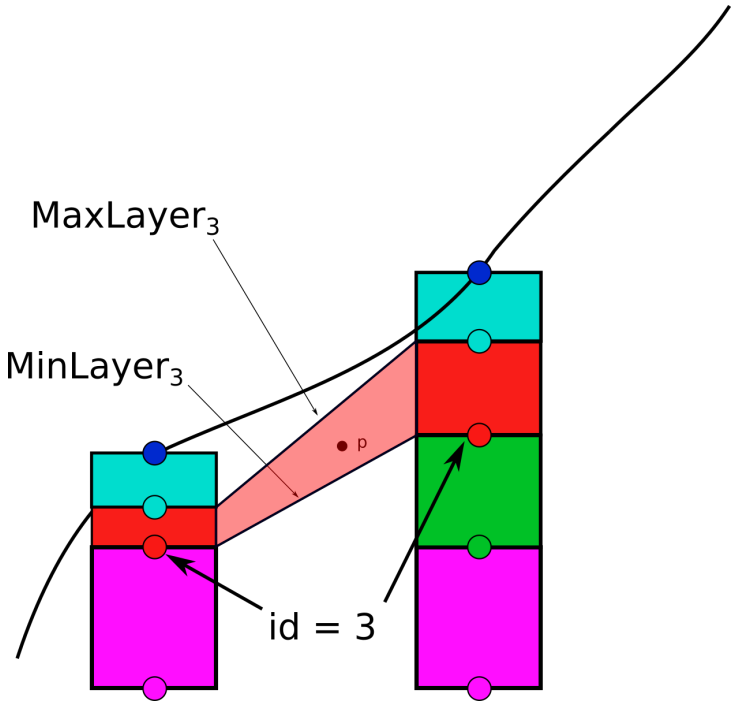


Figure 5.11: Visual representation of the key idea at the base of the minimization algorithm. Every sedimentation layer has a lower and upper edge, and we can use those edges to reconstruct the function that encloses the sediment layer. In this case, since p lies between the maximum and the minimum functions defined by $MaxLayer_3$ and $MinLayer_3$, p will be associated with $id = 3$.

We generate the data on which we interpolate by selecting the k-closest intermediate sedimentation points to p with material id id. Remember that the data structure is created in such a way that each intermediate sedimentation point represents the start of a sedimentation layer with a particular id. Therefore, each of these intermediate selected points defines an interval that starts from the position of the point and ends at the position of the next point in the list of intermediate sedimentation points. With these k-intermediate sedimentation points, we can approximate the two functions with the RBF method. Figure 5.11 depicts this situation.

It is possible that near point p there are two different and distinct layers with the same material id. Figure 5.12 shows a possible scenario of this issue. We solved this problem by introducing a penalty factor for points that are spread too

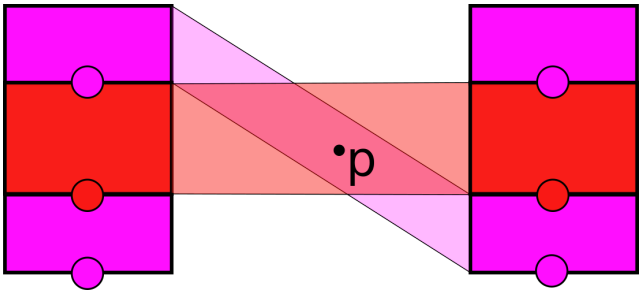


Figure 5.12: In this scenario, the point p is enclosed between the purple and the red layers. Since our utility functions favour layers that are close to p, the purple layer will be selected. The solution that we propose is to penalize layers that are constituted by points with higher variance. In this case, since the red points are aligned, their variance is minimum, so the purple id is penalized and the red id, the correct one, will be selected.

much, hence we define $f' = Var(minPoints)^2 + Var(maxPoints)^2 + f$, where minPoints and maxPoints are the points that will be used to calculate the lower and upper value of the layer, and Var is the variance operator.

Unfortunately, the results obtained were really unstable: we were not able to decide whether this was caused by the implementation of the RBF method or because the method itself is not suited for this situation. The main cause might be that the points that we are interpolating are quite close to each other and this might cause some instability. In particular, we noticed that the interpolation was generating values that were out of range of the height of the points, i.e. it was creating values smaller than the height of the lowest point or bigger than the highest. For this reason we defined

$$f'' = f' + |RBF(x, y, minPoints) - min.z| + |RBF(x, y, maxPoints) - max.z|$$

where $min.z$ is the height of the lowest point in the minPoints set and $max.z$ is the height of the highest point in the maxPoints set. With f'' we were finally able to create results that were comparable with the previous methods - see Figure 5.8. However, as we will see later, the results with the simulation data are worse than the ones obtained with the NPTS technique. Nevertheless, we described this algorithm because we still believe that modelling this task as a minimization can lead to better results.

5.2.4 Sedimentation Shader

The sedimentation shader that will be used at render time is relatively simple: every shaded point that is selected by the mask associated with the sedimentation shader will fetch the data stored in the closest point inside the point-cloud map dedicated to the sedimentation shader. The data passed is simply a float that will be used as an index in an array of colors to select the color associated with

that index. In this prototype we use an array of colors to represent the materials for simplicity, however, in the future it would be possible to create an array of shaders and use the data to index it.

6. Implementation

This chapter is dedicated to discussing the various technologies used in the development of our project. Furthermore, we will describe more in detail how to run the application.

6.1 Technologies

In this section, we will present the various technologies used and why they have been chosen.

6.1.1 C++

For the development of our project, we decided to implement everything using the C++ programming language. We decided to use this language for various reasons. Firstly, the pointcloud API in RenderMan supports only C++, and furthermore, the only possible way to write a pattern not using the OSL is in C++. Secondly, the "Terrain Erosion" project was already implemented using this language, so it was natural to keep developing using the same language. It was also chosen because it was the language in which we were most confident.

6.1.2 RenderMan

RenderMan is a render engine developed and owned by Pixar Animation Studios. We selected this render engine because we were already familiar with it.

6.1.3 OpenMesh

OpenMesh [3] is a C++ library that has been developed by RWTH Aachen University. We used the OpenMesh library to read the input file and store the data.

6.1.4 PointCloudLibrary

We used PCL (PointCloudLibrary), which can be found at[4], to store the points in a pointcloud.

6.1.5 Project

6.1.6 MathToolBox

MathToolBox [2] is a C++ library that we used to interpolate data using the RBF approach for visualizing the sedimentation data.

6.1.7 FastNoise

We used the FastNoise GitHub project [1] to evaluate the 3D noise function in the LIC section.

6.1.8 Third Party software

We used QtCreator as IDE and RenderDoc to debug some errors caused by the preview window. Furthermore, we used both Maya and Blender to visualize the mesh outputted by the simulation.

6.2 Running the project

The project has three folders[1]: Simulation, AutomaticShading, Data. In the first folder, it is possible to use the Terrain Erosion Project with our proposed algorithm to generate the input scene and the simulation data. In the second folder one will find the Qtproject file that is needed to build our framework. Once compiled and launched, the program will search in the folder named "Data" for the input file and the simulation data. These two files are already present in the folder. As output, our framework will generate four pointclouds: two for the wind erosion and two for the sedimentation. It will also automatically generate a rib scene.

[1] Available at https://github.com/dariolanza95/AutomaticShading

7. Evaluation

Deciding whether a generated terrain is realistic or not is not a trivial task, this because it is hard to quantify the realism of one image over another. Most of the time, a qualitative approach is used by authors by comparing the images obtained in previous results or some images from reality. It is left to the reader to decide whether or not the results are achieved. Therefore we will, as is usually done in literature, show our results and compare them with some reference images, highlighting their pros and cons. In this chapter, we will analyze the various results obtained with the proposed framework and the two examples.

To do so, we run a series of simulation with the help of our modified version of the project [5].

7.1 Experiments

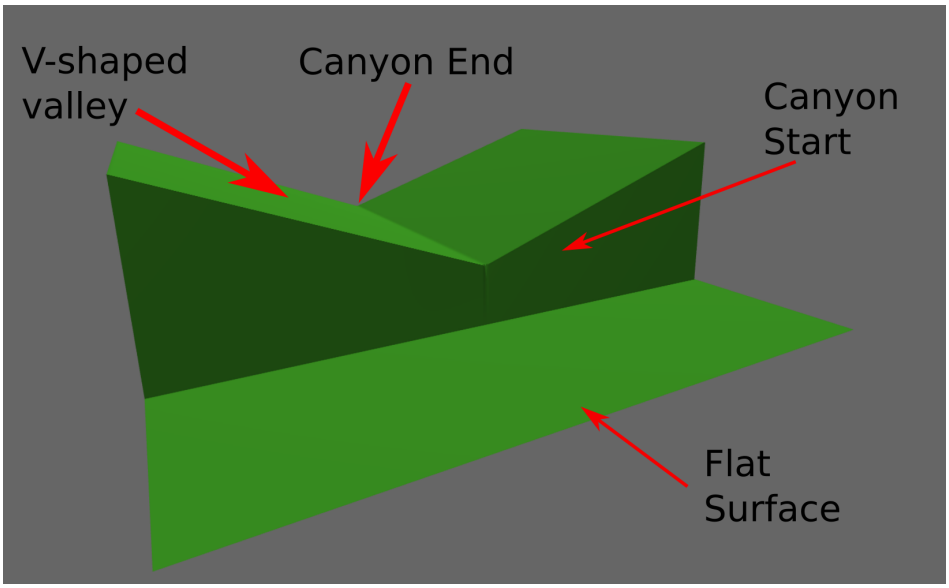


Figure 7.1: Scene that will be used for the simulation of both phenomena, only the simulation will change between one scenario and the other.

Since the phenomena that we wanted to reproduce are not that common, a general simulation would have required an extremely long period of time to be able to properly show them. For this reason, to prove the validity of our framework we had to use some special set-up that would have highlighted the phenomena that we wanted to simulate. For our final results, we started with the same scene that can be seen in Figure 7.1, but we used different simulation set-ups. As can be seen in the figure, the scene is divided into two parts: a V-shaped valley, and a flat surface.

We ran two experiments with our simulation. One simulation was focused on highlighting the wind erosion phenomenon, whereas the second had the goal of showing the sedimentation process.

With the data from these two simulations, we generated a series of images to show the results that obtainable with our approach.

7.2 Wind Erosion

This section will be dedicated to commenting and discussing the results obtained with the wind erosion.

Simulation Set-up

To highlight this phenomenon, the wind erosion, we thought to create a canyon and let the wind flow through it. Therefore a river is created at the end of the valley and is allowed to carve the bedrock until a canyon enough deep is created. The fact that the valley is V-shaped speeds up the process of carving the canyon. Later, the wind simulation is used to collect data about the flow of the air.

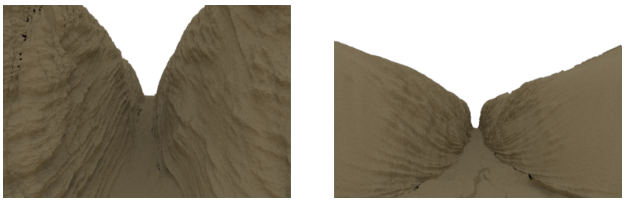


Figure 7.2: Images of the canyon viewed from the beginning and the end.

Figure 7.3: (a) Antelope Canyon,Arizona (b) Wave Coyote, Arizona

Rendered Images

Figure 7.2 shows the result of using our FlowShader. Figure 7.3 shows two images that we used as reference.

As it is possible to see, our flow shader can recreate the general sense of flow showed in Figure 7.3, but some improvements are still needed before reaching realism. We believe that three main problems need to be addressed to reach more convincing results.

Firstly, the macro-geometry: as it is possible to see in the figure, the geometry in image 7.3-a is rather complex, showing often overhangs. The geometry in our scene is comparatively simpler and the overhangs are not present at all. This is caused by the fact that the simulation, which uses a heightmap approach, cannot simulate this behaviour. A more physically plausible simulation would have probably been able to recreate such complex macro-geometry. Furthermore, although not in every situation, this is a problem, as it is possible to see in Figure 7.3-b, in which the part in the foreground shows a rather simple macro-geometry that our images shows.

The second problem that should be addressed is about the texture: the reference image shows, in many points, a correlation between color and the displacement of the geometry beneath it, which is highlighted in Figure 7.3-b. This figure shows an example of wind erosion mixed with a sedimentation layer: as it is possible to see, the color and the geometry matches, which is caused by the fact that layers of different materials, and hence of different colors, can be harder to erode than others. This situation causes some small layers of material that are harder to erode, to stand out from the rest of the geometry nearby, creating these lines. This condition is not addressed in our shader, because we initially did not consider how the wind erosion might be influenced by the sedimentation layers. For now, we only model the displacement caused by wind erosion, but in future, this might be addressed. A solution that we would like to try in future is to blend the noise texture with the sedimentation data: in this way, the texture that will be convoluted in the LIC passage will be influenced by the sedimentation data.

The third problem is about the frequency of details: as it is possible to see in the reference figure, there are many small details that our shader can partially handle thanks to the DDA algorithm. However, those details created by the DDA do not follow the general flow as the LIC map does. We believe that better results could be obtained by increasing the resolution of the LIC map, this because the details created through the DDA are just an approximation of the wind flow.

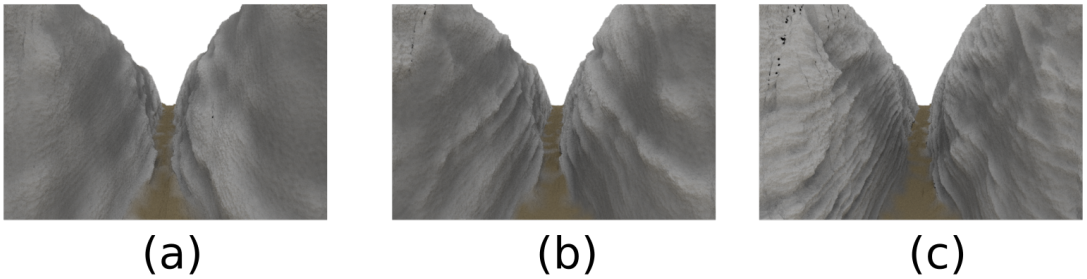


Figure 7.4: (a) Subdivision level = 1, (b) Subdivision level = 2, (c) Subdivision level = 3

As it is possible to see in Figure 7.4, increasing the number of subdivision creates more details that better approximate the reference images. However, increasing the number of subdivision has a heavy cost: as we will see later in section 7.5, the time needed to calculate the LIC map increases rather fast with the number of subdivisions. One solution to this problem would be to implement a parallel version of this algorithm. This should be possible since the Line Integral Convolution of one point can be done independently from the others. For now the images that we are showing have a lower amount of details, if compared to the ones present in the reference images. At the same time, the amount of details that we are showing is rather high if we compare it to the number of vertices that were present in the input mesh after the simulation. Refer to Figure 7.5 for a comparison.

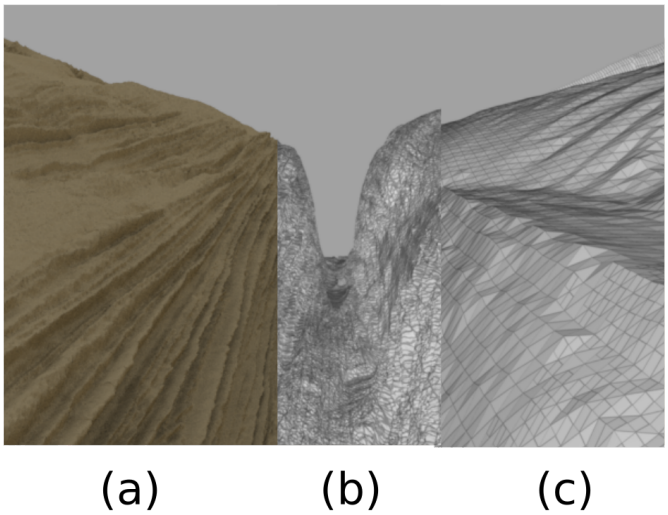


Figure 7.5: (a) Final results with colors, (b) Geometry used in the final result,(c) Geometry of the input mesh

As we have already explained in the dedicated chapter, it is possible to add some

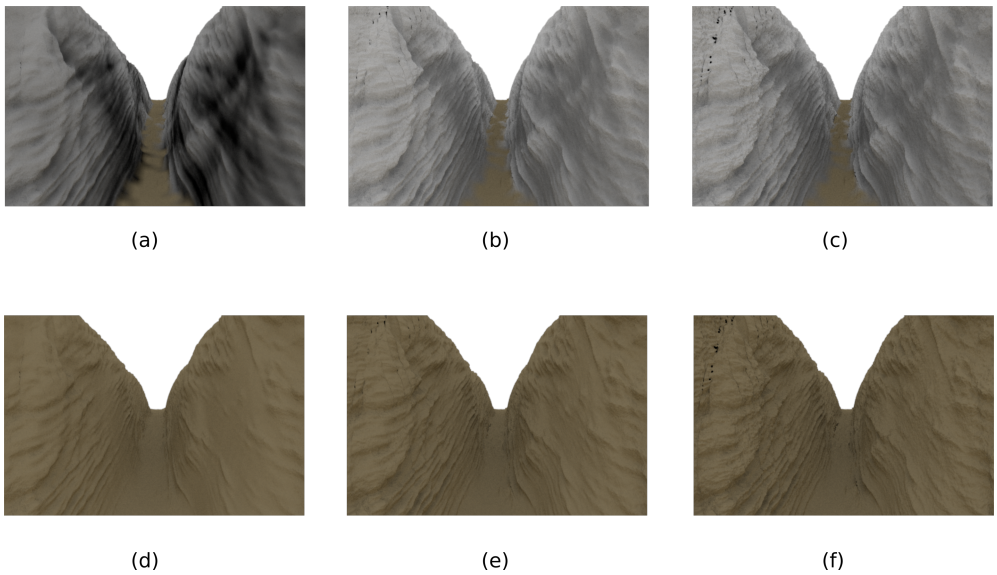


Figure 7.6: Comparison of the various techniques that can be used to add more details to the scene. In the first row the material is colored in a grey-scale to better highlight the subtler details of the displacement map. In the second row, we show the final result. (a) only LIC , (b) LIC + DDA , (c) LIC + DDA + Fbm , (d) only LIC , (e) LIC + DDA , (f) LIC+DDA+Fbm. The results with the LIC are the smoothest. The DDA is used to add minor details using the flow normal. Finally, the Fbm is applied to add further details.

minor improvements to the results that can be obtained through some extra techniques (Fbm, DDA). Figure 7.6 compares these various techniques. The use of the displacement can cause some errors as can be seen in the figure. Currently, the only way to reduce this type of error is by decreasing the global strength of the displacement. In future, it might be interesting to run, during the classification and computation step, some test that calculates how much the geometry can be displaced without unwrapping on itself. In this way we could generate a mask able to locally reduce the strength of the displacement.

For now, the main limitation to this technique lies in the memory space consumption needed to calculate the LIC map: for having more details, the frequency of the noise texture and the number of intermediate points must be increased, as can be seen in Figure 7.4. This because the more rapid the texture is changing, then the more points are needed to catch these little changes. This also influences the choice of the parameter *step_size*, which should decrease as the density of secondary points increase.

In conclusion, we think that the solution that we are proposing to simulate the erosion driven by a fluid is a good approach, although many improvements can be done.

Firstly, an automatic choice of the parameters is needed, and secondly a better logic to classify the vertices is also required: as it is now, every vertex that has a normal almost horizontal and a flow normal associated with it is selected. As can

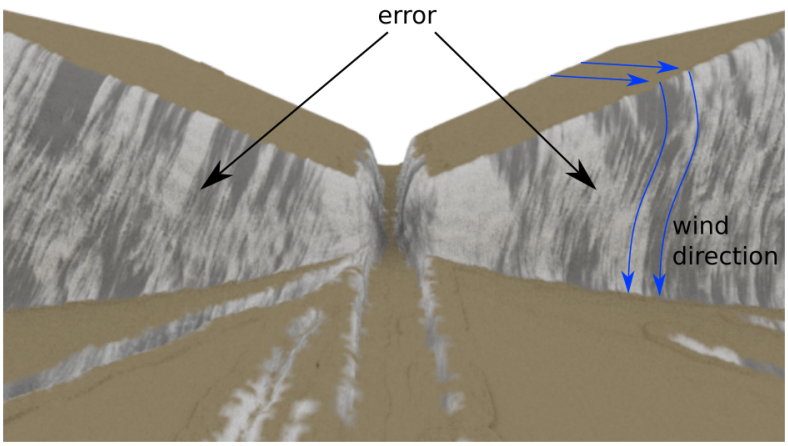


Figure 7.7: Example of error due to a wrong classification of vertices, an aspect that should be improved. The blue arrows represent the direction of the wind, as it is possible to see the air "fall down" like a fluid, this because our simulation of the wind is rather imprecise. The fact that the cliffs are selected to be part of the wind erosion is not an error of the wind simulation but from the classifier: the vertices that compose the cliffs have associated with them a flow normal, and for this reason they are selected without checking if the air would have enough strength to erode a large portion of the landscape. .

be seen in Figure 7.7, this is a mistake: the wind erosion is noticeable only when the winds flow with a remarkable pressure for long periods of time. In future, this might be addressed by introducing new data to take from the simulation: the average air pressure. However, there is a problem with using such a parameter. What we would usually expect is to receive the air pressure in N/m^2. The problem lies in the fact that most of the time the simulations are substantially a-dimensional, so calculating the pressure might be problematic. One idea would be to make the pressure a-dimensional as well, but at that point a new problem would arise: different simulations might calculate this a-dimensional value in different ways.

Furthermore, the implementation of the modified LIC algorithm that we are proposing should be made more space and time efficient. As we explained before, these are the main limits in having texture with higher resolutions.

Note that this is not required to visualize the FlowShader in real-time: once the LIC map has been calculated, it remains the same for the entire scene. The problem for real-time applications that want to use this algorithm might lie in the pointcloud that must be loaded. One possible workaround that we did not try, due to the lack of time, was to store the result of the evaluation of the flow shader in a 2D texture, and then use the 2D texture in the render, instead of the shader. In this way, at render time there is no need to look-up in a 3D pointcloud and do further adjustments. We believe that this workaround could improve the render time by a good factor.

7.3 Sedimentation

In this section we will analyze and comment on the results obtained with our framework to describe the sedimentation phenomenon.

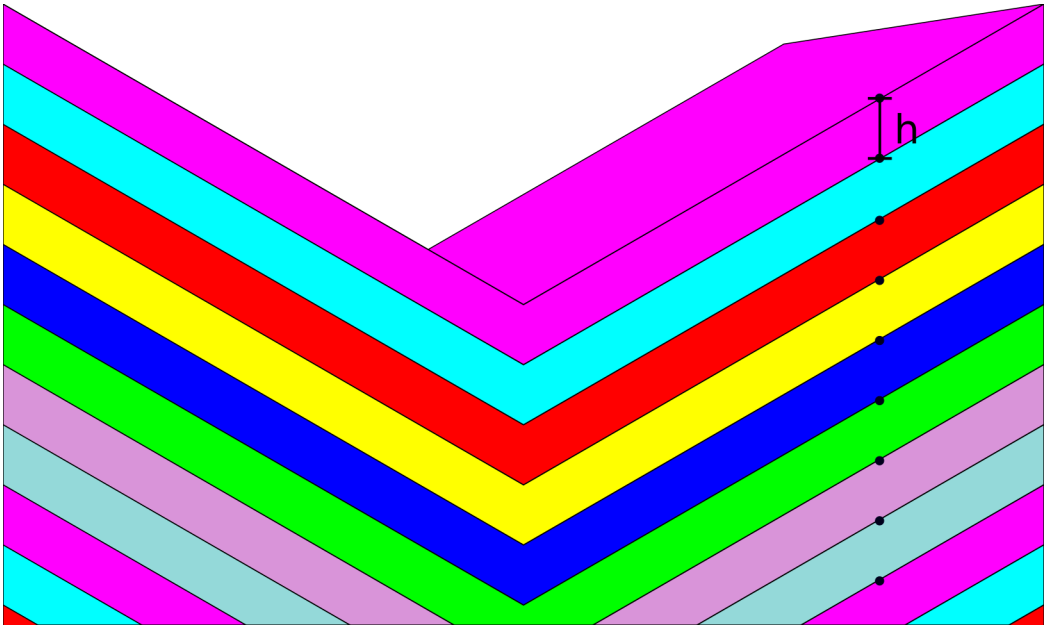


Figure 7.8: Disposition of the various sediment strata that are initialized before the simulation. Starting from each vertex's height a new layer is created after h units length until reaching the base of the scene. Note that we have a fixed number of material ids so after a certain number of iterations the material ids are re-used.

For this scenario we used the same scene in Figure 7.1 but with a different simulation set-up from the one used for the wind erosion. The scene, in fact, is now initialized with some sedimentation layers to represent the sedimentation that happened in the past. In this scenario, each vertex on the coarse mesh will have associated multiple layers based on its height. Figure 7.8 shows how they are disposed. In this way the upper part of the scene will be initialized with multiple layers of sedimentation, while the second part, that is flat, will have only one layer since it is at the zero level. We start the simulation by letting rain in the V-shaped valley: in this way the water will erode and transport the sediment present in the upper part and will deposit it in the lower part.

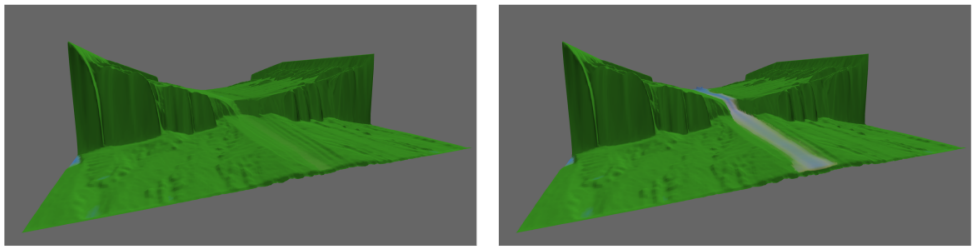


Figure 7.9: (a) After the first part in which the rain will transport the sediment from the upper part to the lower part of the scene, a ramp will be created between these two zones. (b) The river that will carve the ramp revealing its sedimentary structure. The rain stops after a fixed amount of time, and during this time a ramp that connects the two parts will be formed. The ramp is then carved by a river, in order to reveal its inner structure.

This initial phase of rain is then stopped when a ramp of sediment has connected the lower part with the upper, as can be seen in Figure 7.9. When this situation is reached, a river is used to carve the scene and reveal the sedimentation pattern present in the scene. With this approach, we can be sure that the sediment present in the lower part of the scene is entirely due to the modified sediment transport that we are proposing.

The rest of this section will be focusing on two aspects: the simulation itself, and the visualization of the data from the simulation.

7.3.1 Simulation

This subsection will be dedicated to talking about the results obtained through the sediment transport simulation. We dedicate this subsection to this argument because it plays a key role in the visualization of this phenomenon. If the data that the framework is receiving are incoherent or wrong, a visualization algorithm can hardly show any reasonable results.

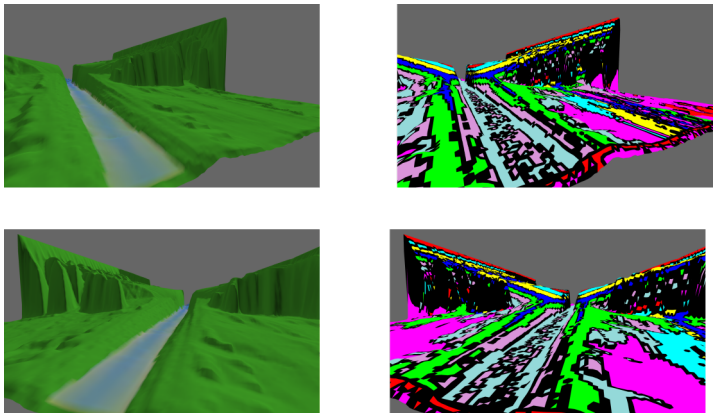


Figure 7.10: On the right, the pre-visualization of the sediment data. On the left, the pre-visualization of the geometry.

In Figure 7.10 it is possible to see the results of our modified simulation. Overall, the multiple sediment transport seems to be working although it is rather imprecise and noisy. Two proofs allow us to make this claim.

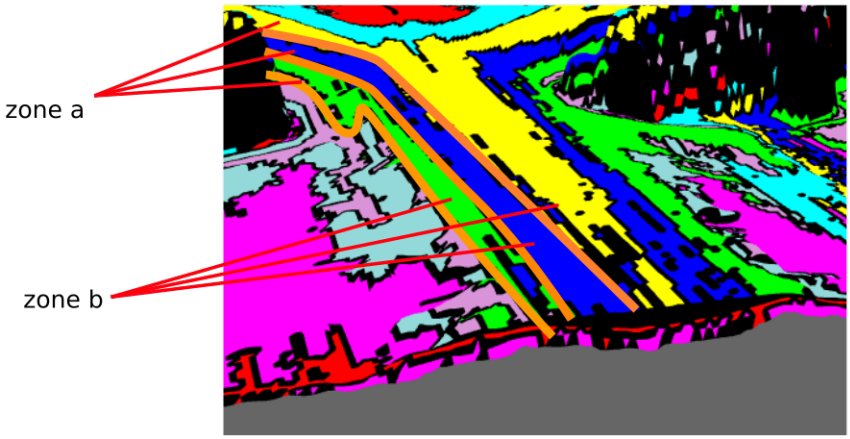


Figure 7.11: Note how the disposition of the sediment in the zone is replicated in the zone b. The orange bands are added to highlight the edges of the sediment layers. This is evidence that suggests the transport of the various sedimentation materials.

First, the disposition of the sediment layer in the lower part continues the disposition of layers present in the upper part. Note that the disposition of layers in the upper part was defined by the initialization of the various sediments layers, whereas in the lower part there was only one sediment layer. This means that

the actual disposition of the sediment layers in the lower part is entirely due to the sediment transport. The second evidence that we want to show is depicted in Figure 7.12, which shows the evolution of the river banks. As it is possible to see, by carving the river bed, a stratified pattern is revealed. This is a further demonstration that our modified algorithm for multiple sediment materials work.

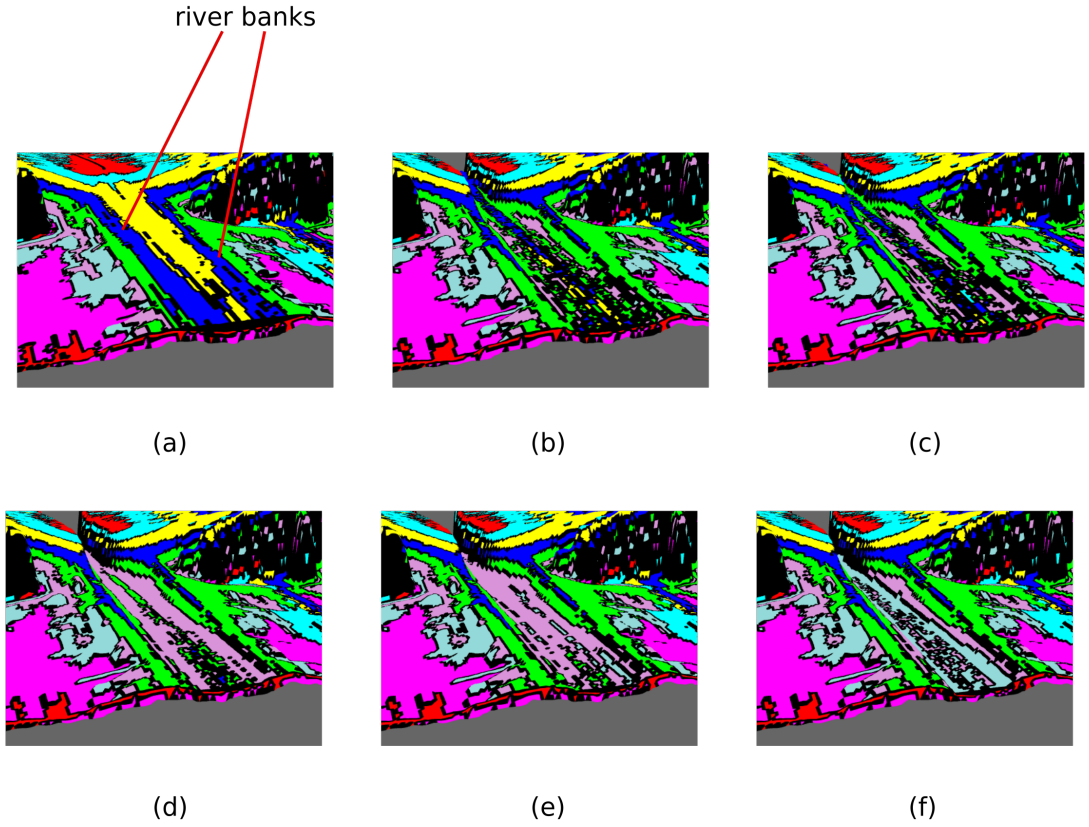


Figure 7.12: The images show the evolution of the river banks and the slow revealing of the underlying stratified structure. Figures a-b-c depict how the blue sediment is removed showing the green sediment. In figures d and e, the light purple sediment is revealed by consuming the green sediment. Finally, in f we can see how the light blue sediment is, once more, taking the place of the upper layer (light purple).

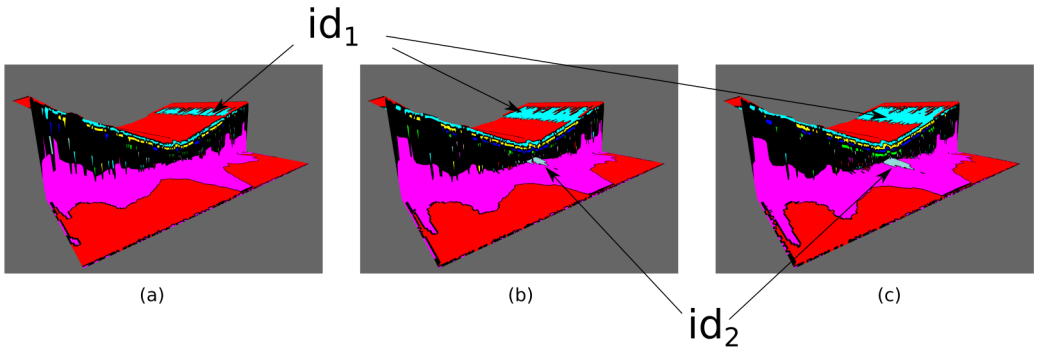


Figure 7.13: Example of how the various sediment types slowly spread in the scene. (a) The material id_1 is uncovered due to the erosion, since it is visible in the debug tool, it should also start to spread the sediment. (b) The material id_1 starts to spread, but only the zones near its starting point, in the lower part the material id_2 starts to spread as well. (c) the situation is the same in (b), what we would have expected though was to see id_1 spreading also in the lower part of the scene, since the water flow in that direction but that did not happen.

What we have noticed, though, is that the sediment spread rather slowly. Figure 7.13 depicts this situation: in (a) in the lower part, the id_1 sediment is spreading, while in the upper part the id_2 sediment is eroded: what we would have expected was to see the id_1 sediment spreading also in the lower part, since the id_1 sediment started to diffuse earlier than id_2. This can be explained by the high sediment counter that we imposed. Recall that the sediment counter is a threshold that we imposed to avoid instability: before a new sediment layer is accepted the same sediment must be received at point p at least c times. This means that if c is a high value, the simulation will have a more conservative approach, hence the sediment transport is slower, but with the benefit that there is less noise. One thing that we noticed is that if we lower this threshold, the simulation starts to become pretty noisy. This because every time that a particle deposit is almost immediately accepted, creating a new sediment layer, with the effect of generating an almost infinite stack of microscopic layers, that in the end will have a visually negligible impact. With a high value of x this problem is smoothed: particles that randomly deposit over the point p are essentially ignored in the creation of a new sediment layer.

The question is which of the two behaviours should be used? The more chaotic one, but in line of principle, more correct, or the stabler one, which approximates the sedimentation stack? Note that in any case the second version also has a considerable quantity of noise, this because the problem itself is chaotic: in the scene, there are 8 different types of material ids that are spreading across the scene at the same time. Some blending of these materials is expected, but it is not clear to us how much. Some more physically accurate model should be used and the results of the two models should be compared.

To the best of our knowledge, we were not able to find, in existing computer

graphics literature, any model to represent the transport of multiple sediments material. In [21], the authors devised an algorithm to handle the transport of more than one type of sediment, however after the sediment has been deposited it is considered to be part of the bedrock and hence it will not be possible to erode again. The novelty of this algorithm lies exactly in this: allow sediment that was previously deposited to be eroded and transported more than one time. The approach that we are proposing also has the benefit of being able to scale for a high number of material types.

Our conclusion about this part of the thesis is that the proposed algorithm works, but more tests should be done and it definitely must be made more stable. Furthermore, the process of carving and revealing the stratified nature is really slow: less than 10 minutes were needed to spread the material in the lower part, but more than one hour and a half were spent to carve the canyon and reveal the stratified nature of the scene. This, in theory, can be decreased by modifying the value of k_s, the dissolution constant inside the fluid simulation. However, we did not try this approach because it did not seem physically plausible since we are eroding more sediment than we are depositing.

7.3.2 Visualization Algorithms

Figure 7.14 compares the results of the three algorithms that we developed to visualize the sedimentation phenomenon on mock-up data. The mock-up data were generated in a similar way to how the initialization data were generated for the upper part of the scene. In this case we run two tests: one with a new sedimentation layer every unit length and one every 0.5 unit length, thus doubling the frequency.

As it is possible to see in Figure 7.14, in the first row, the Nearest-Point algorithm is not well suited for this type of scenario. The second and third algorithms are, visually speaking, comparable. In the second row it is possible to see the same three algorithms with data generated every 0.5 unit length: the only one that is still able to deliver a decent visualization of the data is the NPTS. We believe that the third algorithm failed due to its unstable nature, which we have already discussed in the previous chapter.

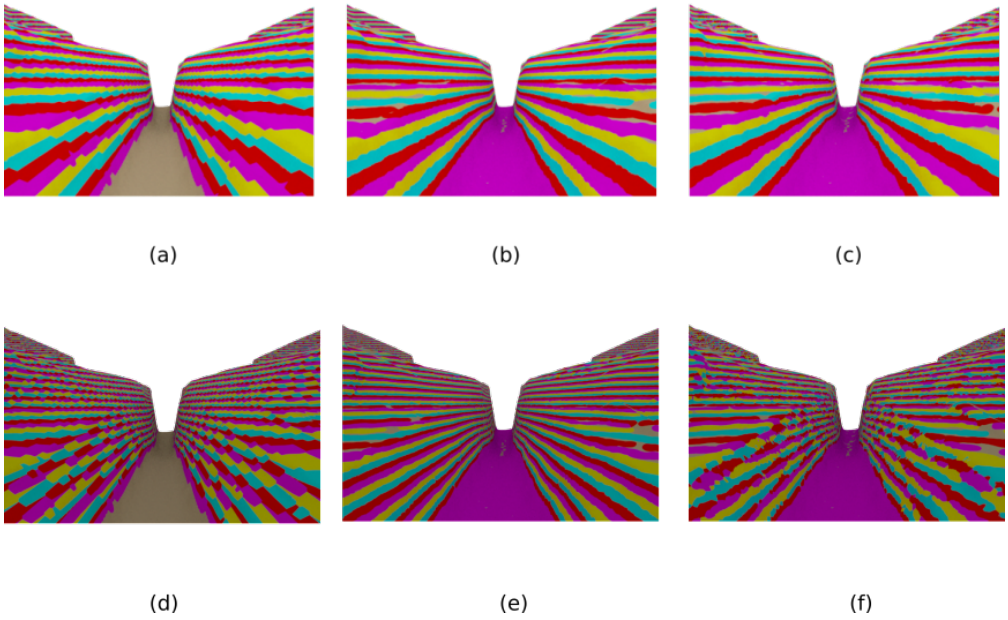


Figure 7.14: Results of the algorithms for visualizing the sediment layers, using mockup data. In the first row, the frequency of the mockup data was set to 1 unit-length, in the second row it was set to 0.5. (a) NP , (b) NPTS, (c)RBF , (d) NP, (b) NPTS, (c) RBF. The stabler algorithm is the NPTS.

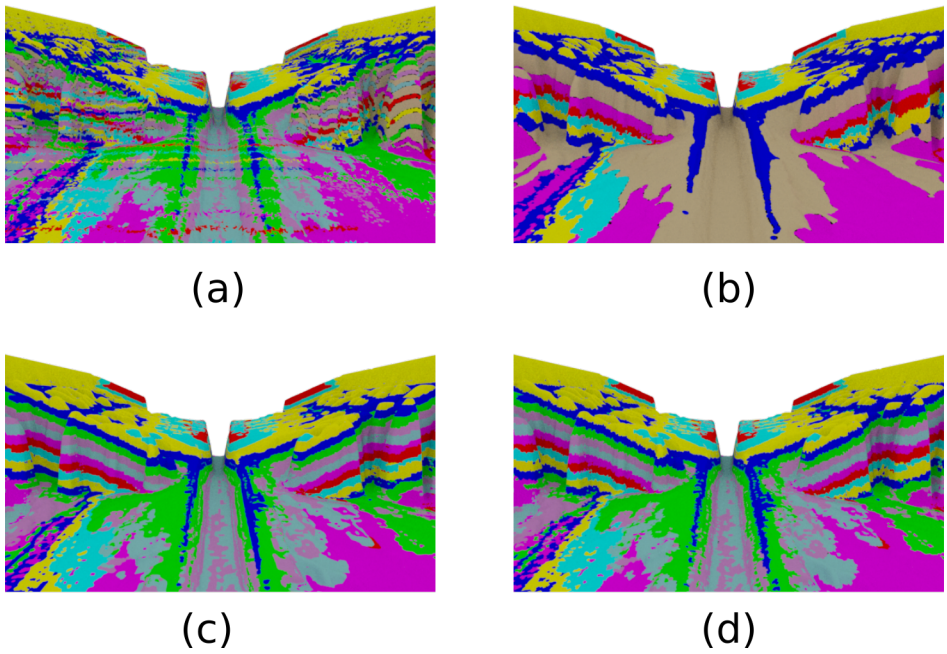


Figure 7.15: Visualization algorithms used on data computed through the simulation. (a) NP , (b) RBF approach ,(c) NPTS, (d) NPTS + Voronoi . In (d) we used a Voronoi pattern to "break" the smooth interface between different levels and give a more natural look.

Our conclusion about these algorithms is that a new stabler algorithm to visualize the interface between layers of different materials should be devised. Especially for thin layers, which in our case we were able to reproduce just by using mockup data. For this reason, a problem that should also be addressed in future work is the transport of multiple sediment material: only with a better and more precise simulation, in fact, it would be possible to generate thinner layers and then visualize them with a new visualization algorithm.

7.4 Combination of results

In this section we want to show the combination of the two shaders that we developed in the same scene, to prove that our framework can merge automatically, and with some good results, two different materials.

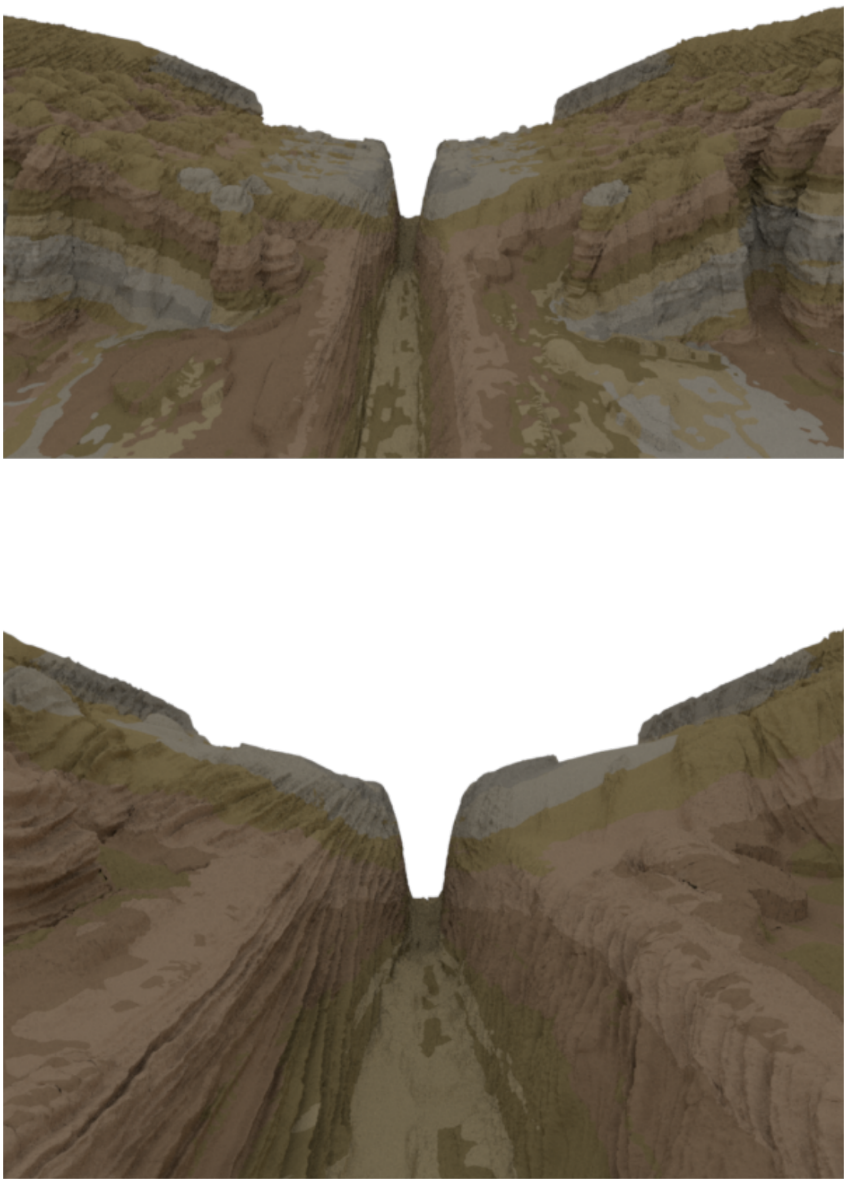


Figure 7.16: Combination of the two phenomena: the color is driven by the sedimentation shader, meanwhile the displacement is guided by the wind erosion.

7.5 Performance

In this section, we will provide the reader with some statistics about the memory and space needed to calculate all the various intermediate steps of this project. All our tests have been done on a laptop with Ubuntu 16.0 64-bit as Operative System, having 16GB of RAM and a Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz.

7.5.1 Framework

In this section, we will talk about the performance of our framework. We will start by analyzing the amount of time and memory spent on the computation of the LIC map for the first phenomenon.

subdiv	space	points	evaluation time	rendering time
1	505 MB	63k	6 mins	7:22 mins
2	884 MB	218k	40 mins	7:48 mins
3	2.3 GB	801k	4:20 h	8:25 mins

Table 7.1: Statistics for the LIC algorithm. In the first column the number of subdivision steps is shown. In the second one the memory space used during the computation. The third column shows how many secondary points have been generated. The fourth column stores the time needed to complete the evaluation of the LIC algorithm, and the last column shows the time needed to render the various images.

As it is possible to see in Table 7.1, the time spent to calculate the appearance data increases with the number of subdivision. The number of used points almost quadruplicate at each new iteration used, which was expected since we are using the Catmull-Clark methods, that at each iteration generates four vertices per face. An interesting and unexpected result is the rendering time: although the size of the pointclouds quadruplicates at each iteration, the time spent increases only by 30 secs.

For the sedimentation phenomenon we will show similar statistics only for the NPTS algorithm.

Overall the processes that we are describing are slow and memory consuming and in future works this might be improved.

subdiv	space	points	evaluation time	rendering time
1	979 MB	266k	19 secs	4:39 mins
2	2.6 GB	864k	1:07 mins	4:46 mins
3	9.4 GB	31136k	5:05 mins	5:05 mins

Table 7.2: Statistics for the NPTS algorithm. In the first column the number of subdivision steps used is shown. In the second column the memory space used during the computation.In the third column it is possible to see how many secondary points have been generated. The fourth column stores the time needed to complete the evaluation of the NPTS algorithm, and the last column shows the time needed to render the various images.

Conclusion

As we stated in the introduction, our goal for this thesis was to demonstrate that it is possible to use simulation data to drive the procedural generation of shader textures and displacement.

We set up a new, general framework able to handle three sub-problems:

- Reading of the file inputs
- Classifications of vertices and computation of the appearance data
- Writing of output files

In the foremost, we read the mesh and the data associated with each vertex. The second point of the list represents the main bulk of our framework. During this phase, the vertices of the mesh are classified, and their appearance data are computed. The output of this phase of classification and computation will be outputted in the form of two pointclouds for each phenomena that is present in the scene. One pointcloud is used as a mask, while the other to store the appearance data. The framework also outputs a scene with a shader network already built in. While we were working on the framework we also started to research interesting phenomena that we thought would have been possible to describe with our framework. We ended up selecting the wind erosion and the sedimentation. Once the phenomena that we wanted to simulate were decided we started to devise how to collect the data needed by our framework. This led to the implementation of two algorithms: a simple approximation of the wind flow and a new algorithm for the sediment transport that allows the deposition and transport of different materials. Parallel to this work, we tried different approaches how to visualize the data collected in the simulation. For the wind erosion we tried two algorithms: the first involved the creation of a stack of planes, that in the end was discarded in favor of the second algorithm: the Line Integral Convolution. For the sedimentation phenomenon we tried three different algorithms that we first tested on mockup data and than later with data from the simulation.

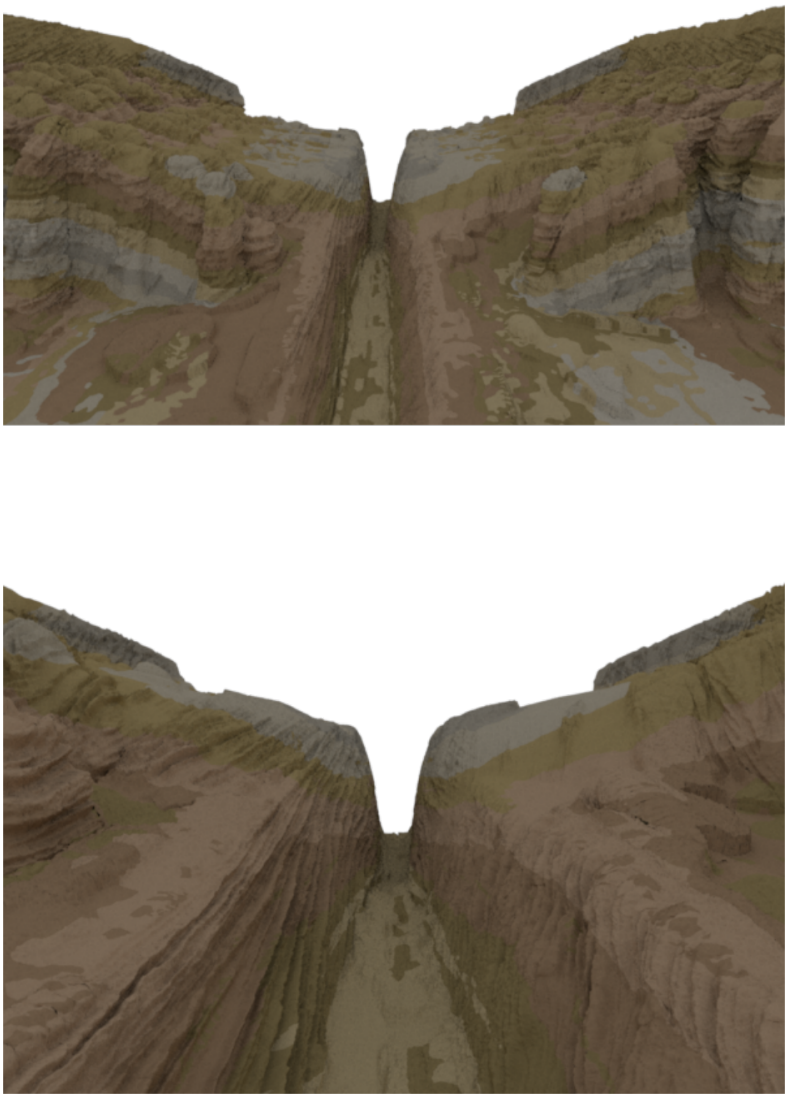


Figure 7.17: Example of the final result that is possible to achieve with our framework.

In Figure 7.17 it is possible to see one of the results that we managed to obtain through our project. We believe that we have accomplished our goal in demonstrating that it is possible to derive shaders from the data calculated during the simulation. The amount of details provided, if compared to the initial geometry, is significant. As it is now, the time needed to calculate the appearance data is rather high, especially for small details, and it cannot be used to speed up the creative process. Nevertheless, we believe that with the right improvements this project can be be used to decrease the time needed by the artist in developing the final scene.

Future Work

Many of the possible expansions and future works were already described in the previous chapters. Nevertheless, we will recall them here, as well as present some new ones.

Wind Erosion

For the wind erosion, we believe that two big improvements must be done: firstly an automatic selection of parameters and secondly an investigation on how to allow the sedimentation data to influence the final output, thus approximating in this way the correlation between the sediment layers and the erosion caused by the wind. There are also secondary tasks that should be addressed: a faster and more memory-efficient implementation should be devised, a better selection of the vertices during the classification step and finally provide an automatic way to solve the error due to the displacement map. Another possible improvement could be to devise a more sophisticated wind simulation.

We think that this technique also has the potential of becoming a texture that could be used by an artist during their tasks. To do so, in our opinion, two more steps are required. Firstly, we would like to give the possibility to the user to paint over the noise texture that will be used in the first LIC pass, allowing in this way a better control on which texture should be convoluted with the flow field. Secondly, it would make sense to let the user define the direction of the flow vector through some GUI tool. With this feature, the user would be able to define where the flow of the texture will go. The aim of these two improvements would be to give more freedom to the user, a requirement for every texture widely used in the industry.

Sedimentation Layers

The sedimentation layers phenomenon has essentially two items which should be improved: the simulation and the visualization algorithms. As we have already discussed in the previous chapter, the simulation that we are proposing should be tested against a more physically plausible one. Furthermore, it should be made more stable. As it is now, the simulation is not able to create stacks of thin layers. This problem should be addressed, in parallel with the issue of visualization of these thin stacks. The visualization techniques that we developed, in fact, should be revisited and made more stable for thin layers. We believe that modelling this issue as a minimization problem is the key to solving it.

We would like also to discuss here some general improvements that could be interesting to implement in future works.

As explained in the previous chapter, it would be interesting to put the result

of the 3D point cloud in a 2D texture and then use this texture in the render. Another interesting improvement that should be investigated is different ways of blending the various masks generated by the classifiers. As it is now, it is only possible to do a smooth transition between one material to another, when in theory it is also possible to do a harsh transition: by simply putting more and more points along the interface of the two materials. But maybe some other elegant, and possibly less costly in terms of memory-space, solutions are available.

During the development of our thesis, we also thought that a future evolution of this project would be to use it to train a generative neural network. In the future it might be interesting to generate a 2D texture just by knowing the geometry of the scene.

This would require having a data-set that stores geometry and appearance data. For now, there are two methods: capturing the data from the real world or asking an artist to create many scenes. The problem with these methods is that they cannot easily generate, in a reasonable amount of time, enough scenes. One of the main problems in training a neural network is that a huge data-set must be used in the learning process. With our approach, instead, the data set can be generated by running different simulations. If the data from the simulations are correct, and the algorithms used to visualize these data are correct as well, then it might be interesting to train a generative neural network using the renders from the landscape and the geometry as data-set for this problem.

Bibliography

[1] Fast Noise. URL: https://github.com/Auburns/FastNoise.

[2] mathtoolbox. URL: https://github.com/yuki-koyama/mathtoolbox.

[3] Open Mesh. URL: https://www.graphics.rwth-aachen.de/software/openmesh/.

[4] Point Cloud Library. URL: https://pointclouds.org/.

[5] Terrain Erosion. URL: https://github.com/karhu/terrain-erosion.

[6] M. Beardall, M. Farley, D. Ouderkirk, C. Reimschussel, J. Smith, M. Jones, and P. Egbert. Goblins by spheroidal weathering. In *Proceedings of the Third Eurographics Conference on Natural Phenomena*, NPH'07, page 7–14, Goslar, DEU, 2007. Eurographics Association.

[7] Bedrich Beneš and X Arriaga. Table mountains by virtual erosion. In *Proceedings of the First Eurographics conference on Natural Phenomena*, pages 033–039. Eurographics Association, 2005.

[8] Bedřich Beneš and Rafael Forsbach. Visual simulation of hydraulic erosion. 2002.

[9] Bedřich Beneš, Václav Těšínskỳ, Jan Hornyš, and Sanjiv K Bhatia. Hydraulic erosion. *Computer Animation and Virtual Worlds*, 17(2):99–108, 2006.

[10] Jiří Bittner, Michal Hapala, and Vlastimil Havran. Incremental bvh construction for ray tracing. *Computers & Graphics*, 47:135–144, 2015.

[11] Brian Cabral and Leith Casey Leedom. Imaging vector fields using line integral convolution. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, page 263–270, New York, NY, USA, 1993. Association for Computing Machinery. doi:10.1145/166117.166151.

[12] Nuttapong Chentanez and Matthias Müller. Real-time simulation of large bodies of water with small scale details. In *Symposium on Computer Animation*, pages 197–206, 2010.

[13] Guillaume Cordonnier, Eric Galin, James Gain, Bedrich Benes, Eric Guérin, Adrien Peytavie, and Marie-Paule Cani. Authoring landscapes by combining ecosystem and terrain erosion simulation. *ACM Trans. Graph.*, 36(4), July 2017. doi:10.1145/3072959.3073667.

[14] David S Ebert, F Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing I& modeling: a procedural approach*. Morgan Kaufmann, 2003.

[15] Eric Galin, Eric Guérin, Adrien Peytavie, Guillaume Cordonnier, Marie-Paule Cani, Bedrich Benes, and James Gain. A review of digital terrain modeling. In *Computer Graphics Forum*, volume 38, pages 553–577. Wiley Online Library, 2019.

[16] T. Ito, T. Fujimoto, K. Muraoka, and N. Chiba. Modeling rocky scenery taking into account joints. In *Proceedings Computer Graphics International 2003*, pages 244–247, 2003.

[17] Balázs Jákó and Balázs Tóth. Fast hydraulic and thermal erosion on gpu. In *Eurographics (Short Papers)*, pages 57–60, 2011.

[18] M. D. Jones, J. Butler, M. Farley, and M. Beardall. Directable weathering of concave rock using curvature estimation. *IEEE Transactions on Visualization I& Computer Graphics*, 16(01):81–94, jan 2010. doi:10.1109/TVCG.2009.39.

[19] B Mandelbrot. How long is the coast of britain. *Science*, 156:636–638, 1967.

[20] Benoit B. Mandelbrot and Benoit B. Mandelbrot. *The fractal geometry of nature / Benoit B. Mandelbrot.* W.H. Freeman New York, updated and augmented [ed.] edition, 1983.

[21] X. Mei, P. Decaudin, and B. Hu. Fast hydraulic erosion simulation and visualization on gpu. In *15th Pacific Conference on Computer Graphics and Applications (PG'07)*, pages 47–56, 2007.

[22] M Milligan. Geology of goblin valley state park, utah. *Geology of Utah's Parks and Monuments*, pages 421–432, 2003.

[23] F. K. Musgrave, C. E. Kolb, and R. S. Mace. The synthesis and rendering of eroded fractal terrains. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '89, page 41–50, New York, NY, USA, 1989. Association for Computing Machinery. doi:10.1145/74333.74337.

[24] F. Kenton Musgrave, Craig E. Kolb, and Robert S. Mace. The synthesis and rendering of eroded fractal terrains. In *SIGGRAPH '89*, 1989.

[25] Kenji Nagashima. Computer generation of eroded valley and mountain terrains. *The Visual Computer*, 9(13):456–464, 1997.

[26] Benjamin Neidhold, Markus Wacker, and Oliver Deussen. Interactive physically based fluid and erosion simulation. volume 5, pages 25–32, 01 2005. doi:10.2312/NPH/NPH05/025-032.

[27] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, July 1985. doi:10.1145/325165.325247.

[28] A. Peytavie, E. Galin, J. Grosjean, and S. Merillou. Arches: a framework for modeling complex terrains. *Computer Graphics Forum*, 28(2):457–467, 2009. URL: https://onlinelibrary.wiley.com/

doi/abs/10.1111/j.1467-8659.2009.01385.x, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2009.01385.x, doi:10.1111/j.1467-8659.2009.01385.x.

[29] Andrew Selle, Ronald Fedkiw, ByungMoon Kim, Yingjie Liu, and Jarek Rossignac. An unconditionally stable maccormack method. *J. Sci. Comput.*, 35:350–371, 06 2008. doi:10.1007/s10915-007-9166-4.

[30] Ondrej Stava, Bedrich Benes, Matthew Brisbin, and Jaroslav Krivanek. Interactive terrain modeling using hydraulic erosion. *Symposium on Computer Animation*, pages 201–210, 07 2008.

[31] Ondřej Šťava, Bedřich Beneš, Matthew Brisbin, and Jaroslav Křivánek. Interactive terrain modeling using hydraulic erosion. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 201–210, 2008.

[32] Luther A Tychonievich and MD Jones. Delaunay deformable mesh for the weathering and erosion of 3d terrain. *The Visual Computer*, 26(12):1485–1495, 2010.

[33] Jarke J. van Wijk. Spot noise texture synthesis for data visualization. *SIGGRAPH Comput. Graph.*, 25(4):309–318, jul 1991. doi:10.1145/127719.122751.

[34] Korneliusz Warszawski, Sławomir Nikiel, and Marcin Mrugalski. Procedural method for fast table mountains modelling in virtual environments. *Applied Sciences*, 9:2352: 1–20, 06 2019. doi:10.3390/app9112352.

List of Figures

1 General flow chart of the entire process of the thesis. The three macro-blocks corresponds approximatevely to chapter 3,4,5. . . . 8

1.1 National Park, Utah 11

1.2 Example of a sedimentation process. (a) The particles of one material are transported by the flow and are deposited near the river banks where the flow is calmer. (b) Particles of a second material are now the main material transported by the flow and they are deposited over the first layer. (c) The same process in (b) happens again, leading to the creation of a new layer. (d) The third layer is completed, and a new layer is going to begin again. 11

2.1 Image from the film: "Guardians of the Galaxy". The alien structure in the picture has been created using 3d fractals. 15

3.1 General flow chart of our framework 20

3.2 Flow chart of the classification module. Vertices and data are read, then each classifier uses these inputs and the shared data to select the appropriate vertices. The data computed that are stored through the abstract class *AShader*. The outputs of the classifications phase are then used to update the shared data. The cycle goes on until all the classifiers are used. 24

3.3 Example of a shader network generated automatically. The shaders presented in the list are used to instantiate two blocks: one for the shader and one for the mask. The shaders are then combined using the mask to weigh the blending between the various nodes. This is done until the last element in the list is added to the network. A similar scheme is used to calculate the displacement height. . . 28

4.1 The big dark-blue arrows represent the direction of the wind imposed by f. The light-blue arrows represent the local flow of the wind influenced also by the geometry of the canyon. 31

4.2 Modified sediment transport. In this example the material id at point p is selected by looking at p'. The position of p' is known by inverting the direction of v, the velocity of the fluid at p, and multiplying it by dt. Once p' is known, we calculate the weights w_1, w_2, w_3, w_4. Each weight is equal to the inverse distance between the corner and the point P'. Then we sum w_1 with w_2 and w_3 with w_4 since they have the same color id. Then we select a random number x between 0 and 1, if $x < w_1 + w_2$ then P will transfer the yellow material, otherwise the red. In this example the yellow material has been selected. 33

4.3 Example of a creation of a new sedimentation layer. (a) The green sediment deposits at x_0, y_0. Since $z' < z + t$ the green sediment is stored and its counter is increased by 1. (b) Another sediment comes, but it has a different material id, therefore the counter is decreased by 1. (c) Another particle come to position x_0, y_0, the counter was at zero so the new sediment is accepted as the new active sediment. (d) The counter is increased by 1 because a particle with the same id has arrived. (e) The situation in d is replicated until the counter is superior to a certain threshold. (f) A new layer is created with the id of the previous active layer, the counter is set to zero and the active id is set to a void value, ready to repeat the process for the new incoming particle. 36

5.1 (a) Parco Nazionale di Belluno ,(b) Triestinen Karst Both images are examples of a karst phenomenon knows as Rillenkarren. Karst phenomena happens when acidic rains corrode limestone or dolomite bedrock. 37

5.2 Example of stack of planes algorithm. The point to be shaded p is projected along the direction of v_{orth} finding p_1. The sin function is used to define whether or not the projected point is inside or outside the stack of planes. In this case, $sin(p_1)$ is positive, hence the point p is selected to be part of the stack of lines that have direction v. 39

5.3 (a) Errors caused by interpolating the value of v , (b) Same situations of (a) but with a lower frequency.(c) Result with the cellular automata approach, the output is better but when we decrease the scale of the details we obtain (d). As it is possible to see in figure (d) there is a general sense of flow, but the lines are not always smooth and it is possible to see the interface between cells. . . . 39

5.4 General diagram of our implementation of the LIC for our purposes. 41

5.5 Visual representation of how the data are structured for our algorithm. The red points are the vertices present in the coarse mesh. The dark green arrows represent the flow normal and the blue points the secondary points. Only the red vertices with a flow normal are inserted in the kd-tree and in the input_list. The secondary points are generated only between points that will be put in the kd_{input}. The red points that are inside the kd_{input} are also put in the list of secondary_ points. 43

5.6 Note how the vertices found in the path with a smaller step size (green) are spatially closer together with respect to the ones found with a larger step size (orange). Since the green points are closer together, the evaluation of the noise function at those points will differ a bit. This means that the result of the convolution at point p_1 will be close to the result obtained at point p_2. This is because the values involved in the convolution are similar. If we used the orange paths, the discrepancy in the results will be bigger. Note also how the frequency of the noise texture plays a relevant role: if the frequency is low, the result of the convolution at point p_1 and p_2 using the orange path will be very similar because the evaluation at the points in the orange path will be very similar since the noise function will change slowly. 44

5.7 Visual representation of the data structure for the sedimentation phenomenon. 47

5.8 Results on the mockup data with the three visualization algorithms (a) NP (Nearest Point). (b) NPTS (Nearest Point To Surface). (c) RBF approach 48

5.9 Visual representation of the Nearest point to surface algorithm. Secondary points are created around the surface. The material id of the secondary points is taken from the closest intermediate sedimentation point stored in each vertex of the input mesh. The points are then smoothed, but this is not represent in the picture 49

5.10 Visual representation of the Nearest point to surface algorithm. Secondary points are created around the surface. The material id of the secondary points is taken from the closest intermediate sedimentation point stored in each vertex of the input mesh. The points are then smoothed, but this is not represented in the picture 50

5.11 Visual representation of the key idea at the base of the minimization algorithm. Every sedimentation layer has a lower and upper edge, and we can use those edges to reconstruct the function that encloses the sediment layer. In this case, since p lies between the maximum and the minimum functions defined by $MaxLayer_3$ and $MinLayer_3$, p will be associated with $id = 3$. 51

5.12 In this scenario, the point p is enclosed between the purple and the red layers. Since our utility functions favour layers that are close to p, the purple layer will be selected. The solution that we propose is to penalize layers that are constituted by points with higher variance. In this case, since the red points are aligned, their variance is minimum, so the purple id is penalized and the red id, the correct one, will be selected. 52

7.1 Scene that will be used for the simulation of both phenomena, only the simulation will change between one scenario and the other. . . 56

7.2 Images of the canyon viewed from the beginning and the end. . . 57

7.3 (a) Antelope Canyon,Arizona (b) Wave Coyote, Arizona 57

7.4 (a) Subdivision level = 1, (b) Subdivision level = 2, (c) Subdivision level = 3 59

7.5 (a) Final results with colors, (b) Geometry used in the final result,(c) Geometry of the input mesh 59

7.6 Comparison of the various techniques that can be used to add more details to the scene. In the first row the material is colored in a grey-scale to better highlight the subtler details of the displacement map. In the second row, we show the final result. (a) only LIC , (b) LIC + DDA , (c) LIC + DDA + Fbm , (d) only LIC , (e) LIC + DDA , (f) LIC+DDA+Fbm. The results with the LIC are the smoothest. The DDA is used to add minor details using the flow normal. Finally, the Fbm is applied to add further details. 60

7.7 Example of error due to a wrong classification of vertices, an aspect that should be improved. The blue arrows represent the direction of the wind, as it is possible to see the air "fall down" like a fluid, this because our simulation of the wind is rather imprecise. The fact that the cliffs are selected to be part of the wind erosion is not an error of the wind simulation but from the classifier: the vertices that compose the cliffs have associated with them a flow normal, and for this reason they are selected without checking if the air would have enough strength to erode a large portion of the landscape. 61

7.8 Disposition of the various sediment strata that are initialized before the simulation. Starting from each vertex's height a new layer is created after h units length until reaching the base of the scene. Note that we have a fixed number of material ids so after a certain number of iterations the material ids are re-used. 62

7.9 (a) After the first part in which the rain will transport the sediment from the upper part to the lower part of the scene, a ramp will be created between these two zones. (b) The river that will carve the ramp revealing its sedimentary structure. The rain stops after a fixed amount of time, and during this time a ramp that connects the two parts will be formed. The ramp is then carved by a river, in order to reveal its inner structure. 63

7.10 On the right, the pre-visualization of the sediment data. On the left, the pre-visualization of the geometry. 64

7.11 Note how the disposition of the sediment in the zone is replicated in the zone b. The orange bands are added to highlight the edges of the sediment layers. This is evidence that suggests the transport of the various sedimentation materials. 64

7.12 The images show the evolution of the river banks and the slow revealing of the underlying stratified structure. Figures a-b-c depict how the blue sediment is removed showing the green sediment. In figures d and e, the light purple sediment is revealed by consuming the green sediment. Finally, in f we can see how the light blue sediment is, once more, taking the place of the upper layer (light purple). 65

7.13 Example of how the various sediment types slowly spread in the scene. (a) The material id_1 is uncovered due to the erosion, since it is visible in the debug tool, it should also start to spread the sediment. (b) The material id_1 starts to spread, but only the zones near its starting point, in the lower part the material id_2 starts to spread as well. (c) the situation is the same in (b), what we would have expected though was to see id_1 spreading also in the lower part of the scene, since the water flow in that direction but that did not happen. 66

7.14 Results of the algorithms for visualizing the sediment layers, using mockup data. In the first row, the frequency of the mockup data was set to 1 unit-length, in the second row it was set to 0.5. (a) NP , (b) NPTS, (c)RBF , (d) NP, (b) NPTS, (c) RBF. The stabler algorithm is the NPTS. 68

7.15 Visualization algorithms used on data computed through the simulation. (a) NP , (b) RBF approach ,(c) NPTS, (d) NPTS + Voronoi . In (d) we used a Voronoi pattern to “break” the smooth interface between different levels and give a more natural look. . 68

7.16 Combination of the two phenomena: the color is driven by the sedimentation shader, meanwhile the displacement is guided by the wind erosion. 70

7.17 Example of the final result that is possible to achieve with our framework. 74

List of Tables

7.1 Statistics for the LIC algorithm. In the first column the number of subdivision steps is shown. In the second one the memory space used during the computation. The third column shows how many secondary points have been generated. The fourth column stores the time needed to complete the evaluation of the LIC algorithm, and the last column shows the time needed to render the various images. 71

7.2 Statistics for the NPTS algorithm. In the first column the number of subdivision steps used is shown. In the second column the memory space used during the computation.In the third column it is possible to see how many secondary points have been generated. The fourth column stores the time needed to complete the evaluation of the NPTS algorithm, and the last column shows the time needed to render the various images. 72