University College London

MSc Project Dissertation

Interactive Design and Fabrication of Table-top water Fountains

Gilles Rainer
supervised by
Dr. Tim Weyrich

September 4^{th} 2015

Abstract

The goal of this project was to create an interactive system that allows users to design small water fountains digitally that they can then fabricate through 3D printing. Ideally, the user should provide a rough topology for the fountain and a pattern for the flow of water. The system would then adapt the structure of the fountain to match the water behaviour required by the user.

The work hence drew on multiple areas of Computer Graphics: interactive 3D modelling, fluid simulations, and performative design. The goal was to combine all of these into a single system. Implementing a modelling interface was not needed as many programs already exist for this, the water was simulated using the Shallow Water equations, and the optimisation relies on the Simplex Downhill algorithm ([NM65]). After designing several test cases, we 3D printed one of those water fountain models and created a functioning setup with an aquarium pump and a PVC tube. The behaviour of the real water fountain validates the results obtained with the system.

Acknowledgements

I would like to acknowledge the following, without whom this thesis would not have been completed:

- Tim Weyrich, my supervisor, for supporting and guiding me through a project that I was not confident in finishing at first.
- Jacques Cali, who assisted me for the 3D printing and helped me take pictures of the final water fountain setup.
- My family and friends, for their unconditional love and support.

Contents

List of Figures

Chapter 1

Introduction

Luxurious sports cars, yachts, planes, complex architectural buildings etc. all have in common the two essential needs to satisfy an artist's vision and design as well as physical and functional constraints. Their creation traditionally follows an iterative process of compromising between the artistic aspirations of the designer and the physical limitations and uses the object needs to respect. Architects used to work in collaboration with engineers so that both the artistic and technical aspects of the project were taken care of. However, we are now at a turning point, where computer-assisted design is becoming a more and more accessible and widely used strategy for many different types of complex object fabrications. Computer-assisted design systems allow designers and artists to model their creative intentions and concepts digitally and get an immediate feedback from a virtual simulation. Indeed, the system will calculate the physical constraints the object would be exposed to in order to assess whether the design is sustainable or not, and hence prevent the fabrication of physically invalid designs. Some systems will then even optimise the designer's input by slightly altering and correcting parts of it in order to satisfy the physical constraints while staying as close as possible to the original concept.

The considerable advantage of this computer-assisted method compared to the previous back and forth between artists and engineers is its speed, flexibility and efficiency: The artist can play around with concepts and de-

signs, fully exploit his creativity without filtering anything, receiving a direct feedback from the system based on millions of computations, without even needing an in-depth understanding of the physical constraints. This computer-aided conception of objects in respect to their use is referred to as "performative design".

The aim of this Master's project is to explore this area for one particular type of objects: We will investigate the creation of table top (tiny) water fountains – from the digital conception and optimisation to the actual physical fabrication. Performative design systems already exist for a number of objects, from car engines to paper airplanes ([UKSI14]). In this sense, one could argue that a table top water fountain design system would not be very different from most other performative design systems out there. However, the particularity of this project is that, through the water fountain object, it integrates various disciplines of Computer Graphics to the problem. While the objective for an architectural design system for instance would be to calculate and optimise its exposure to gravity, weather and material constraints etc. (example in [Man10]), the objective of a performative design system for table top water fountains will be to adjust the flow and behaviour of the water to the user's concept. The idea would be to allow the user to model a fountain and specify where the water should flow, and then let the system simulate the physics and adjust the geometry in order to make the user's concept come true.

It hence links the classical topic of fluid simulations to digital sculpting, to finally apply optimisation tactics on the fountain geometries. This makes it a very interesting project with a wide scope of background literature, but at the same time very tricky and perplexing, since each of these topics already represents a very prolific research area by itself. Indeed, the biggest challenge working on this project has been to clearly outline the extent of each part, and the amount of work and research that should go into every one of these. It was crucial to find competent solutions early, in order to rapidly obtain a prototype that has a working implementation for each part and balance the work put into the user input, the water simulation and the optimisation.

One of my aims approaching this problem was to find out more about

water simulations, as they play a huge role for computer-generated environments (for instance in video games or in visual effects for movies), and implement one myself. Physical simulations and renderings play an important part in Computer Graphics and fluid simulations have since the beginning been amongst the biggest areas of research, adapting physical laws and calculations to the structures and schemes used in Graphics systems, in order to create something realistic and aesthetic. Although the initial description was quite vague and there were no prior constraints or requests on the language to write it in, the science behind it or even the methods to use, another clear aim of the object was to create a global, finished and somewhat self-contained system. All the elements of a performative design system should be present in the final prototype.

From the beginning of the project, we decided to focus most of the efforts on the optimisation part, rather than on a fancy user interface. The objective for the design of the system was to allow the user to give a rough model or concept of the shape of his desired fountain as input, along with a specification of where the water should flow. The system would then run a fluid simulation in order to retrieve the flowing pattern of water and assess whether it matches the user's wishes. This would constitute the basis to run an optimisation algorithm that would correct the topology of the fountain. Finally, the user would retrieve the result of the optimisation in an appropriate format to fabricate the fountain through 3D printing (since table top fountains are rather small).

Resulting from the concept for the system, the project work was organised in three phases: I first developed and implemented a simple model for the water simulation and made sure it worked realistically in all the cases needed on water fountains (water flowing out of the source and off the sides, taking into account the shape of the fountain...). Once this worked well enough, I tried out different optimisation strategies, one of which I implemented myself in order to have more control over the optimisation process. Finally, in the last phase, I tested the system, produced results and wrote a script to convert the geometries to meshes. This made it possible to 3D print the input and output fountains and assess whether the results produced by the system were

actually physically accurate.

In a first step, Chapter 2 will provide a context to the project and an overview on the related work that has been done in the field. We will then analyse the design strategies we considered for the system in Chapter 3, followed by a more concrete explanation of the implementation process and tactics in Chapter 4. Chapter 5 will detail how the system was tested and validated. In Chapter 6, we will display the results we were able to obtain and how the system can be used to create real-world water fountains. Finally, Chapter 7 will summarise the project and provide ideas for further work and features that could be added if the project was to be continued.

Chapter 2

Background

The background research for the project naturally divided itself into three core areas: 3D modelling interfaces, fluid simulations, and optimisation strategies for performative design systems. Even though these seem to have little to do with each other, they are three of the biggest topics in Computer Graphics, and even if some of the techniques we discovered were not used in the final system, the research and investigation was still beneficial to the project.

2.1 Freeform Sculpting Interfaces

Digital sculpting of 3D objects is a very challenging field in Computer Graphics. The problem of allowing the user to sculpt three-dimensional objects on a two-dimensional surface (computer screen) using a mouse is very hard to resolve. Even though many different interfaces and techniques have been invented and implemented (a broad panorama of techniques is presented in [Ber13]), none of them are really able to cope with the challenges of virtual 3D modelling perfectly.

One interesting approach to digital free-form sculpting was implemented in the "FiberMesh" project ([NISA07]). Even though the name "sculpting" might imply otherwise, a very efficient approach is simplifying the problem by reducing its dimension. The interface allows the user to draw 2D sketches

from any specified view, using different types of lines. After every new contour, the system automatically computes the 3D topology of the object by inflating the 2D lines to 3D surfaces using the parametric curves drawn by the user.

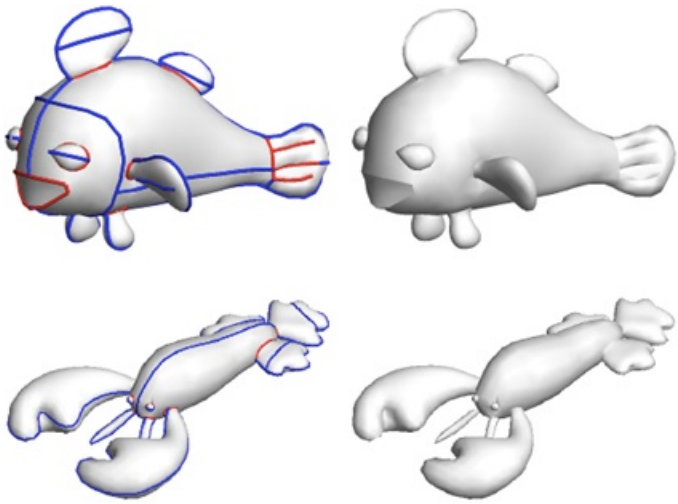

Figure 2.1: Some results generated with the FiberMesh system ([NISA07]). The user sketch is shown by the coloured lines: The blue curves are smooth curves, the red ones mark discontinuities.

Even if the automatic transition from 2D to 3D performed by the system is not always entirely true to the intention of the user, this method of sculpting through sketching is, for now, one of the more efficient ways to design 3D objects from scratch, and quite a lot of research is carried out in that area ([EBC+15]).

For this project however, implementing such a complex free-form sculpting system seemed too complicated and time-consuming, knowing that using such a complete solution would also be disproportionate in relation to the other parts of the problem. This is why we decided to at least postpone the implementation of a sculpting tool or interface, and maybe even completely skip it if there was not enough time left. Test cases for the rest of the system could be easily generated choosing an already existing sketching or 3D modelling tool, out of the variety of 3D processing programs (such as BLENDER[1]

[1]Blender is a free and open source 3D creation suite. It supports the entirety of the 3D

for instance). There would have been no point in re-implementing a sculpting tool unless it had any really novel features, which was not essential to this project. This, however, only motivated me further to implement my own fluid simulation (rather than use an existing package).

2.2 Fluid Simulation in Computer Graphics

Fluid simulation has been one of the biggest areas of research in Computer Graphics for a long time. The state of the art is impressive: From off-line computations for photorealistic results to real-time animation and interaction, lots and lots of techniques, approaches and code have been generated over the years. Generally, they can all be placed into three categories: procedural animation, particle simulation and height fields (good panorama of fluid simulation techniques is given by Matthias Müller-Fischer in [MF08]).

2.2.1 Overview of Fluid Simulation Techniques

Procedural water animation is mostly used in real-time rendering of water simulation for large unbounded fluid surfaces such as oceans. The final animation often consists of tricks that approximate the behaviour of the fluid without taking into account the particular conditions and constraints of the scene. This makes it a very good approach to simulate large quantities of fluids very quickly and efficiently (for lakes, waves in the ocean as in figure 2.2...), because it uses simple primitives (such as Fourier wavelets) to generate a rough animation of the fluid's behaviour. However, this technique does not take into account the interaction between the fluid and the rest of the scene, which is why the realism of the final results compares poorly to the next approaches.

The second method is based on the representation of the water as a height field. This is a common technique in Computer Graphics, as it makes it possible to reduce the three-dimensional simulation to a two-dimensional

pipeline—modeling, rigging, animation, simulation, rendering, compositing and motion tracking, even video editing and game creation.

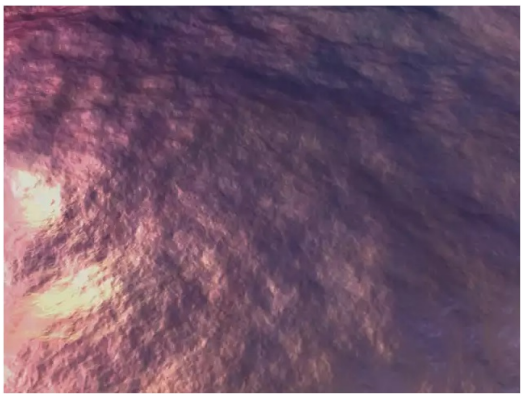

Figure 2.2: Rendering of a water body simulated with a procedural method. Retrieved from `http://www.sfdm.scad.edu/faculty/mkesson/vsfx319/wip/best_winter2004/ca301.3/palmer_sean/water/water.html`.

problem. The fluid's surface is simply represented as a 2D function h(x,y), where h symbolises the height of the water surface at that point in space. This considerably reduces the complexity of the problem, however it only works in the cases where there is only one height value per (x,y): It cannot simulate situations like breaking waves or objects sinking into a fluid.

Most height fields simulations are based on solving d'Alembert's wave equation for the surface of the fluid:

$$\Delta h(x, y, t) = \frac{\partial^2 h(x, y, t)}{\partial^2 t} \tag{2.1}$$

This amounts to modelling the surface of the water as a vibrating string (same equation) that has an elastic force that brings it back to its rest position. More complex cases can be simulated by involving other forces in the wave equation (depending on what happens in the scene). The general solution obtained for this equation is a sum of waves going in different directions, which generates the final pattern for the movement of the water surface.

Height fields are great for simulating fluid behaviour in constrained environments like ponds or rivers, where water has to move in a realistic way and to react to changes in the scene. However, very complex behaviours like splashing water or breaking waves cannot be modelled with this technique. While they take in much more physical information and precise constraints

than procedural animations, height fields still tend to approach the issue of fluid simulation as a global problem, modelling the fluids behaviour as a single surface. But as the water manipulation is the scene gets more complex and more violent, involving different forces and more interaction with other objects in the scene, the simulation has to take into account smaller fluid entities to be able to model it realistically. Since the constraints and conditions change very rapidly throughout different positions in space, the most precise approach to water simulation is to consider the fluid as a collection of small particles.

Particle simulation for fluids is optimal to animate splashes, water drops, small puddles etc. It is a very appropriate technique for situations where very complex water behaviour has to be modelled for relatively small quantities of water. This implies, of course, that it is not suited for oceans, rivers, etc. The particle representation also makes it difficult to keep track of the water as a surface, unlike in the case of height fields.

The science behind particle simulations is quite straight-forward (see detailed explanation in [Har04]). A fluid is represented by a velocity field u and a pressure field p, which both depend on space and time. Adapting Newton's second law to the case of incompressible, homogenous fluids (like water), we get the following Navier-Stokes equation for incompressible flow:

$$\frac{\partial u}{\partial t} = -(u * \nabla u) - 1/\rho * \nabla p + \nu * \Delta^2 u + F \tag{2.2}$$

ρ represents the density of the fluid, ν the kinematic viscosity and F corresponds to all of the external forces that are applied on the fluid. The viscosity term can be ignored under certain circumstances to make the simulation simpler.

Adding the nullity of the divergence of u, there are two equations to solve to obtain two unknowns, u and p. Since the aim of these simulations is to generate the behaviour of the fluid over time, incremental numerical approaches are used to solve these equations. The quantity we are most interested in is u, since, once it is known, the fluid's movements can be retrieved easily. A basic approach for this would be Euler's method (as developed in [Har04]): The

velocities are incremented according to the Navier-Stokes equation, which then makes it possible to update the positions of all the particles in the system by translating them by their velocity multiplied by the time step. It is a discretisation of a continuous reconstruction, so it necessarily approximates the solution. This means that the results can "explode" if for example the time steps dt are too big.

Most systems, in order to avoid this divergence of the calculations, resort to implicit solving methods (rather than explicit solvers). This approach was first adapted into a quick and efficient fluid solver by Jos Stam ([Sta99]). The key is to trace each point in the field backwards in time, to compute its new trajectory from its former position.

Figure 2.3: Example of a particle water simulation (shows the modelling with particles on the left, transformed into a fluid rendering on the right). Taken from [MCG03].

All of these methods have their advantages and drawbacks. It is thus in the hands of the programmer to judge which approach suits his application's needs best (there is an obvious trade-off between realism, complexity of the simulation and time of computation). For this project, it was essential that we find a good compromise between the complexity of the science used for the simulation and the physical accuracy. We finally chose to use the "Shallow Water Equations", which treat the water as a height field, but are based on an extension of the Navier-Stokes equation (hence a sort of intermediate approach between the height fields and the particle approaches developed earlier).

2.2.2 The Shallow Water Equations

The Shallow Water Equations, sometimes also referred to as Saint-Venant Equations, are a system of Partial Derivative Equations modelling fluid flows for rivers, channels, ocean currents, tsunamis, or even atmospheric flows. Their main characteristic is that since the water is "shallow" (the depth is considered orders of magnitude smaller than the horizontal scale), so we allow ourselves to make approximations that don't take into account the vertical dimension.

To derive the shallow water equations (entire demonstration in the appendix of [TH]), we need to assume a few characteristics for the fluid (water) whose behaviour we want to simulate: We assume the fluid is incompressible and isothermal in order to use two simple principles to derive our equations: the conservation of mass and the conservation of momentum. Furthermore, for shallow water we assume that the velocity differences throughout the fluid body are fairly small, which means that the effect of viscosity can be ignored. Also, the numerical integration of this system of equations can contain a small amount of damping, which basically acts in the same way as the viscosity coefficient. We can thus assume an inviscid fluid. Finally we also ignore the friction of the water on the fluid bed.

Concretely, to derive the final shallow water equations (see entire formulation in [DM08]), we need to proceed to another very important assumption: The pressure is considered hydrostatic. This means that we can model the fluid as a height field with the pressure p acting as the sole vertical variable, and that the velocity is essentially a two-dimensional, horizontal vector field. It is possible to make this assumption because in shallow water, the vertical velocity of the fluid is considered much smaller by orders of magnitude than the horizontal velocities, which means that the pressure and gravity forces are keeping the fluid in an equilibrium along the vertical axis, hence the hydrostatic pressure assumption. Additionally, we neglect the Coriolis force, we consider the density of the water uniform and the velocity independent of the depth of water (constant over an entire water column). This is gives us the following equations that govern our shallow water system (as seen in

[CH14], only without the Coriolis forces):

$$\frac{\partial h}{\partial t} + \frac{\partial uh}{\partial x} + \frac{\partial vh}{\partial y} = 0 \tag{2.3}$$

$$\frac{\partial uh}{\partial t} + \frac{\partial u^2 h + gh^2/2}{\partial x} + \frac{\partial uvh}{\partial y} = -gh * \frac{\partial B}{\partial x} \tag{2.4}$$

$$\frac{\partial vh}{\partial t} + \frac{\partial uvh}{\partial x} + \frac{\partial v^2 h + gh^2/2}{\partial y} = -gh * \frac{\partial B}{\partial y} \tag{2.5}$$

In these equations, h represents the height (or depth) of the water colum at one point, u and v are the velocities along the x and y axes, g is the gravitational constant, and B is the height field of the ground topology (as seen in [Bro11]).

We hence obtain a classical partial derivative system, which we need to solve numerically. Countless numerical solvers exist, but it is important to choose one that balances between obtaining an accurate enough solution without spending too much time on computations. The classical numerical solver for partial derivative equations would be the explicit Euler method explained previously. Indeed, this method is very fast, but it always tends to deviate from the actual correct values the more time advances, because it uses the derivatives of the quantities it calculates as if they were constant along one time step. The solution here is to use intermediate calculations between time steps to retain more accuracy. The commonly used Runge-Kutta integrations (different orders of integration are possible) are based on this principle, however, in this case, we will use the Lax-Wendroff relaxation (a sort of two stage Runge Kutta) to compute a numerical approximation to the solution. Indeed, most implementations of Shallow Water Equations use this numerical solver (see [Mol]).

First of all, we can easily modify the previous equations to translate them into a classical partial derivative equation scheme: We introduce the following vectors:

$$U = \begin{pmatrix} h \\ uh \\ vh \end{pmatrix} \tag{2.6}$$

$$F(U) = \begin{pmatrix} uh \\ u^2h + 1/2gh^2 \\ uvh \end{pmatrix} \tag{2.7}$$

$$G(U) = \begin{pmatrix} vh \\ uvh \\ v^2h + 1/2gh^2 \end{pmatrix} \tag{2.8}$$

The Shallow Water Equations can thus be re-written in their hyperbolic form as:

$$\frac{\partial U}{\partial t} + \frac{\partial F(U)}{\partial x} + \frac{\partial G(U)}{\partial y} = \begin{pmatrix} 0 \\ -gh * \frac{\partial B}{\partial x} \\ -gh * \frac{\partial B}{\partial y} \end{pmatrix} = S \tag{2.9}$$

In a computer implementation, these quantities will all be defined discretely at every cell of a regular grid that divides space. $U_{i,j,n}$ for instance will represent the value of the quantity U at the grid cell (i,j) at the time step n. The Lax-Wendroff scheme we will base our simulation on, will divide each time step into two stages (as proposed in [Mol]). During the first half step, the values of U will be calculated for the time n+1/2 and at the midpoints between the cells (that is i+1/2 and j+1/2) using the following equations:

$$U^{n+1/2}_{i+1/2,j} = 1/2 * (U^n_{i+1,j} + U^n_{i,j}) + \frac{\Delta t}{2\Delta x} * (F^n_{i+1,j} - F^n_{i,j}) \tag{2.10}$$

$$U^{n+1/2}_{i,j+1/2} = 1/2 * (U^n_{i,j+1} + U^n_{i,j}) + \frac{\Delta t}{2\Delta y} * (G^n_{i,j+1} - G^n_{i,j}) \tag{2.11}$$

Once these half-step quantities are computed, the values of U for the next time step can be calculated using:

$$U^{n+1}_{i,j} = U^n_{i,j} - \frac{\Delta t}{\Delta x} * (F^{n+1/2}_{i+1/2,j} - F^{n+1/2}_{i-1/2,j}) - \frac{\Delta t}{\Delta y} * (G^{n+1/2}_{i,j+1/2} - G^{n+1/2}_{i,j-1/2}) + \Delta t * S^n_{i,j} \tag{2.12}$$

As explained in [CH14], $S^n_{i,j}$ is the source term that represents the influence of the topology of the sea bottom or the bed the water is flowing on. For additional precision, it can be replaced by $1/2(S^n_{i,j} + S^{n+1}_{i,j})$.

The Lax-Wendroff scheme works very well for the Shallow Water Equa-

tions, however it is known to have one drawback: It tends to amplify small oscillations resulting from noise in the calculations, and eventually the computed solutions will diverge even though, for quite a long time, the solutions behave correctly. Nevertheless, this problem can be prevented, as we will see later on, by adding a damping parameter (that is not initially contained in the Lax-Wendroff scheme).

The final aim of this project, however, is to use such a water simulation on the user-created water fountain sculpture, in order to match it to the specified flow pattern using an optimisation routine on the sculpture's topology. It is thus first and foremost an optimisation problem, which pushes it into the category of "performative design".

2.3 Performative Design Systems

There are systems that compute simulations in real time, based on the user's input design, to show how it performs, in order for the user to interactively change his design to match his performance requisites. However, the idea here is to consider this task as an inverse problem - have the system optimise the user's design automatically to match constraints that the user specified to the system in the beginning.

The previous literature on performative design systems is very vast and heteroclite, because design by optimisation can be used for virtually any real-world objects one would like to fabricate. It can range from performing calculations on architectural buildings to assess whether they sustain their own mass, high wind forces, or earthquakes, to creating elastic shapes and predicting how much they will bend under gravity ([PTC+15]), to creating puppets of bulls that run realistically using only the rotation of a motor as input ([TCG+14]). However, to my knowledge, nothing has been done so far for table-top water fountains.

The article [UKSI14] presents a very interesting example of performative design, as there are many parallels between their project and mine. Their system, "Pteromys", allows the user to first design a glider model using simple building primitives (basic paper airplane parts). They then present a

new method to efficiently simulate the effects of the air on the plane during flight. Their simulation is based on physical properties and equations and on experiments they performed, but it combines it all in a simplified simulation strategy, in order to facilitate computation and optimisation. Putting these constraints together, they come up with a cost function (energy function) to minimise in order to optimise the problem.

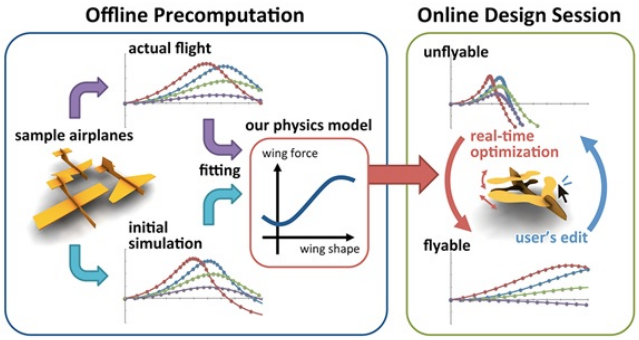

Figure 2.4: Diagram of the performative design pipeline of Pteromys ([UKSI14])

Their final results clearly show that their performative design system enables novice users, who are familiar neither with flight aerodynamics nor with 3D modelling, to design and create functional paper gliders. Their system fulfils all the requirements and shows how performative design applications can be used to improve any fabrication process. The idea of this project was to apply the same methodology for the creation of table top water fountains.

One classical way of performing the optimisation is to approach it as an "inverse problem": The problem is modelled mathematically as an equation that can be solved (for example by taking the inverse of a matrix). This means that if we can come up with an exact mathematical representation of the problem, it will be possible to find an exact solution, or at least to find an optimal solution (think of pseudo-inverses in Linear Algebra).

However, with the water simulation, things get slightly more complicated: The function to optimise, which would be the distance between the simulated water flow and the user-specified water flow, cannot be modelled mathe-

matically. It is a black box, since the distance is only calculated after the simulation has run a certain number of iterations, which is completely unpredictable. It is impossible to find a mathematical equation that would model the entire problem. Furthermore, we can make no assumptions about the continuity of the function, as a very slight change in ground shape can result in a very big and sudden change in water flow. We thus had to resort to a very specific sort of optimisation strategies: derivative free optimisation.

2.4 Optimisation Algorithms

Several algorithms of that sort exist, and they all have one aspect in common: The algorithm takes as many different points as it can in the parameter space, varying each parameter widely, because we do not know at all how the parameter space is constructed. We do not know whether it is continuous, how many local or global minima it has...

2.4.1 The Simplex Downhill Algorithm

Amongst all the derivative free optimisation algorithm, the most notorious for its simplicity but also for its fairly certain convergence to a global minimum (less likelihood of being stuck in a local minimum), would be the Simplex Downhill Algorithm. This optimisation method (also called Nelder-Mead algorithm) was developed in 1965 by Nelder and Mead ([NM65]) and only uses function evaluations (no derivatives) to iterate and converge. It is based on the concept of the simplex, which is the smallest geometric construct for the dimension of the parameter space: If the parameter space is one-dimensional, the simplex is a line, two-dimensional, the simplex is a triangle, three-dimensional, a quadrilateral ... This geometric construct is used to move through the parameter space, moving one of its points at each iteration according to a certain set of rules (see figure 2.5).

Concretely, the simplex method starts off by initialising the simplex: If the dimension of the problem is N, the simplex will consist of N+1 points scattered through the merit/parameter space. The cost function (to opti-

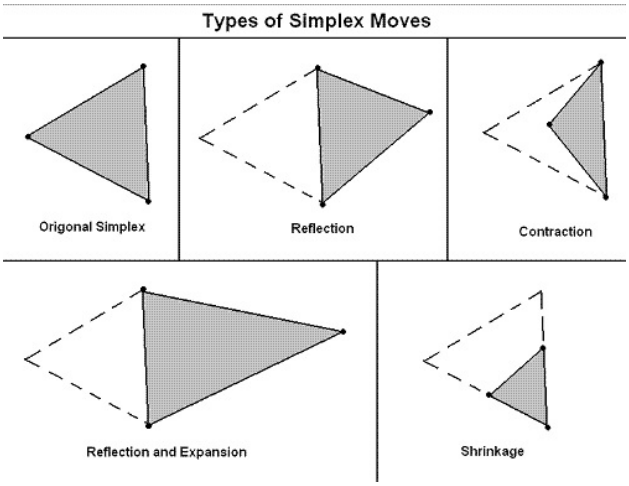

Figure 2.5: Diagram of the Simplex rules in two dimensions. Taken from `https://wiki.ece.cmu.edu/ddl/images/005.gif`

mise) is then evaluated at each of these points and they are ordered, from the "best" to the "worst". The centroid (the average/barycenter of all the simplex points) is also computed. The algorithm then takes the point with the worst distance and applies transformations to its values (either a reflection in relation to the centroid, or an expansion etc.) until it is not the worst point anymore (see flowchart 2.6).

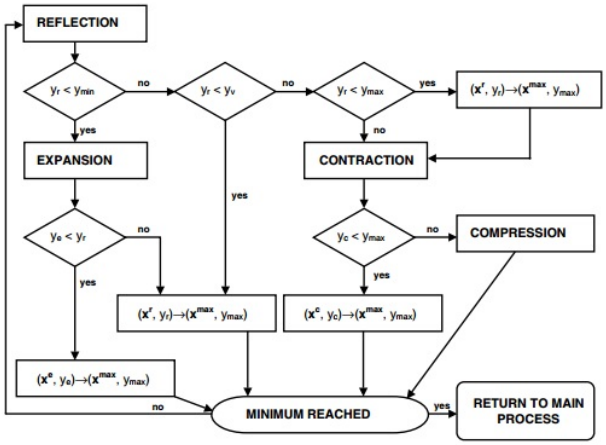

Figure 2.6: Flowchart of the Simplex Downhill Algorithm ([Qui12])

The great advantage of the simplex method is that it is easy to under-

stand and to implement and it requires no information about the cost function (other than just the evaluation of the function at the simplex points). Furthermore, if the algorithm parameters are chosen well (see [Kos02]), it is able to crawl out of local minima due to the scattering of the simplex's points across the parameter space (though of course, it can happen to get stuck). For these reasons, the simplex method was chosen for this project, as I could easily implement is myself and thus have control and overview of the optimisation. However, we thought just settling on the Simplex might be a bold choice, so we decided to also try out another strategy to get an idea how the Simplex performed and if it was really the best choice.

While the simplex uses a very logical and grounded approach, some of the most widely used optimisation algorithms rely on more complicated rules for their iterations, which are based on statistics. The obvious counterpart to the Simplex Downhill Algorithm hence seemed to be the Genetic Algorithms.

2.4.2 Genetic Algorithms

Genetic Algorithms were formally introduced in the United States by John Holland in the late 1970s ([Hol75]). They are inspired by the processes of natural selection and biological evolution. The method uses a population of individual solutions, represented as genomes. At each iteration, the algorithm selects individuals at random that will be parents and produce children for the next "generation" (iteration). Mimicking nature, the algorithm creates generation after generation, using selection, mutation and genetic crossover rules, until the genomes evolve toward an optimal solution.

The main difference between genetic algorithms and classical optimisation algorithms is that the rules these genetic processes use to modify their selection of solutions at each iteration are based on random probabilities (individuals are selected at random, and probabilities are used to determine how parents generate new children genomes). This makes genetic algorithms much more suited for complex optimisation problems where the objective function can be discontinuous, non-differentiable or nonlinear. They are also less susceptible to getting stuck in local minima than for example gradient

descent algorithms. However, these genetic methods are highly computationally expensive and can take much longer to converge.

2.5 Summary

This project touches many different areas of Computer Graphics and Mathematics, where significant advances have been made and research has been rewarding, which is why the background literature behind this project is very extensive. But the challenge, rather than to find or design a new method or approach, is to pick only simple algorithms and methods out of all these parts and combine them to produce a fully working system. In this sense, the project relates mostly to specific performative design projects, which also build on a lot of background research from different domains and try to unite specific solutions in a single coherent system.

Chapter 3

Analysis and Design

All the information gathered in the background research evidently conditioned the design of the final system we developed. The first choice that was made concerned the water simulation, as all of the rest would obviously build on it. Researching the Shallow Water Equations, it seemed that a classical way of implementing them was through the MATLAB language. Indeed, the terrain is split into square cells and the water over it is represented as a height field. Through the Lax-Wendroff scheme, every cell's new height can be calculated in parallel, independently from the others, as it only depends on the heights from the previous time step. This is why the matrix representation common to MATLAB comes in very handy to solve the Shallow Water Equations.

Additionally, the MATLAB representation allowed me to easily create animations for the water simulation: Indeed, it is possible to simultaneously display several height fields (the ground and the water) as surfaces of different colours and transparence. The display can then be updated at every iteration, hence creating an animation of the water simulation. This allowed us to easily review the realism of our simulation.

However, once the decision of using MATLAB for the water simulation was taken, the most reasonable option was to code everything in MATLAB in order to stay coherent. This meant that we would not be focusing on creating a fancy user interface or a complex modelling interface. The input

fields for the fountain topologies would be simple MATLAB matrices, which pointed us towards an easy way of generating various user inputs: Indeed, I could use any sketching program that allows pen transparency to draw a height map of the input water fountains (the luminance of a pixel is the height of the point in 3D) and directly import them into MATLAB.

The water simulation part hence didn't present any problems in MATLAB. For the optimisation, in terms of coding, MATLAB didn't create any obstacles either: I implemented the Simplex Downhill algorithm myself to have more control and be able to follow the process at every iteration, and as for Genetic Algorithms, there is already a built-in version available in the global optimisation toolbox. However, the choice of programming language did become an issue at the end, since, once the system was nearly complete, it turned out that the computations were quite expensive and the optimisation could sometimes take up to one or two days to run for large fountains. If the project had been longer, it may have been wise to transfer the computations for the optimisation to the GPU to speed up the process. Nevertheless, MATLAB had the advantage of the ease of access and coding, and in terms of data structures, was perfectly suited for this project.

All these factors had an obvious influence on the final design of the system, as well as the choice to concentrate on the performance and achievements of the system rather than a fancy high-level user interface. The final system consists of a single MATLAB function which contains all the necessary functions for the water simulation and the optimisation, and a script that loads the user input and executes that function. The input consists of the position of the source in the fountain, and of two images: one represents the geometry (as a height map) of the initial water fountain, the other one represents the desired flow as a binary map (ones where water should be, zeros where it should be dry). The size of the images (which has to be the same) represents the spatial extent of the fountain, divided into square cells (pixels of the images).

During the execution, MATLAB performs the Downhill Simplex algorithm: At each iteration, the Shallow Water simulation is run and the topology of the water fountain is altered in order to minimise the objective func-

tion, defined as the difference between the resulting flow (dry and wet cells mask) and the input binary flow map. The system returns the optimal fountain design (also as a height map) once the algorithm has converged or exceeded the maximum number of iterations. This result can then be exported to a PLY format mesh using a MATLAB script I wrote, which in turn can be converted to a suitable format for 3D printing. Even though the final system does not have any particular user interface, it still constitutes a fully functioning entity, from the conception and the optimisation all the way to the physical output.

The system is hence fairly coherent and solid as a whole, each part of it is based on concrete theories, methods and background literature. Nevertheless, an important aspect of the project still consisted in using tricks as well in the methods as in the implementation, to make everything work together as a whole.

Chapter 4

Implementation

4.1 Implementation of the Water Simulation

The shallow water equations are perfectly suited to model certain natural phenomena like tsunamis for instance, where the water depth is orders of magnitude smaller than the horizontal scale of the problem, the height variation of the fluid due to the waves is very small compared to the average water depth, and the height variation of the sea bed is also tiny compared to the water depth and thus only has a small influence on the water body's behaviour. In this project however, none of these cases are respected: the height variation of the underlying surface (the water fountain) is very important compared to the water depth, the water flow is quite violent and irregular (nothing like small waves in the ocean) and most importantly, there are immerged as well as dry areas on the fountain. We hence needed to adapt the Shallow Water model to the problem.

I started by implementing a regular, simple Shallow Water simulation, inspired by the code in [Rob11], making good use of the MATLAB's matrix structures to speed up the calculations.

The main variables used in the simulation script are the height map of the ground (B), the height map of the water (H), and the momentum matrices along the x and y-axis, U and V. The other matrices (Hx, Uy ...) are used to store the interpolated values during the half-steps of the Lax-Wendroff

scheme. The main parameters of the simulation are:

- The size of the matrices: The input topologies are always square images (100x100 pixels for instance). I set the size parameter n to the size of the input along one axis, minus two. This means that n would be 98 for a 100×100 input image for example. This is because there are no calculations on the border cells, which miss a neighbour cell in one direction, so the actual computations only happen on the $n \times n$ inner grid. Depending on the boundary conditions we want to simulate, the values at the border cells are then set manually at every iteration.

- The time step: It controls how much time passes between two iterations of the simulation. I determined its value empirically by finding the lowest time step at which the simulation diverges, and choosing a number slightly below that one, in order to have a fast, but stable simulation.

- The physical size of the grid cells: Virtually any number can be chosen for those, but in order to make the computations easier, I set them to 1 millimetre (a 100×100 grid would then make a 10×10 cm water fountain). Since I used millimetres as units for the grid cells, I needed to set the gravitational constant to 9800 $mm \times s^{-2}$.

Out of these parameters, the time step is the most important and unpredictable. There is no easy way of finding a good value for it other than trial and error. Once these parameters are set however, the rest of the implementation of a simple Shallow Water Equation based simulation is fairly straight-forward: The code consists of a while-loop that updates the depth map of the water using the Lax-Wendroff scheme. To simuate reflective boundaries (acting as walls), the height of the water is set manually, at every iteration, to the same as the inner water columns just before the boundary, and the momentum orthogonal to the boundary set to zero.

Fortunately, it is very easy to plot height maps in MATLAB. This made it possible to create dynamic animations of our water simulations by plotting the ground topology in 3D and updating the surface of the water at every

Lax-Wendroff iteration, which was very useful to test the realism of the simulation and keep track of artefacts or divergent behaviours.

Finally, I added a damping constant (effects illustrated in figure 4.1): If left unsupervised for a long time, the Lax-Wendroff integration tends to create additional waves from computational noise, but most importantly, ignores dissipation of energy. If a droplet hits a water surface modelled with the Lax-Wendroff method, the surface will keep oscillating indefinitely. To simulate the loss of momentum, I added a damping constant (inspired by the method in [WJT11]): At every iteration, the resulting momentum along the X and Y axes would be multiplied by this damping factor (inferior, but very close to 1). This way, the simulation would behave realistically, as the surface would come back to a relaxed state after a while.

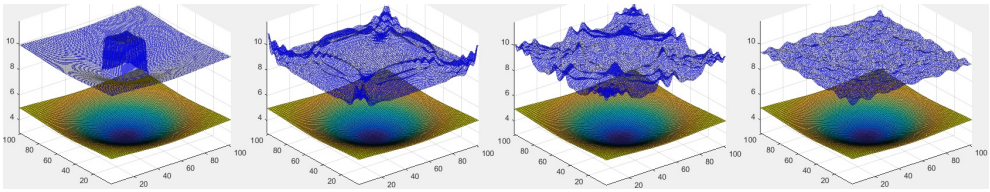

Figure 4.1: Effects of damping on the propagation and reflection of the waves.

This simulation worked perfectly fine for entirely immersed structures, however the goal of the project was to obtain a water simulation for a flowing fluid, hence a space with wet as well as dry areas. Multiple papers have been published on keeping track of the wet bed and the position of the water front using shallow waves, but the methods described were too complex to be used in this project. Most practices are based on the HLLC (Harten-Lax-van Leer-Contact) solver, which was introduced by E.F. Toro in [TSS94]. The method calculates the wave speeds and differentiates different types of scenarios in order to keep track, at every iteration, of the contour of the fluid. However, this method is very complex, and even if extensions to two dimensions have been developed (see [GWS13]), it would have cost me too much time to try to implement it in this project. Additionally, this would have resulted in heavy computations, which would have made the water simulation much slower. For this project however, we needed a quick and efficient fluid simulation so

that it could be used for the optimisation.

I decided, instead of trying to achieve the utmost physical accuracy, to use a computational trick, inspired by the method in [Rob11]: The water depth should never be null, since the Lax-Wendroff scheme involves a division by the water depth, which would result in a divison-by-0 error. So I simply set a constant epsilon to an arbitrarily small value and said that water depths below epsilon would be set to epsilon, but considered null, and water depths above epsilon would be considered part of an immersed area. Since epsilon is small and the momentum values in the "dry" cells are set to zero, these areas behave as if there really was no water.

In theory, this method sounds solid enough, but the implementation revealed otherwise: Unless the value chosen for epsilon was substantially high (about a tenth of the average water depth), the simulation would diverge. Additionally, in some dry areas, water would start flowing even though the concerned cells were never reached by the main water body.

I decided to apply another trick: At every iteration, a wetness map would be created, registering all the cells with a water depth superior to epsilon. The wetness map from the previous iteration would be dilated by a small structural element (one cell radius) and then applied as a mask to the new wetness map. If cells far from the main water body misbehave and are suddenly classified as "wet" in the new map, their water depth will be set back to epsilon, and their momentum values set to zero, because we apply the previous wetness mask. This way, only a one cell expansion of the water body per iteration is allowed, and artefacts such as water suddenly flowing in dry areas will be prevented. Surprisingly, this also resolved some of the divergence behaviours and made it possible to assign a fairly small value to epsilon (it is now set to the same value as the cell size).

Finally, the default boundary conditions for Shallow Water Simulations are usually reflective boundaries, acting as walls. But for this project, we wanted boundaries that absorb the water, as if the water just fell off at the boundaries of the fountain. To model this, I set the water depth at the boundaries to epsilon, the momentum parallel to the boundary to the same value as in the inner row, and the speed of the water orthogonal to the

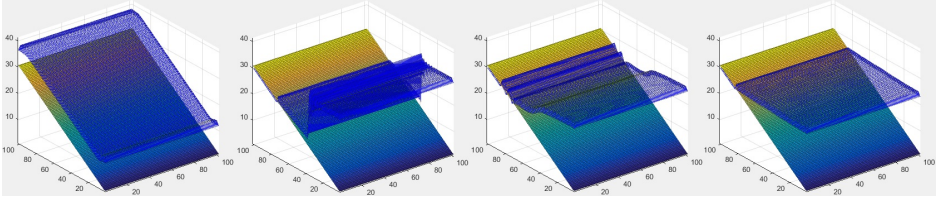

Figure 4.2: Shallow water simulation involving wet and dry areas.

boundary to twice the speed in the inner row (the momentum is then simply the water mass, equal to the height of the column in this case, multiplied by the water speed). This means that the water at the boundary would "fall off" twice as fast as just before the boundary, therefore dragging the water off the sides, simulating, in some way, the viscosity present in real fluids.

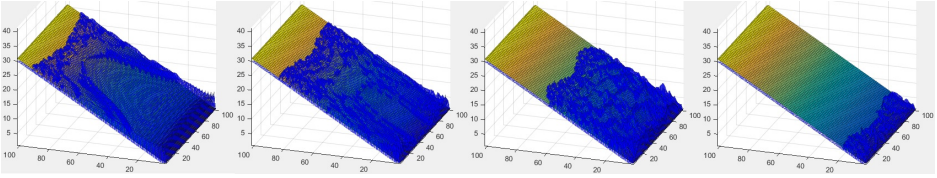

Figure 4.3: Shallow water simulation with dry areas and water "drains" on the sides.

As for the sources that will also need to be modelled in the fountain simulations, they were simulated by simply adding a certain quantity of water, at every iteration, to all the cells in a small array (5 by 5 cells) around the source position.

As the figures (4.1, 4.2 and 4.3) show, at this point, the water simulation fulfilled all the requirements and worked for all the scenarios needed for the system. However, we wanted to obtain the flow of the water or, to be precise, a binary map indicating where water will flow and which areas of the topology will be dry. In order to obtain this, we needed the water simulation to reach a stable state, where the wetness map undergoes very little or no changes at all from iteration to iteration. This means we needed to determine a criterion which, when satisfied, would signal that the water simulation is stable and won't change substantially no matter how much longer it runs.

The final stopping condition I chose works fairly well, but depends a lot

on the value chosen for the time constant and the size of the maps. Indeed, if the time constant is very small, the changes in immersed areas from iteration to iteration will also be very small even though the water body might still be expanding in reality, and if the map is very big, there might be a temporary stable state (while a hole is being filled for instance) that does not coincide with the final stable state the simulation would reach a little bit later.

The final criterion is based on two parameters: a threshold and a number of iterations. At every iteration, the percentage of difference between the current wetness map and the one from the previous iteration is calculated. If that percentage happens to be lower than the said threshold (set at 2% in my simulation), the simulation waits for a certain number of iterations (set at 300 for a 50x50 fountain grid). If during those iterations, the percentage of difference never gets higher than twice the threshold (which means the wetness map stays stable), the simulation stops and we consider having obtained a stable outcome. However, if the difference exceeds twice the threshold, the counter of iterations is reset to zero and restarts counting. Basically, this means that the simulation will run until the wetness difference gets lower than the threshold percentage, and will then run at least a certain number of iterations where the difference is not allowed to exceed twice the threshold, otherwise the stopping criterion is reset. Additionally, I set a maximum number of iterations for the simulation, to make sure it does not run indefinitely if the criteria never were to be satisfied.

Here again, the values for these parameters were set empirically by testing them out, watching the simulation and seeing if the automatic stop coincided with something that looked like a stable state in the dynamic simulation. For both the minimum number of iterations and the threshold, they were set so that we kept a safety margin, running the simulation longer than probably needed, but to make sure it reaches a stable state. The values chosen were tested on several examples, and depend mostly on the size of the grid (the bigger the grid, the longer the simulation has to run, and the smaller the percentage of change at every iteration is). In the end, we settled on 50 by 50 cell topologies, because they allowed to get fast simulation results for the optimisation algorithm (1 or 2 seconds for 500 water simulation iterations)

while still permitting a certain amount of detail in the fountain designs.

Through adding all these additional methods and schemes, I managed to adapt the Shallow Water Simulation to our problem, when it was, in fact, not conceived for situations of this type (big slopes, wet and dry areas ...). But the biggest part of the project remained: using the results from the water simulation to set up an optimisation system.

4.2 Implementation of the Simplex Downhill Optimisation

The optimisation of the fountain topology to match a required flowing pattern is not an easy problem, as we do not know anything about the function we want to optimise. We defined this function as the sum (element by element) of the absolute difference between the binary user input and the binary wetness map obtained by the simulation. This basically means that the distance for a topology is the number of cells that are in a different state (wet/dry) from the user input flow map. Furthermore, we wanted the optimal topology returned by the optimisation to stay somewhat close to the initial fountain designed by the user (in order not to completely ignore his concept and artistic intent). To achieve this, a second term was added to the objective function: the sum (element-wise) of the absolute differences between the height map of the initial fountain design and the new topology. For the optimisation to stay focused on matching the flowing patterns however, we multiplied this second term by a small constant (2%) to decrease its impact.

Let S be the simulated wetness mask returned by the water simulation, I_U the user defined wetness mask, G_S the variable geometry of the new fountain and G_U the user defined starting geometry. We define the distance d of our objective function as:

$$d = \sum_{i,j} |S(i,j) - I_U(i,j)| + 0,02 * \sum_{i,j} |G_S(i,j) - G_U(i,j)| \tag{4.1}$$

Defining the objective function was not very complicated. Nevertheless,

the problem is that we do not have any mathematical information about it: We do not have access to any mathematical expression for the result of the simulation, which means we do not have a mathematical expression for our objective function. We do not know whether it can be differentiated, if it is even continuous (it probably isn't), and how many local or global minima it possesses.

I starting tackling the optimisation issue by testing out some built-in MATLAB optimisation functions (for constrained derivative-free optimisation) on very small grids (10x10 cells) using the height map of the fountain directly as the vector to optimise (100 parameters). I tried out the built-in pattern-search algorithm for instance, because it did not require any mathematical information on the objective function. It seemed promising, but ended up failing completely. In addition to taking almost two hours to compute, only two of the cell heights in the fountain geometry were altered and the modifications did not make any sense at all. Since the algorithm was already implemented, I had no access or control over the iterations and could not get any further information on the problem. This is why we decided it would be best to implement an algorithm myself in order to be able to monitor the optimisation more directly.

The implementation of the Simplex Downhill algorithm was very straightforward. The choice of the parameter vector that the method should optimise, however, was not that obvious. Indeed, there were multiple possibilities: The most obvious choice for the parameter vector is simply to use the heights of every point in the grid. However, this meant that the parameter vector would be very large (100 parameters for a 10 by 10 grid) and there was no easy way to guarantee the smoothness of the result. One could have added a smoothness term to the objective function, but the optimisation would still have been very gruesome due to the large number of parameters. Indeed, every optimisation algorithm would have treated the height at every point as independent parameters, when in fact, the height at a single point is very dependent on the height of its neighbours in a smooth design. This option did not capitalise on this smoothness assumption, which is why it was quickly discarded.

Instead of taking the height of each point into account, we decided to use a lower resolution approach: Indeed, rather than working on single heights, it seems more sensible to work with larger bumps and holes to adapt the water flow. From this concept of bumps came the idea to use a lower resolution grid of Gaussian peaks. To optimise a 50 by 50 fountain for instance, rather than to use a 2500 parameter vector, it makes much more sense to use a vector containing for example 25 parameters that represent the heights of the Gaussian peaks positioned on a 5 by 5 grid over the fountain (one Gaussian peak positioned every 10 fountain cells). The standard deviation of the Gaussians is fixed and defined so that they slightly overlap with one another. In addition to considerably simplifying the optimisation (100 times fewer parameters), this approach will also conserve the overall smoothness of the fountain topology: it is a both low resolution and low frequency method.

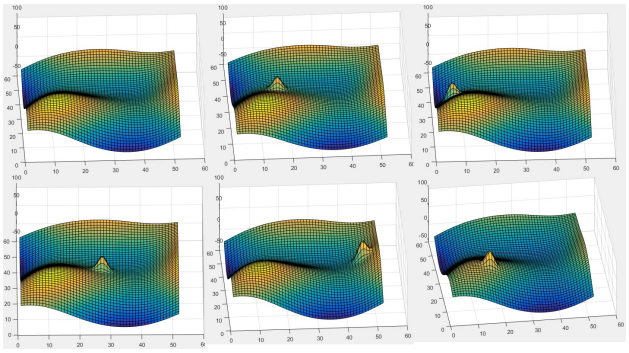

Figure 4.4: Example of a fountain topology (top left) with added Gaussian peaks.

The Simplex Downhill algorithm will then start by initialising all the points of the geometric simplex. If the parameter vector contains n elements, the size of the simplex will be (n+1). The first point of the simplex will be the starting point of the optimisation (in our case the user's input design), and the other points will be created by taking this starting point and translating it along one of the dimensions of the parameter space, i.e. adding a certain value to one of the parameters. For our problem, this would mean that

the starting point would be $\begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$ since no Gaussian peaks are added to the original fountain design. The second point would then for instance be $\begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$, the third $\begin{pmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix}$ etc., until the last one $\begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix}$, spanning the entire parameter space.

Alternatively, there is a second way of initialising the simplex that seemed more promising for this case: Instead of instantiating the simplex points in a deterministic way, it is also possible to take a new random vector for every simplex point. The probability of obtaining two linearly dependent vectors is virtually null so it is a perfectly acceptable initialisation. This method is more hazardous, but can, with a little luck, accelerate the optimisation if one of the starting vectors is already fairly close to the optimal solution.

The value of each element of these parameter vectors will represent the height of the Gaussian peak added to the water fountain at the associated position. This means that to convert a parameter vector to a height map, I needed to take the original water fountain topology and for all the elements in the vector, add a Gaussian peak of the specified height at the corresponding position.

Repeating this process at every simplex iteration, every time a new simplex point is created, though, seemed quite inefficient. To resolve this, I instantiated the simplex points not as parameter vectors containing the heights of the Gaussian peaks, but directly as water fountain height maps. This substitution was possible because the Simplex Algorithm's operations are linear operations on the vectors containing the heights of the Gaussians. These are equivalent to the same linear operations on the entire height maps corresponding to the parameter vectors.

So in the end, the simplex is constituted of height maps that were derived from the original design by adding Gaussian peaks of random heights on a lower resolution grid. If this is a 5 by 5 grid for the Gaussians, on a 100 by 100 fountain height map, for instance, the simplex will contain 26 height maps of size 100 by 100. The algorithm then performs the simplex

iterations following the simplex rules as explained in [Qui12], in order to find the optimal topology that matches the water flow requirements specified by the user. But how can we assess when a solution is "good enough", when a fountain is "close enough" to the optimal design?

The usual termination criteria (see [PTVF92, p. 410]) for the Simplex Downhill algorithm are controlled by a tolerance value on either the distance function or on the parameter vector that is being optimised. For example, one could set a threshold so that at every iteration, the algorithm calculates how much the distance value for the centroid (barycenter) of the simplex points was decreased and compares that value to the threshold. However, choosing a good termination criterion for the Simplex Downhill is known to be delicate and the criteria we just described can easily be fooled. This is why, in general, the algorithm gets re-initialised once a minimum is found, and the process is reiterated several times to be sure the optimisation does not get stuck in a local minimum.

For this project, it was very difficult to find a good termination criterion that made sense. Indeed, all the quantities related to the distance between the simulated wetness map and the user input depend on many parameters such as the size of the grid, the form of the user input and its difference from the flow on the initial fountain design etc. There is no accurate way of predicting the values these quantities will take and deciding just with the values, if the result of the optimisation is good enough. We could have, for instance, set that the algorithm would stop iterating when the change to the centroid, from iteration to iteration, was lower than a certain percentage of the change during the first iterations. However, this would not have been optimal either, because the centroid does not evolve linearly: It could get stuck near a local minimum until a shrinkage of the entire simplex is performed and the algorithm starts converging faster again. Such a termination criterion could stop the simplex before this shrinkage step, thus preventing it from reaching a better solution.

In the end, we could not figure out any perfect termination criterion. This is why the breaking condition of our simplex algorithm was eventually just set to a maximum number of iterations. I used two threshold: one

for the total simplex and one for the number of minimum changes. Indeed, when the reflection or contraction returns a new point that is more optimal than the current best point of the simplex, a new minimum is found. The values of these thresholds were set empirically as well. There was no way of predicting how much the Simplex Downhill algorithm could improve the distance between simulated and input flow on a particular geometry, so we just decided on a certain amount of iterations that seemed reasonable and produced correct results. We will get back to the number of iterations in more detail in Chapter 5.

4.3 Built-in Genetic Algorithms

In order to put the results of the optimisation with the Simplex Downhill algorithm into perspective, we decided to test out another classical optimisation scheme: the Genetic Algorithm. In MATLAB's derivative-free optimisation toolbox[1], an implementation of the Genetic Algorithm is already available. Since the purpose of this was simply to have an element of comparison to assess the Simplex Downhill performance, we deemed it sufficient to directly use the built-in MATLAB genetic optimiser instead of implementing something from scratch.

Multiple tuning options for the genetic algorithm are offered: It is possible to change the mutation and crossover rules that determine how the next generation is created, the scale and shrink rates that control which proportion of the population is altered etc... For this project however, since we only desired a reference to compare the performance of the Simplex with, we left all these parameters at their default value. The only specifications were the maximum number of generations and the number of individuals per generation. Furthermore, since the algorithm was already implemented in MATLAB, we would not have had any close control over the variables from iteration to iteration, so there was no point in taking this approach much

[1]Set of built-in functions that perform derivative-free optimisation algorithms. MATLAB R2014b 32-bit version was used. Documentation at: `http://uk.mathworks.com/help/gads/index.html`

further. Nevertheless, it offered a good element of comparison to assess the performance of the Simplex Downhill algorithm on this problem.

Chapter 5

Testing

The testing of the system was done in two phases: First we had to test the water simulation, because it would be the basis of the optimisation. Without good water simulation results, there would be no way of testing the optimisation, since we would not know wether the bad results would be caused by a fault in the optimisation or by a failure in the fluid simulation. Once we were sure that the water simulation met our criterai, we moved on to testing the optimisation.

5.1 Testing the Water Simulation

There was no absolute way to objectively assess the quality and validity of the water simulation. A big part of the testing was done intuitively, by simply looking at the test results and gauging whether they seemed realistic enough or not. For that purpose, I modelled different input geometries, ran the simulation on those test cases and looked at the resulting animation. An sequence of still frames of a simulation over a large height field is shown in figure 5.1.

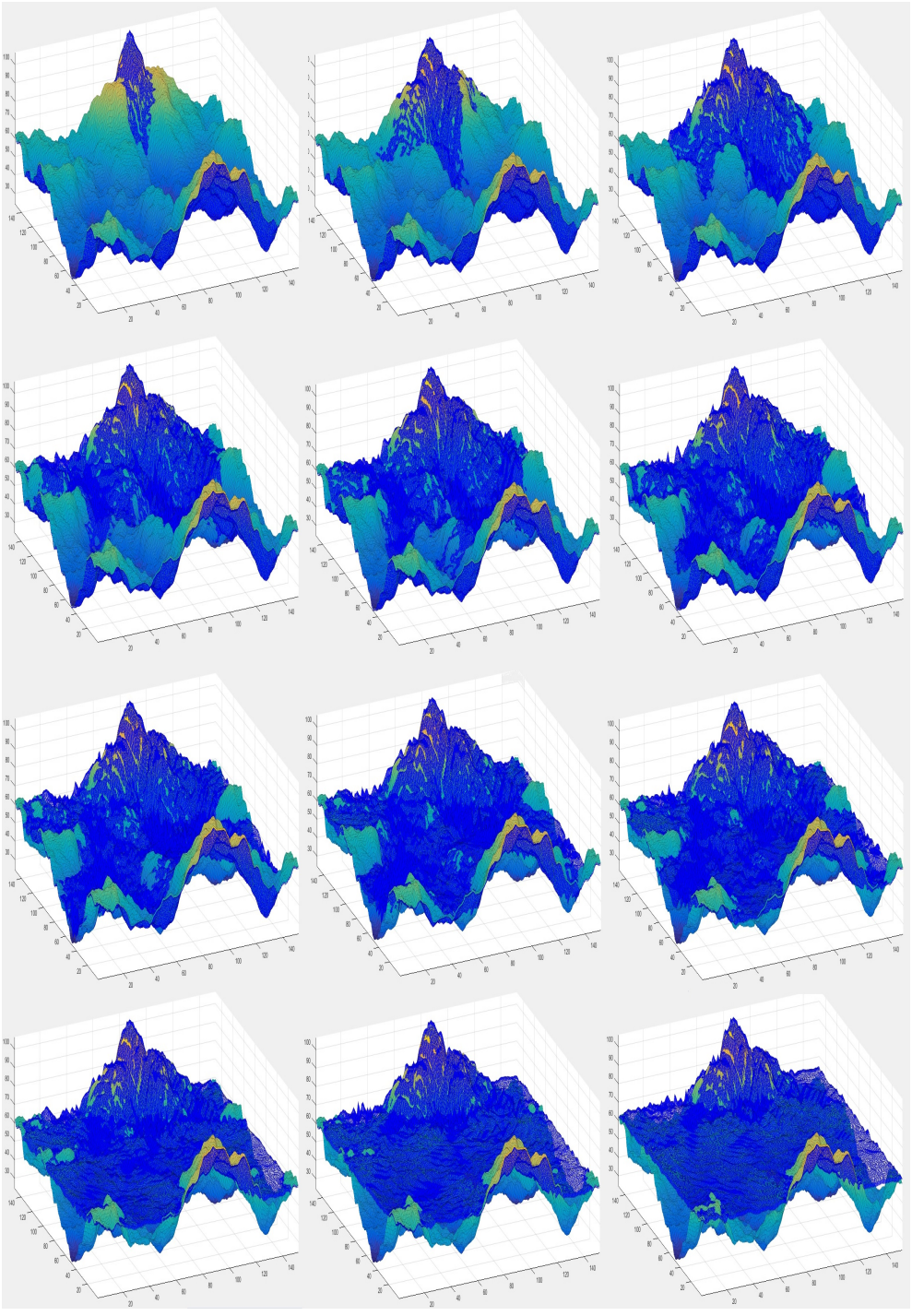

Figure 5.1: Sequence of screenshots of water filling a randomly generated height field. Boundaries are modelled as walls.

Even though the Shallow Water Equations are not always perfectly suited to model the cases we use them for in this project, the water simulation looks good enough and seems to behave realistically. However, if the numerical values for the parameters are not in the perfect range, the simulation can easily contain slight artefacts (too high and abrupt waves, water flowing out of a dry area...) or even diverge and crash (this mostly happens if the time step is too big, or if the epsilon-depth value is too small).

5.2 Two different Optimisation Strategies

In order to test whether the optimisation worked well, I designed three small input height maps (see figure 5.2). The idea was to design iconic topologies with very particular geometries for the flow, to see if the optimisation would return valid results in all cases. We would then run MATLAB's genetic algorithm on the same test cases and thus be able to assess the performance of the simplex (in terms of both speed and accuracy of the results).

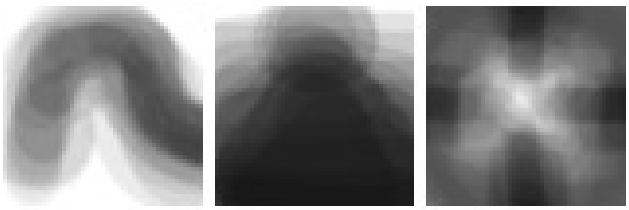

Figure 5.2: The black and white images of the input height fields.

The three 50 by 50 cell (so that the simulation and optimisation would run quickly) height maps were created by simply drawing black and white images (of 50 by 50 pixels) in Microsoft Paint.NET [1]. In those greyscale images, the brightness of a pixel corresponds to its height in 3D. The user required flow patterns were created by drawing a binary image (as in figure 5.3). The wet areas are drawn in black and the dry areas in white.

The first test case is a simple meander with a very close flow requirement to test the accuracy of the algorithm, the second case is a diverging stream,

[1]Paint.NET is a free image and photo editing software for PCs that run Windows. It is an extension of the original Microsoft Paint available on Windows systems by default.

Figure 5.3: The binary flow masks specified by the user.

which the user requires to be split in the middle, and the final one consists of a cross created by four streams, which the user wants slightly rotated. The initial user design was purposely initialised fairly close to the solution, to make sure the algorithm would not get stuck in local minima. Nevertheless, there is enough initial mismatch for the algorithm to be able to achieve significant progress (see figure 5.4).

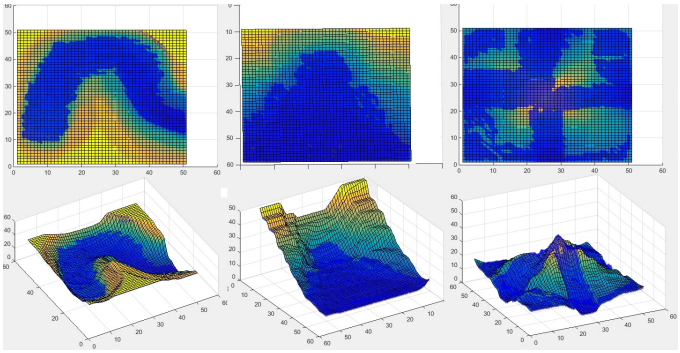

Figure 5.4: Initial flow simulations on the input geometries from two view angles.

Rather than using only one execution of the Downhill Simplex algorithm, I used three successive optimisations. For all three of them, the maximum number of simplex iterations was set to 1000 and the maximum updates of the simplex minimum to 80. In order to apply some sort of a multi-resolution optimisation, I set the variable controlling the spacing between the Gaussian peaks to 10 grid cells for the first optimisation, and to 5 for the two others. This means the first simplex contains 26 topologies and will perform many iterations compared to its size, while the two other simplexes contain 101 points, but will proportionally undergo fewer iterations or modifications.

The last variable I experimented with in this multi-optimisation scheme was the maximum height of the Gaussian peaks. Indeed, the random vectors used to initialise the simplex points will contain values between a quantity –M and +M, where M corresponds to this maximum height. This value was set to a tenth of the maximum height of the fountain for the first optimisation, a sixth for the second, and a tenth again for the last. These exact numbers probably don't have a great influence, since the algorithm can create bigger peaks (by expansion) or smaller peaks (by contraction). Nevertheless, the idea behind it was to have a rough optimisation at first with larger bumps, a finer optimisation afterwards with narrower and higher peaks, and finally a very fine optimisation with small, narrow peaks.

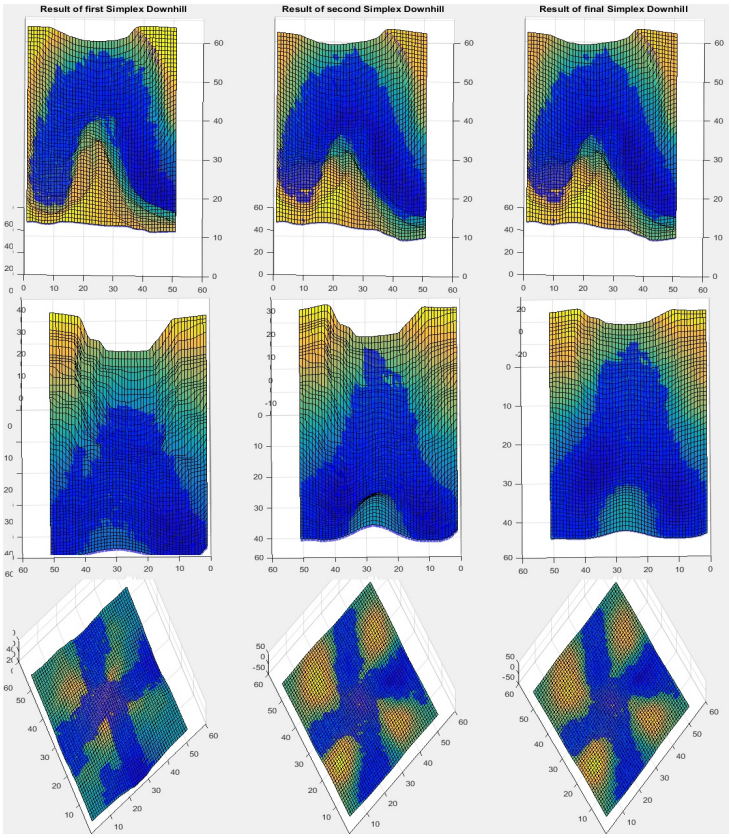

Figure 5.5: Results of the three Simplex Runs for the three input cases.

After running the optimisation algorithm, the results we obtain are very satisfying: As we can see in 5.5, the final flows almost match the user re-

quirements perfectly. To have an element of comparison, I ran MATLAB's Genetic Algorithm on the same input water fountains with the same flow requirements. In order to make the comparison as valid as possible, I set the population size to 30 individuals and the number of generations to a hundred. This means the algorithm will run the water simulation about 3000 times, which is approximately the same amount as for the three successive Simplex Downhill Optimisations.

The parameter vector optimised by the genetic algorithm, however, is the same as for the two last steps of our Simplex optimiser: It is a vector of length 100, containing the heights of all the Gaussian peaks added on the 10 by 10 grid over the fountain. This means the genetic optimisation has access to the same order of precision on the modifications of the fountain topology as the Simplex version.

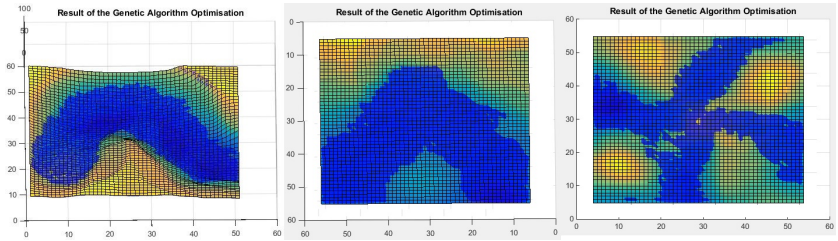

Figure 5.6: Results of the built-in Genetic Algorithm for the three input cases.

The results generated using the genetic algorithm (see figure 5.6), however, seem slightly less optimal than those obtained through the Simplex Optimisation. The first two examples, which are a bit simpler, work fine with the genetic optimisation as well, but on the third we see that the results did not turn out entirely as good as with the Simplex. Nevertheless, the genetic algorithm has a drawback: its optimisation strategy is not as structured and logical as the Simplex, it is more based on probabilities and random mutations and crossovers between the best individuals. Furthermore, it does not use an initial guess: this means the initial fountain design that the Simplex works on is not even used by the genetic algorithm. Still, it does have an influence on the distance function, which will eventually guide every

optimisation algorithm to stay close to that initial design. But ultimately, the genetic algorithm does not exploit that initial starting design.

As expected of a built-in algorithm, the genetic algorithm does perform faster (three times on average) than our Simplex Downhill implementation. This comparison was solely based on the number of times the two methods call the water simulation method (about 3000 times for both). The Simplex Downhill algorithm would take around six hours to finish, while the genetic optimisation would only take two. However, the Simplex results are obviously much more accurate, so all this comparison tells us is that the Simplex Downhill makes better use of the information of the simulation than genetic algorithms, which means it is better suited for problems with heavy computation times for the objective function. This shows that our custom optimisation based on the Simplex Downhill algorithm was indeed the right choice for this project. It was easy to implement, it does not run extermely quickly, but the results are good and the algorithm does not seem to get stuck in local minima (especially thanks to the re-initialisations).

Finally, an objective way of testing the results of our system is to 3D print the output, let actual water flow on the fountain and compare the results to the simulation. However, as the 3D printing process can be quite long and expensive, we decided to wait until the end of the project, when we had tested the system digitally on concrete examples and were confident enough in its functionality, to move on to physical fabrication and testing.

Chapter 6

Results

The tests performed confirmed that the optimisation was working and that we had obtained some sort of whole, functioning system. However, we wanted to really push it and find its limits, so we decided to test both the Simplex and the genetic algorithm on a the most complex concept of water fountain design that came to mind: a spiral.

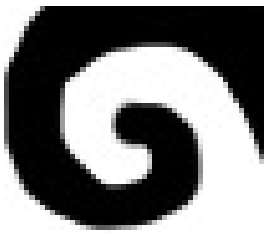

Figure 6.1: Input flow mask.

The idea here was to have both optimisers start out from a flat topology, having only the user flow input as a spiral, hence adding additional difficulties to the optimisation because the initial design is very far from the goal.

The Simplex Downhill method performed very well on this example (figure 6.2). The genetic algorithm however, seems to have had more difficulties. We can clearly notice that the more complex the user input gets, the less precise the genetic algorithm's results are, compared to those of the Simplex Downhill algorithm.

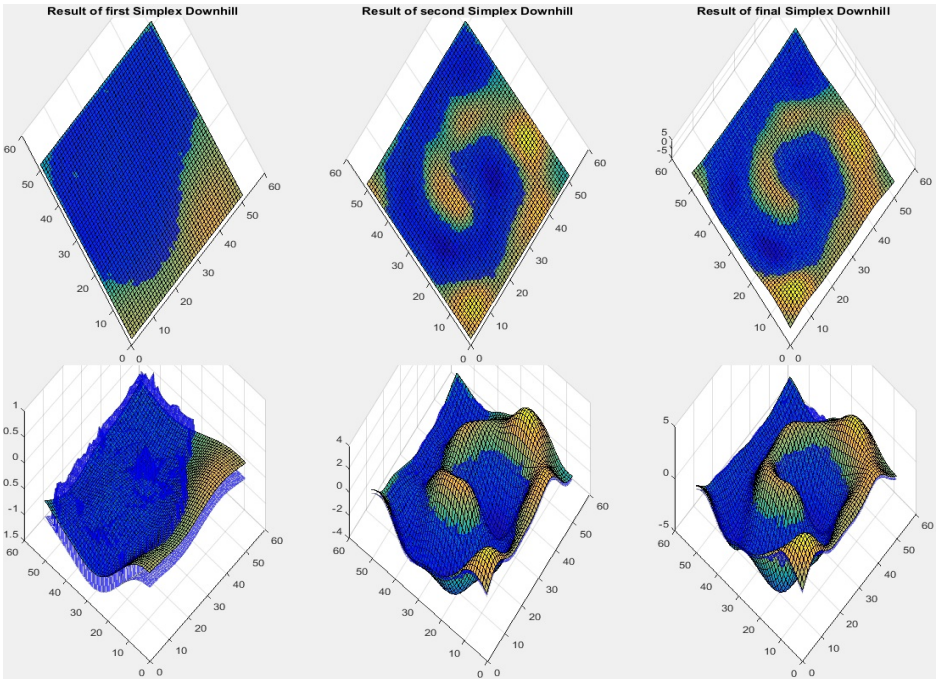

Figure 6.2: Results of the three Simplex Downhill optimisations for the spiral case.

As in the previous section, it was adjusted so that both methods would call the water simulation the same number of times. The Simplex Downhill optimisation took almost four times as long to finish (7 hours versus 2 for the genetic algorithm), but the results were much closer to the optimal solution. This does not mean that any of the two methods is better or preferable to the other, however it does give crucial information about the efficiency of the use of the water simulation: The Simplex method may waste more time on other computations, but it makes a more efficient use of the information gathered at every call of the water simulation function.

This means that if we wanted to run the system on a much larger height field (say 500 by 500 cells), while all the other parameters (number of simplex points, fluid simulation parameters ...) remain the same, the Simplex method would be preferable: Since most of the computation time would be caused by the water simulation, the two methods would take approximately as long to finish, but the Simplex's results would be much more accurate.

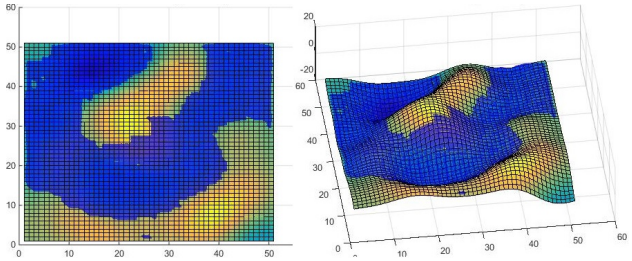

Figure 6.3: Results of the genetic optimisation for the spiral case.

Finally, since the original purpose of the project was to create a system to design real-world water fountains, we 3D printed one of the models. I chose the third design (on the right in figure 5.3) from the testing section, which contained four streams spreading from the center.

Real-world Fabrication of a Water Fountain

I wrote a MATLAB function to convert the height map output into a PLY format mesh. The 3D object is simply the surface resulting from the height field, placed on a cubic base, so that it would stand on its own. This PLY file can then easily be converted into an STL file, a format that is used for 3D printing.

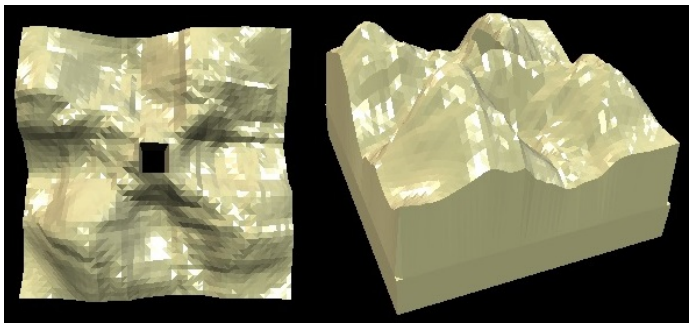

Figure 6.4: The STL mesh for the 3D printing.

In the simulation and the optimisation, the grid cell dimensions are set in millimeters. Since the model for this fountain was a 50 by 50 height field, this means it would be equivalent to a 5 by 5 centimeter object. However,

since the Shallow Water simulation does not involve any viscosity or surface tension, or any quantity whose proportional influence changes with the size of the model, these absolute dimensions did not have any importance. Indeed, whether we model the fountain as a 5 by 5 kilometer or a 2 by 2 millimeter object does not change the behaviour of the water simulation (as long as we adapt the other quantities such as the gravitational constant to the units). In order to have more flexibility with the aquarium pump and tube later, it was hence possible to simply double the size of the mesh of the fountain to make it a 10 by 10 centimeter object.

Using mesh processing software such as BLENDER, it was then possible to virtually drill a hole into the mesh, where the source should be, via CSG operations. This hole would then be needed to put a PVC tube through it, which, plugged on to a small aquarium pump, would make water flow down the 3D printed fountain from the position of the source in the same way as in the computer simulation. For the final setup, we placed the fountain on a structure of LEGOs to hold the PVC tube in place. This construction was in turn put inside a plastic basin with the aquarium pump, which was submerged in water.

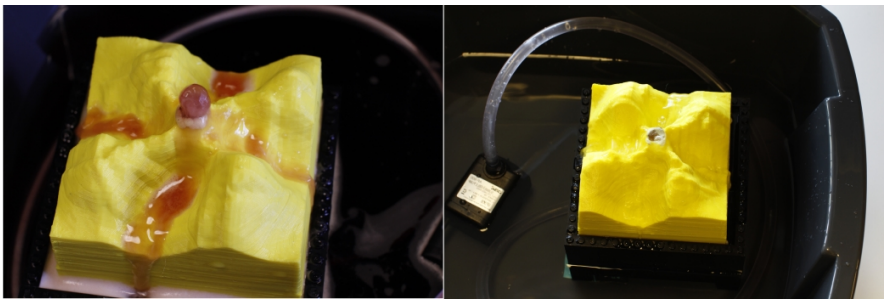

Figure 6.5: The finished water fountain setup.

The water was coloured in red using food colouring, so that the streams would be more visible on the yellow object, and the output rate of the pump was adjusted so that the water would not shoot into the air, but just slowly flow down the fountain. We set up a camera on a tripod and took pictures from above: This allowed us to concretely compare the physical flow on the 3D printed object with the original user input mask. Figure 6.6 shows the

three stages of the process: the user input, the optimised topology with the water simulation, and finally the 3D printed object in the water pump setup.

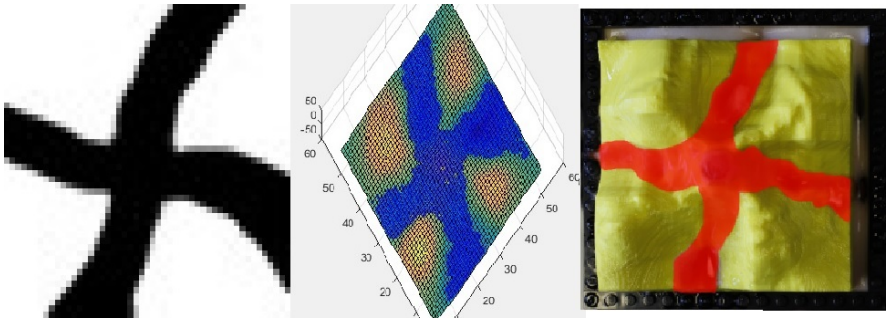

Figure 6.6: The three stages of the system.

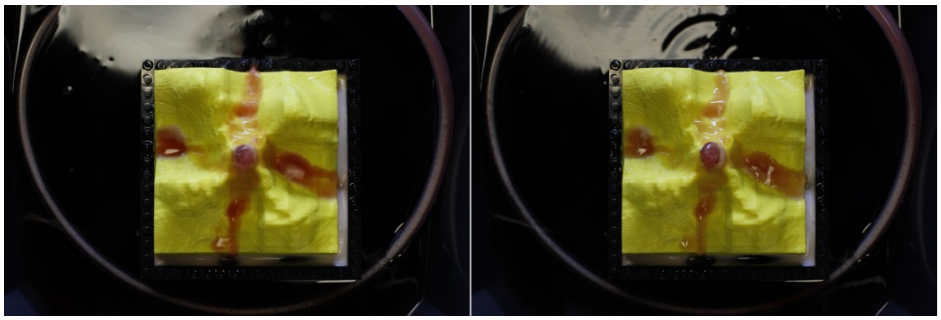

Figure 6.7: Two still frames of the water flow.

On the last picture (in figure 6.6), I highlighted the water flow in red to make the flow of the water more visible. Indeed, the appearance of the water on the photograph is influenced by its depth: In deep areas, the dye will make the water much darker, whereas in shallow areas it will be almost transparent. We can see this very clearly in figure 6.7, where even between two successive frames, the flowing pattern and water depths can change. The photographs do not convey the actual behaviour of the fluid very well: Some areas might seem almost dry, when in fact there was water flowing there. This is why I highlighted the flowing pattern based on the sequence of images we captured (see figure 6.8).

The results obtained with the real fountain are close to the required flow,

even if the final design is not perfect. A few discrepancies between the simulation and the real world setup prevail. Nevertheless, the end result behaves, for the most part, as expected, and there are no crucial mismatches.

Finally, we should also take into account that the setup itself could have influenced the outcome. Indeed, the technique of fitting of an aquarium pump tube through the object could certainly be improved: The water, at the output of the tube, tended to go in circles, running down each one of the four streams alternatively. This behaviour at the exit of the tube can distinctively be recognised on the sequence of pictures in figure 6.8. This might have been caused by the fluid coming out of the tube with too much speed. But although the aquarium pump had an adjustable output flow rate, this behaviour could not be prevented. Indeed, when the output was too weak, not enough water would run down the sides for one to clearly distinguish the flowing pattern.

As we just noticed with the still frames, it is hard to assess to what extent the requirements were satisfied. Nevertheless, the setup does work very well with the 3D printed fountain and the small aquarium pump, and the behaviour of the water looks satisfying. In the end, we hence managed to obtain a fully functioning system that offers a solution to every part of the problem: from the conception and the digital optimisation to the actual physical fabrication.

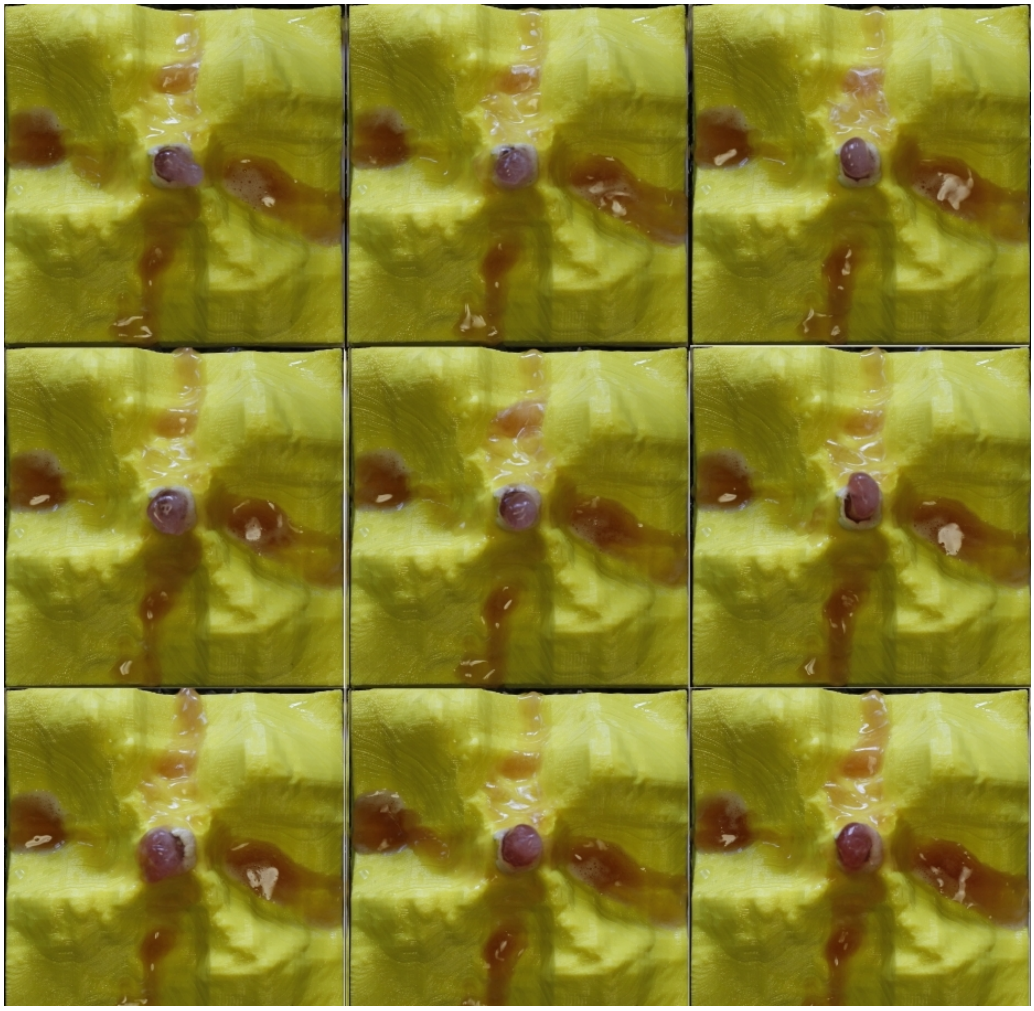

Figure 6.8: Nine successive shots of water flowing on the 3D printed fountain, taken within less than a second. Reads from left to right, and top to bottom.

Chapter 7

Conclusions, Evaluation and Further Work

This project combines very different areas of Computer Graphics into a single problematic. The key was to find a balanced system between the physical accuracy of the water simulation part, the features of a possible sculpting interface, and the optimisation. All of these topics could probably make for research projects of their own as there is a lot to explore, so the challenge was to find an equilibrium of simple yet efficient implementations for each of these problems. It could have been easy to get stuck in one of these areas, which would have made the project fail, or over-achieve on parts of the problem and hence end up with lack of performance on other parts.

One of the biggest challenges was the wide scope of the topic, which created many different possibilities and paths to go down, both in terms of theoretical approach and implementation. But this variety is also what made the project very interesting, as it allowed me to gain insight into several key areas of Computer Graphics, and to gain experience on handling such open research topics, where nothing is really specified or fixed at the beginning. The starting point only specified the aim of the project, which was an optimisation system for water fountains, but the way to approach the problem was completely open to decision.

Therefore, the initial strategy for the project was to attempt very rapid

prototyping, in order to quickly have a functioning system, and then refine each part step by step. However, in the end, the project was developed very linearly: I started with the water simulation, worked on it until it was evolved enough for the problem and satisfied all our requirements, and only then switched to the optimisation part. Nevertheless, it all worked out well and the project was finished in time.

The water simulation was based on the Shallow Wave Equations, which, using a few modifications and extensions, achieved enough realism and flexibility of use to meet the requirements of the system. The optimisation, developed in a second stage of the project, uses the Simplex Downhill algorithm to adapt the water fountain topology to the flow pattern specified by the user. Even though the solutions take a lot of time to compute, especially if the input height maps are large, the algorithm manages to find topologies that satisfy the requirements of the user. Overall, the system is consistent and performs reasonably well.

Finally, the fabrication part through 3D printing extended the project in a very nice way, and allowed us to objectively test the performance of the developed system. Indeed, the foremost important goal of a performative design system is, after the conception, the real-world fabrication of the designs. Even though the pictures do not represent the real behaviour of the fluid on the setup objectively (it is easier to see in the video attached on the DVD), the results turned out very well, and are quite close to the digital simulation.

If the project was to be continued or re-done however, with the theoretical knowledge gained through this work, it would be much wiser to implement the optimisation and the simulation on the GPU, in order to be able to run many more calculations much faster. MATLAB, even if it is not the fastest one available for the processing of huge computations, is a very good programing language for prototyping because of its ease of use and clarity of syntax. This is why it was the perfect choice for this project, where we did not know since the beginning where we were going and how we would approach the problem. But now that an efficient approach has been found, it would be very beneficial to re-implement the system in a faster running language to fully exploit its potential.

This would make it possible to have a truly "interactive" system. For this project, we were aware that achieving an efficient optimisation would likely mean creating an off-line process. We decided to implement interaction as an iterative scheme of user inputs: Indeed, users can fetch the optimisation results, correct them and restart the optimisation. With a faster running system however, this could be done in real-time, instead of having to wait a few hours for the system to finish computing.

Besides, it might be a good addition to implement a modelling interface to complete the system. Indeed, the only thing missing is a way for the user to create a fountain design without having to switch to another piece of software. This modelling interface, though, does not need to be very complex or have an incredible number of features. It does not even need to be a 3D modelling interface: it could very well be a 2D sketching panel, that simply allows to add layers to the ground in order to create a height map (similarly to what I did to create the test examples) and perhaps contain an additional feature to specify the position of the source(s).

Implementing a user interface automatically also raises further questions about user control and constraints: For now, the optimisation works on matching two wetness masks. But what if we allowed the user to express a desired amplitude of the final height field of the water, or to specify certains parts of the fountain design that should not be altered? In the way the system has been implemented, the objective function for the optimisation can be modified very easily and it would not be a challenge to have to match height fields instead of binary masks, or to add a huge additional cost when "untouchable" areas of the topology are altered. However, the modelling interface would then need to be extended to contain more features for the user input.

Nevertheless, even without a proper user interface, the system we created is coherent and contains efficient solutions for both the water simulation and the optimisation. We were able to complete all the stages of a performative design system: from the conception to the optimisation and finally the physical fabrication.

Appendix A

System and User Manual

The MATLAB code attached on the DVD consists of six files. 'WaterAnimationScript' is a MATLAB script that contains several examples of height fields on which a dynamic water simulation can be run. The user simply needs to uncomment the desired case and comment the others. The script produces an animation of water flowing on the input height field and runs until it is stopped by the user. The function 'mutliscalenoise' is used to produce random topologies as input for this script.

The script 'runSystem.m' is the main script that calls the function 'completeSystem.m' (performs optimisation and simulation), which in turn calls 'height2PLY.m' to produce a PLY format mesh, and 'dynamicWaterSimulation.m' to create an animation on the result topology.

To use the entire system, the user simply needs to modify the beginning of the 'runSystem.m' script. In the first section, the simulation and optimisation parameters can be modified, and in the second section (where 4 test cases have already been written and commented), the user can create a new test case, where he will need to specify the path to the image of the input fountain height field (if there is one), the position of the source, and the path to the input desired flow mask.

The script 'runSystem.m', since it controls all the parameters of system and paths to the user input files, is hence the file to modify in order to use the system.

Finally, the DVD also contains all the input images that were used to generate results, and a video taken of the setup with the 3D printed foutain. It shows water flowing over it, while dye is added, to progressively make the water darker.

Appendix B

Code Listing

The file 'runSystem.m' is the main script that sets all the necessary parameters for the water simulation and the optimisation, also specifying the path to the user input and the type of optimisation. The middle section contains the three test cases and the spiral case (the case to run should be uncommented and the rest commented). After running the optimisation process (by calling 'completeSystem.m', [p. 72-79]), the result topology is exported to a PLY format file (by calling 'height2PLY.m', [p. 80-82]), and an animation is run to show water flowing on the optimised fountain.

```
%%================================================
% This is the main script. It sets all the necessary parameters and calls
% 'completeSystem.m'.
%================================================
close all;
tic

%================================================
% PARAMETERS FOR THE WATER SIMULATION
%================================================
sim = []; %structure containing all the parameters
sim.g = 9810; %gravitation in mm*s^-2
sim.dx = 1;  sim.dy = 1; %spatial steps in mm
sim.dt = 0.0003; % time step in s
sim.epsilon = sim.dx; % minimum water depth (H<epsilon means cell is dry)
sim.damping = 1-0.05*sim.dt; %constant used for water flux sim.damping
sim.sourceIntensity = sim.dx/30; % source height increment per iteration
sim.simulThresh = 0.02; % less than ..% change to stop simulation
sim.iterThresh = 250; % minimum number of stable iterations
```

```
20 sim.maxIter= 600; %maximum number of iterations in the water simulation
21 %================================================
22
23
24 %================================================
25 % DIFFERENT USER INPUT  TEST CASES (Uncomment the desired one)
26 %================================================
27 % im=imread('Meandre.jpg'); im=rgb2gray(im); im=double(im);
28 % surf(im); title('Initial fountain design'); figure;
29 % userSp = imread('MeandreInput.jpg'); userSp=rgb2gray(userSp);
30 % userSp = double(userSp); userSp = userSp./max(max(userSp));
31 % userSp= (userSp<0.9); imshow(userSp); title('Desired flow'); figure;
32 % sim.iS = 14; % coordinates of the source where the water comes from
33 % sim.jS  = 9;
34 %//////////////////////////////////////////////////////////////////////////
35 % im=imread('multiFlows.jpg'); im=rgb2gray(im); im=double(im);
36 % surf(im); title('Initial fountain design'); figure;
37 % userSp = imread('multiFlowsInput.jpg'); userSp=rgb2gray(userSp);
38 % userSp = double(userSp); userSp = userSp./max(max(userSp));
39 % userSp = (userSp<0.9); imshow(userSp); title('Desired flow'); figure;
40 % sim.iS  = 25; sim.jS  = 25;
41 %//////////////////////////////////////////////////////////////////////////
42 im=imread('split.jpg'); im=rgb2gray(im); im=double(im);
43 surf(im); title('Initial fountain design'); figure;
44 userSp = imread('splitInput.jpg'); userSp=rgb2gray(userSp);
45 userSp = double(userSp); userSp = userSp./max(max(userSp));
46 userSp = (userSp<0.9); imshow(userSp); title('Desired flow'); figure;
47 sim.iS  = 12; sim.jS = 25;
48 %//////////////////////////////////////////////////////////////////////////
49 % % OPTIMISATION FROM A FLAT INPUT _____________
50 % sim.iS  = 25; sim.jS = 25;
51 % userSp = imread('SPIInput.jpg');
52 % im = zeros(51,51); userSp=rgb2gray(userSp);
53 % userSp = double(userSp); userSp = userSp./max(max(userSp));
54 % userSp = (userSp<0.9); imshow(userSp); title('Desired flow'); figure;
55 % sim.dt = 0.001; % Optimisation from flat needs less precision
56 % sim.epsilon = 0.3; % minimum water depth
57 %================================================
58
59
60 %================================================
61 % RUNNING THE OPTIMISATION ALGORITHM
62 %================================================
63 sim.n = size(im,1)-2; % all height fields are (n+2)*(n+2) in size
64 sim.maxH = sim.n; %for optimisation from flat plane
65 sim.BStart = im;
66 if (max(max(im))~=0)  %for optimisation from user design
67     sim.BStart = sim.maxH * im/max(max(im)); % rescale the fountain height
68 end
69 sim.input = userSp; % binary image of desired flow
```

```
70 % simplexUse : true for simplex algorithm, false for genetic optimisation
71 simplexUse = true;
72 FinalFountain = completeSystem(sim, simplexUse);
73 %================================================
74 toc
75
76 %================================================
77 %% EXPORT THE RESULT HEIGHT FIELD AS A PLY MESH
78 height2PLY(FinalFountain);
79
80 %================================================
81 %% RUN A DYNAMIC WATER SIMULATION TO SHOW THE RESULT
82 figure;
83 dynamicWaterSimulation(sim, FinalFountain);
```

The file 'completeSystem.m' is a MATLAB function that performs the optimisation (simplex downhill or genetic algorithm). The file contains three more functions: *simplex(sim)* performs the Simplex Downhill algorithm, *objectiveFunc(sim, topo)* calculates the distance between from the current simulation to the user input, and *waterSimulation(sim, B)* performs the fluid simulation.

```
%% MAIN FUNCTION ===========================================================
function resultTopo = completeSystem(sim, simplexUse)
    %% PROBLEM INITIALISATION ================================================
    % Global Simulation Variables
    sim.H = sim.epsilon.*ones(sim.n+2,sim.n+2); % water Depth
    [sim.x,sim.y] = ndgrid(-(sim.n+1)/2 : (sim.n+1)/2);
    sim.U = zeros(sim.n+2,sim.n+2); % momentum along X (matrix columns)
    sim.V = zeros(sim.n+2,sim.n+2); % momentum along Y (matrix lines)
    sim.Hx = zeros(sim.n+2,sim.n+2);            sim.Hy = zeros(sim.n+2,sim.n+2);
    sim.Ux = zeros(sim.n+2,sim.n+2);            sim.Uy = zeros(sim.n+2,sim.n+2);
    sim.Vx = zeros(sim.n+2,sim.n+2);            sim.Vy = zeros(sim.n+2,sim.n+2);
    sim.wetPrevious = zeros(sim.n+2,sim.n+2);
    sim.wetXR = zeros(sim.n+2,sim.n+2);         sim.wetYD = zeros(sim.n+2,sim.n+2);
    sim.wetU = zeros(sim.n+2,sim.n+2);          sim.wetL = zeros(sim.n+2,sim.n+2);
    sim.Bx = zeros(sim.n+2,sim.n+2);            sim.By = zeros(sim.n+2,sim.n+2);
    disp('Problem Variables set up');
    surf(sim.BStart); hold on; W = waterSimulation(sim, sim.BStart); Ww = sum(W,3)./20;
  surf(sim.BStart + Ww - 2*sim.epsilon*(Ww<=1.0001*sim.epsilon), ...
        'FaceColor', 'b', 'EdgeAlpha', 0.2, 'FaceAlpha', 0.65);
    title('Original design flow'); figure;
    drawnow

    %% OPTIMISATION ==========================================================
    if (simplexUse)
        sim.maxSimplexIterations = 1000;
        sim.gridSampling = 10;
        sim.simplexSize = ((sim.n+2 -mod((sim.n+2),sim.gridSampling)) /sim.gridSampling) ...
            * ((sim.n+2 -mod((sim.n+2),sim.gridSampling)) /sim.gridSampling) +1;
        sim.peakHeight = sim.maxH/10;
        Res1 = simplex(sim);
        % Restart the optimisation problem with a different initialisation
        sim.BStart = Res1;
        sim.maxSimplexIterations = 1000;
        sim.gridSampling = 5;
        sim.simplexSize = ((sim.n+2 -mod((sim.n+2),sim.gridSampling)) /sim.gridSampling) ...
            * ((sim.n+2 -mod((sim.n+2),sim.gridSampling)) /sim.gridSampling) +1;
        sim.peakHeight = sim.maxH/6;
        Res2 = simplex(sim);
        % Restart the optimisation problem again with a different initialisation
```

```
40          sim.BStart = Res2;
41          sim.maxSimplexIterations = 1000;
42          sim.gridSampling = 5;
43          sim.simplexSize = ((sim.n+2 -mod((sim.n+2),sim.gridSampling)) /sim.gridSampling) ...
44              * ((sim.n+2 -mod((sim.n+2),sim.gridSampling)) /sim.gridSampling) +1;
45          sim.peakHeight = sim.maxH/10;
46          Res3 = simplex(sim);
47
48          % Display results
49          subplot(1,3,1); surf(Res1); hold on; W1 = waterSimulation(sim, Res1);
50          Ww1 = sum(W1,3)./20;
51          surf(Res1 + Ww1 - 2*sim.epsilon*(Ww1<=1.0001*sim.epsilon), ...
52              'FaceColor', 'b', 'EdgeAlpha', 0.2, 'FaceAlpha', 0.65);
53          title('Result of first Simplex Downhill');
54          subplot(1,3,2); surf(Res2); hold on; W2 = waterSimulation(sim, Res2);
55          Ww2 = sum(W2,3)./20;
56          surf(Res2 + Ww2 - 2*sim.epsilon*(Ww2<=1.0001*sim.epsilon), ...
57              'FaceColor', 'b', 'EdgeAlpha', 0.2, 'FaceAlpha', 0.65);
58          title('Result of second Simplex Downhill');
59          subplot(1,3,3); surf(Res3); hold on; W3 = waterSimulation(sim, Res3);
60          Ww3 = sum(W3,3)./20;
61          surf(Res3 + Ww3 - 2*sim.epsilon*(Ww3<=1.0001*sim.epsilon), ...
62              'FaceColor', 'b', 'EdgeAlpha', 0.2, 'FaceAlpha', 0.65);
63          title('Result of final Simplex Downhill');
64          resultTopo = Res3;
65      else
66          % Use the Genetic Algorithm from the Matlab Global Optimisation Toolbox
67          disp('Starting Genetic Optimisation');
68          options = gaoptimset('Generations', 100, 'PopulationSize', 30);
69          sim.gridSampling = 5;
70          sim.nVariables = ((sim.n+2 -mod((sim.n+2),sim.gridSampling)) /sim.gridSampling) ...
71              * ((sim.n+2 -mod((sim.n+2),sim.gridSampling)) /sim.gridSampling);
72          genIter =0; % genetic iteration counter
73          % run the constrained genetic optimisation
74          [geneticResult, fval] = ga(@objective, sim.nVariables, [],[],[],[], ...
75              -sim.n/5*ones(sim.nVariables,1), sim.n/5*ones(sim.nVariables,1), [], options);
76          disp('Genetic Optimisation finished');
77          resultTopo = sim.BStart;
78          % reconstruct a height field from the optimised parameters
79          for p = 1: sim.nVariables
80              trow = mod(p* sim.gridSampling, sim.n+2);
81              tcol = (p*sim.gridSampling - trow)/(sim.n+2) *sim.gridSampling +1;
82              tx = sim.x + ((sim.n+3)/2 - trow) * ones(sim.n+2,sim.n+2);
83              ty = sim.y + ((sim.n+3)/2 - tcol) * ones(sim.n+2,sim.n+2);
84              resultTopo = resultTopo + geneticResult(p) * ...
85                  exp(- (tx.^2+ty.^2) / (2*sim.gridSampling*sim.gridSampling));
86          end
87          disp(['The best distance is = ', num2str(objective(geneticResult))]);
88          % display result
89          surf(resultTopo); hold on; W = waterSimulation(sim, resultTopo);
```

```
90          Ww = sum(W,3)./20;
91          surf(resultTopo + Ww - 2*sim.epsilon*(Ww<=1.0001*sim.epsilon), ...
92              'FaceColor', 'b', 'EdgeAlpha', 0.2, 'FaceAlpha', 0.65);
93          title('Result of the Genetic Algorithm Optimisation');
94      end
95
96      %% OBJECTIVE FUNCTION FOR THE GENETIC ALGORITHM ========================
97      function Dist = objective(params)
98          %computes the objective function for a set of input parameters
99          tempGeo = sim.BStart;
100         for pi = 1: sim.nVariables
101             trow = mod(pi* sim.gridSampling, sim.n+2);
102             tcol = (pi*sim.gridSampling - trow)/(sim.n+2) *sim.gridSampling +1;
103             tx = sim.x + ((sim.n+3)/2 - trow) * ones(sim.n+2,sim.n+2);
104             ty = sim.y + ((sim.n+3)/2 - tcol) * ones(sim.n+2,sim.n+2);
105             tempGeo = tempGeo + ...
106                 params(pi)*exp(- (tx.^2+ty.^2) / (2*sim.gridSampling*sim.gridSampling));
107         end
108         tempSim = waterSimulation(sim, tempGeo);
109         tt1 = sum(tempSim,3)./20;
110         tw1 = (tt1(:,:) > sim.epsilon+0.00001);
111         Dist = sum(sum(abs(tw1-sim.input))) + 0.02*sum(sum(abs(sim.BStart - tempGeo)));
112         genIter = genIter+1;
113         disp(['Iteration = ', num2str(genIter),'  Distance : ', num2str(Dist)]);
114     end
115 end
116
117
118 %% SIMPLEX OPTIMISATION FUNCTION ========================================
119 function R = simplex(sim)
120
121     % Evaluation of the initial fountain topology
122     startDist = objectiveFunc(sim, sim.BStart);
123
124     % SIMPLEX DOWNHILL ALGORITHM ===========================================
125     % INITIALISATION
126     alpha = 1;  beta = 2;  beta2 = 0.5;
127     simplex = zeros(sim.n+2,sim.n+2, sim.simplexSize);
128     simulDistances = zeros(sim.simplexSize,1);
129     simplex(:,:,1) = sim.BStart;
130     simulDistances(1) = startDist;
131     disp('Initialising simplex');
132
133     % RANDOM INITIALISATION OF SIMPLEX POINTS
134     for ii=1: sim.simplexSize-1 % Gaussians to respect smoothness
135         randTemp = 2*sim.peakHeight*rand(sim.simplexSize-1,1) - sim.peakHeight;
136         simplex(:,:,ii+1) = sim.BStart;
137         for pi = 1: size(randTemp,1)
138             trow = mod(pi* sim.gridSampling, sim.n+2);
139             tcol = (pi*sim.gridSampling - trow)/(sim.n+2) *sim.gridSampling +1;
```

```
            tx = sim.x + ((sim.n+3)/2 - trow) * ones(sim.n+2,sim.n+2);
            ty = sim.y + ((sim.n+3)/2 - tcol) * ones(sim.n+2,sim.n+2);
            simplex(:,:,ii+1) = simplex(:,:,ii+1) + ...
                randTemp(pi)*exp(- (tx.^2+ty.^2) / (2*sim.gridSampling*sim.gridSampling));
        end
        simulDistances(ii+1) = objectiveFunc(sim, simplex(:,:,ii+1));  %Simulation runs
        disp(['Iteration : ', num2str(ii+1), ...
            ';  Distance : ', num2str(simulDistances(ii+1))]);
    end

    minDist = startDist;
    iterMin=0; iter =0;
    disp('Simplex set up');
    WC = waterSimulation(sim, sim.BStart); WwC = sum(WC,3)./20;
    grid = surf(sim.BStart); hold all;
    gridW = surf(sim.BStart + WwC - 2*sim.epsilon*(WwC<=1.0001*sim.epsilon), ...
        'FaceColor', 'b', 'EdgeAlpha', 0.2, 'FaceAlpha', 0.65);

    % SIMPLEX ITERATIONS
    while ((iterMin <80) && (iter< sim.maxSimplexIterations)) %breaking condition
        [~, Indices] = sort(simulDistances); %sort simplex points
        firstInd = Indices(1,1);
        lastInd = Indices(size(simulDistances,1),1);
        centroid = (sum(simplex,3) - simplex(:,:,lastInd)) / sim.simplexSize;
        if(simulDistances(firstInd) <minDist)
            minDist = simulDistances(firstInd);
            iterMin = iterMin+1;
            disp(['Min iteration = ', num2str(iterMin), ...
                ';  Total iteration : ', num2str(iter)]);
        end
        %display the centroid of the simplex points
        WC = waterSimulation(sim, centroid); WwC = sum(WC,3)./20;
        set(grid, 'zdata', centroid);
        set(gridW, 'zdata', centroid + WwC - 2*sim.epsilon*(WwC<=1.0001*sim.epsilon));
        drawnow

        trialP = centroid + alpha*(centroid - simplex(:,:,lastInd));  %compute trial point
        trialDist = objectiveFunc(sim, trialP);
        if (trialDist < simulDistances(firstInd)) %expansion
            disp(['Iteration = ', num2str(iterMin), ...
                '  Expansion : ', num2str(simulDistances(lastInd))]);
            expandP = centroid + beta*(trialP - centroid);
            expandDist = objectiveFunc(sim, expandP);
            if (expandDist < trialDist)
                simplex(:,:,lastInd) = expandP;
                simulDistances(lastInd) = expandDist;
            else
                simplex(:,:,lastInd) = trialP;
                simulDistances(lastInd) = trialDist;
            end
```

```
190         else
191             if (trialDist > simulDistances(Indices(sim.simplexSize-1,1))) %contraction
192                 disp(['Iteration = ', num2str(iterMin), ...
193                     '  Contraction : ', num2str(simulDistances(lastInd))]);
194                 gamma = 0.5;
195                 contractP = centroid + gamma*(simplex(:,:,lastInd) - centroid);
196                 contractDist = objectiveFunc(sim, contractP);
197                 while ((gamma>1/8) && ...
198                         (contractDist > simulDistances(Indices(size(simulDistances,1)-1,1))))
199                     gamma = gamma/2;
200                     contractP = centroid + gamma*(simplex(:,:,lastInd) - centroid);
201                     contractDist = objectiveFunc(sim, contractP);
202                 end
203                 if (contractDist > simulDistances(lastInd))
204                     disp('Contracting entire simplex'); % contract whole simplex
205                     for si = 1:sim.simplexSize
206                         if (si ~= firstInd)
207                             simplex(:,:,si) = (1 - beta2) * simplex(:,:,firstInd) ...
208                                 + beta2 * simplex(:,:,si);
209                             simulDistances(si,1) = objectiveFunc(sim, simplex(:,:,si));
210                         end
211                     end
212                     iter = iter + sim.simplexSize;
213                 end
214                 simplex(:,:,lastInd) = contractP;
215                 simulDistances(lastInd) = contractDist;
216             else
217                 disp(['Iteration = ', num2str(iterMin), ...
218                     '  Reflection : ', num2str(simulDistances(lastInd))]);
219                 simplex(:,:,lastInd) = trialP; %replace by reflection
220                 simulDistances(lastInd) = trialDist;
221             end
222         end
223         iter = iter +1;
224     end
225     [~, Indices] = sort(simulDistances);
226     R = simplex(:,:,Indices(1,1));
227     disp(['The best distance is = ', num2str(simulDistances(Indices(1,1)))]);
228     clear simplex; clear simulDistances; clear Indices;
229 end
230
231
232 %% OBJECTIVE FUNCTION FOR THE SIMPLEX ALGORITHM ==================
233 function distance = objectiveFunc(sim , topo)  % Distance to user input
234     topoSim = waterSimulation(sim, topo);
235     tt1 = (topoSim(:,:,:) > sim.epsilon+0.00001);
236     tw1 = sum(tt1,3)./20; %averaged over 20 water simulation iterations
237     distance = sum(sum(abs(tw1-sim.input))) + 0.02*sum(sum(abs(sim.BStart - topo)));
238 end
239
```

```
240
241 %% WATER SIMULATION ============================================
242 function simul = waterSimulation(sim, B)
243 tic
244     %Runs the water simulation for all the variables in the struct sim
245     %and an input height field for the ground topology (B)
246
247     % reset all simulation variables
248     sim.H = sim.epsilon.*ones(sim.n+2,sim.n+2);
249     sim.U = 0.*sim.U; sim.V = 0.*sim.V; sim.Hx = 0.*sim.Hx; sim.Hy = 0.*sim.Hy;
250     sim.Ux = 0.*sim.Ux; sim.Vy = 0.*sim.Vy; sim.Uy = 0.*sim.Uy; sim.Vx = 0.*sim.Vx;
251     sim.wetPrevious = 0.*sim.wetPrevious;
252     % ground slope
253     sim.Bx(2:sim.n+1,2:sim.n+1) = B(3:sim.n+2,2:sim.n+1)-B(1:sim.n,2:sim.n+1);
254     sim.By(2:sim.n+1,2:sim.n+1) = B(2:sim.n+1,3:sim.n+2)-B(2:sim.n+1,1:sim.n);
255     sim.H(max(sim.iS-2,1):min(sim.iS+2,sim.n+2),max(sim.jS-2,1):min(sim.jS+2,sim.n+2)) =...
256         sim.H(max(sim.iS-2,1):min(sim.iS+2,sim.n+2),max(sim.jS-2,1):min(sim.jS+2,sim.n+2))...
257         + 20*sim.sourceIntensity;
258     wet = (sim.H > sim.epsilon);        HPrev = sim.H;
259     wetChange = 1;   it = 0;  % loop breaking variables
260     totalIter = 0;  % count the total number of iterations
261     simul = zeros(sim.n+2,sim.n+2,20); % output (20 iterations of wet map)
262
263     while (((it < sim.iterThresh) || (wetChange > sim.simulThresh)) ...
264             && (totalIter < sim.maxIter))
265
266         % SOURCE TERMS
267         sim.H(max(sim.iS-2,1):min(sim.iS+2,sim.n+2),max(sim.jS-2,1):min(sim.jS+2,sim.n+2)) =...
268             sim.H(max(sim.iS-2,1):min(sim.iS+2,sim.n+2),max(sim.jS-2,1):min(sim.jS+2,sim.n+2))...
269             + sim.sourceIntensity;
270
271         % FIRST HALF STEP ==================================================
272         % water height
273         sim.Hx(1:end-1,1:end) = (sim.H(2:end,1:end)+sim.H(1:end-1,1:end))/2 ...
274             - sim.dt/(2*sim.dx)*(sim.U(2:end,1:end)-sim.U(1:end-1,1:end));
275         sim.Hy(1:end,1:end-1) = (sim.H(1:end,2:end)+sim.H(1:end,1:end-1))/2 ...
276             - sim.dt/(2*sim.dy)*(sim.V(1:end,2:end)-sim.V(1:end,1:end-1));
277
278         wetXIdx = (sim.Hx > sim.epsilon);       wetYIdx = (sim.Hy > sim.epsilon);
279         wetXIdx(sim.n+2,:) = 0;             wetYIdx(:,sim.n+2) = 0;
280         sim.Hx = max(sim.Hx,sim.epsilon);           sim.Hy = max(sim.Hy,sim.epsilon);
281         wetXIdx = logical(wetXIdx);     wetYIdx = logical(wetYIdx);
282         sim.Ux = 0.*sim.Ux;    sim.Vx = 0.*sim.Vx;
283         sim.Uy = 0.*sim.Uy;     sim.Vy = 0.*sim.Vy;
284         sim.wetXR(2:end, :) = wetXIdx(1:end-1, :);
285         sim.wetYD(:, 2:end) = wetYIdx(:, 1:end-1);
286         sim.wetXR = logical(sim.wetXR);         sim.wetYD = logical(sim.wetYD);
287
288         % x momentum
289         sim.Ux(wetXIdx) = (sim.U(sim.wetXR)+sim.U(wetXIdx))/2 ...
```

```
290             - sim.dt/(2*sim.dx)*((sim.U(sim.wetXR).^2./sim.H(sim.wetXR) ...
291             + sim.g/2*sim.H(sim.wetXR).^2) ...
292             -(sim.U(wetXIdx).^2./sim.H(wetXIdx) + sim.g/2*sim.H(wetXIdx).^2));
293
294         sim.Uy(wetYIdx) = (sim.U(sim.wetYD)+sim.U(wetYIdx))/2 ...
295             - sim.dt/(2*sim.dy)* ((sim.V(sim.wetYD).*sim.U(sim.wetYD)./sim.H(sim.wetYD)) ...
296             - (sim.V(wetYIdx).*sim.U(wetYIdx)./sim.H(wetYIdx)));
297
298         % y momentum
299         sim.Vx(wetXIdx) = (sim.V(sim.wetXR)+sim.V(wetXIdx))/2 ...
300             - sim.dt/(2*sim.dx)* ((sim.U(sim.wetXR).*sim.V(sim.wetXR)./sim.H(sim.wetXR)) ...
301             - (sim.U(wetXIdx).*sim.V(wetXIdx)./sim.H(wetXIdx)));
302
303         sim.Vy(wetYIdx) = (sim.V(sim.wetYD)+sim.V(wetYIdx))/2 ...
304             - sim.dt/(2*sim.dy)*((sim.V(sim.wetYD).^2./sim.H(sim.wetYD) ...
305             + sim.g/2*sim.H(sim.wetYD).^2) ...
306             -(sim.V(wetYIdx).^2./sim.H(wetYIdx) + sim.g/2*sim.H(wetYIdx).^2));
307
308         % SECOND HALF STEP ================================================
309         % water height
310         sim.H(2:end-1,2:end-1) = sim.H(2:end-1,2:end-1) ...
311             - (sim.dt/sim.dx)*(sim.Ux(2:end-1,2:end-1)-sim.Ux(1:end-2,2:end-1))...
312             - (sim.dt/sim.dy)*(sim.Vy(2:end-1,2:end-1)-sim.Vy(2:end-1,1:end-2));
313
314         % Create a wet zone mask
315         wetP = imdilate(wet, strel('disk',1));
316         wet = (sim.H > sim.epsilon);
317         wet = wet.*wetP;%only allow 1 cell expansion of wet zone per dt
318         sim.H = wet.*sim.H;
319         sim.H = max(sim.H, sim.epsilon);
320         wet(1,:) = 0;               wet(sim.n+2,:) = 0;
321         wet(:,1) = 0;               wet(:,sim.n+2) = 0;
322         sim.wetL(1:end-1, :) = wet(2:end, :); sim.wetU(:, 1:end-1) = wet(:, 2:end);
323         wet = logical(wet);
324         sim.wetU = logical(sim.wetU);
325         sim.wetL = logical(sim.wetL);
326
327         % X MOMENTUM
328         sim.U(wet) = sim.U(wet) ...
329             - (sim.dt/sim.dx)*((sim.Ux(wet).^2./sim.Hx(wet) + sim.g/2*sim.Hx(wet).^2) ...
330             - (sim.Ux(sim.wetL).^2./sim.Hx(sim.wetL) + sim.g/2*sim.Hx(sim.wetL).^2)) ...
331             - (sim.dt/sim.dy)*((sim.Vy(wet).*sim.Uy(wet)./sim.Hy(wet)) ...
332             - (sim.Vy(sim.wetU).*sim.Uy(sim.wetU)./sim.Hy(sim.wetU))) ...
333             - (sim.dt*sim.g)/(4*sim.dx).* sim.Bx(wet).* (sim.H(wet) + HPrev(wet));
334         %Y MOMENTUM
335         sim.V(wet) = sim.V(wet) ...
336             - (sim.dt/sim.dx)*((sim.Ux(wet).*sim.Vx(wet)./sim.Hx(wet)) ...
337             - (sim.Ux(sim.wetL).*sim.Vx(sim.wetL)./sim.Hx(sim.wetL))) ...
338             - (sim.dt/sim.dy)*((sim.Vy(wet).^2./sim.Hy(wet) + sim.g/2*sim.Hy(wet).^2) ...
339             - (sim.Vy(sim.wetU).^2./sim.Hy(sim.wetU) + sim.g/2*sim.Hy(sim.wetU).^2)) ...
```

```
340             - (sim.dt*sim.g)/(4*sim.dy).* sim.By(wet).* (sim.H(wet) + HPrev(wet));
341
342         % Nullify momentum at dry cells and damp in wet zones
343         sim.U = sim.U.*wet;             sim.V = sim.V.*wet;
344         sim.U = sim.damping.*sim.U;     sim.V = sim.damping.*sim.V;
345
346         % ABSORBING/FREE BOUNDARIES
347         sim.H(:,1) = sim.epsilon;     sim.H(:,sim.n+2) = sim.epsilon;
348         sim.H(1,:) = sim.epsilon;     sim.H(sim.n+2,:) = sim.epsilon;
349         sim.U(1,:) = 2*sim.epsilon*(sim.U(2,:)<0).*sim.U(2,:)./sim.H(2,:);
350         sim.U(sim.n+2,:) = 2*sim.epsilon*(sim.U(sim.n+1,:)>0).* ...
351             sim.U(sim.n+1,:)./sim.H(sim.n+1,:);
352         sim.V(:,1) = 2*sim.epsilon*(sim.V(:,2)<0).*sim.V(:,2)./sim.H(:,2);
353         sim.V(:,sim.n+2) = 2*sim.epsilon*(sim.V(:,sim.n+1)>0).* ...
354             sim.V(:,sim.n+1)./sim.H(:,sim.n+1);
355
356         HPrev = sim.H;  %store previous water map
357
358         simul(:,:,1:19) = simul(:,:,2:20); %store the 20 last iterations
359         simul(:,:,20) = sim.H;
360
361         % STOPPING CONDITION=====================================================
362         %test if less than ..% change in wet map
363         temp = sum(sum(abs(wet-sim.wetPrevious)))/sum(sum(wet));
364         if (temp < wetChange)
365             wetChange = temp;
366             it = 0;
367         else
368             it = it+1;
369         end
370         %test if for a certain amount of iterations, the % of change does not
371         %rise again (simulation stays stable)
372         if (temp > 2*wetChange)
373             it = 0;
374         end
375         sim.wetPrevious = wet;  %store previous wetness map
376         totalIter = totalIter +1;
377     end
378     toc
379 end
```

The file 'height2PLY.m' is a MATLAB function that creates a PLY format mesh out of an input height field (i.e. a MATLAB matrix). This code was written to produce the mesh models for the 3D printing from the optimisation results.

```
function height2PLY (B)
% transforms an input height field into a mesh surface on a cube

    n = size(B,1)+2;
    H = zeros(n,n);
    H(2:end-1, 2:end-1) = B;
    H = H - min(min(H));
    H = H+8; %add a minimum height for a solid base

    vertex = zeros(n*n+4,3);
    faces = zeros((n*n+4)*8,3); % enough allocation to contain all faces

    %add vertices
    for j=1:n
        for i=1:n
            vertex(i+n*(j-1),1) = i;
            vertex(i+n*(j-1),2) = j;
            vertex(i+n*(j-1),3) = H(i,j);
        end
    end
    vertex(n*n+1,1) = 1; %upper left
    vertex(n*n+1,2) = 1; %upper left
    vertex(n*n+1,3) = 0; %upper left
    vertex(n*n+2,1) = n; %lower left
    vertex(n*n+2,2) = 1; %lower left
    vertex(n*n+2,3) = 0; %lower left
    vertex(n*n+3,1) = n; %lower right
    vertex(n*n+3,2) = n; %lower right
    vertex(n*n+3,3) = 0; %lower right
    vertex(n*n+4,1) = 1; %upper right
    vertex(n*n+4,2) = n; %upper right
    vertex(n*n+4,3) = 0; %upper right

    %add faces
    faceIdx = 1;
    for i=1:n-1
        for j=1:n-1
            currentV = i+n*(j-1);
            rightV = i+n*j;
            downV = i+n*(j-1)+1;
            diagV = i+n*j+1;
             % / distance:
```

```
43              t1 = (vertex(rightV,1)-vertex(downV,1))^2 +...
44                  (vertex(rightV,2)-vertex(downV,2))^2 +...
45                  (vertex(rightV,3)-vertex(downV,3))^2;
46              % \ distance:
47              t2 = (vertex(currentV,1)-vertex(diagV,1))^2 +...
48                  (vertex(currentV,2)-vertex(diagV,2))^2 +...
49                  (vertex(currentV,3)-vertex(diagV,3))^2;
50              %test case on triangles
51              if (t1>t2)  % \ triangles
52                  faces(faceIdx,1) = currentV; %right triangle
53                  faces(faceIdx,2) = diagV;
54                  faces(faceIdx,3) = rightV;
55                  faceIdx = faceIdx+1;
56                  faces(faceIdx,1) = currentV; %left triangle
57                  faces(faceIdx,2) = downV;
58                  faces(faceIdx,3) = diagV;
59                  faceIdx = faceIdx+1;
60              end
61              if (t2>=t1) % / triangles
62                  faces(faceIdx,1) = currentV; %left triangle
63                  faces(faceIdx,2) = downV;
64                  faces(faceIdx,3) = rightV;
65                  faceIdx = faceIdx+1;
66                  faces(faceIdx,1) = downV; %right triangle
67                  faces(faceIdx,2) = diagV;
68                  faces(faceIdx,3) = rightV;
69                  faceIdx = faceIdx+1;
70              end
71          end
72      end
73
74      %add cube side faces
75      for i=1:n-1
76          faces(faceIdx,1) = i+1; %upper left
77          faces(faceIdx,2) = i;
78          faces(faceIdx,3) = n*n+1;
79          faceIdx = faceIdx+1;
80          faces(faceIdx,1) = (i+1)*n; %lower left
81          faces(faceIdx,2) = i*n;
82          faces(faceIdx,3) = n*n+2;
83          faceIdx = faceIdx+1;
84          faces(faceIdx,1) = n*(n-1)+i; %lower right
85          faces(faceIdx,2) = n*(n-1)+i+1;
86          faces(faceIdx,3) = n*n+3;
87          faceIdx = faceIdx+1;
88          faces(faceIdx,1) = (i-1)*n+1; %upper right
89          faces(faceIdx,2) = i*n+1;
90          faces(faceIdx,3) = n*n+4;
91          faceIdx = faceIdx+1;
92      end
```

```
93      faces(faceIdx,1) = n; %left
94      faces(faceIdx,2) = n*n+1;
95      faces(faceIdx,3) = n*n+2;
96      faceIdx = faceIdx+1;
97      faces(faceIdx,1) = n*n; %down
98      faces(faceIdx,2) = n*n+2;
99      faces(faceIdx,3) = n*n+3;
100     faceIdx = faceIdx+1;
101     faces(faceIdx,1) = (n-1)*n+1; %right
102     faces(faceIdx,2) = n*n+3;
103     faces(faceIdx,3) = n*n+4;
104     faceIdx = faceIdx+1;
105     faces(faceIdx,1) = 1; %top
106     faces(faceIdx,2) = n*n+4;
107     faces(faceIdx,3) = n*n+1;
108     faceIdx = faceIdx+1;
109
110     % cube base faces
111     faces(faceIdx,1) = n*n+2;
112     faces(faceIdx,2) = n*n+1;
113     faces(faceIdx,3) = n*n+4;
114     faceIdx = faceIdx+1;
115     faces(faceIdx,1) = n*n+4;
116     faces(faceIdx,2) = n*n+3;
117     faces(faceIdx,3) = n*n+2;
118
119     %resize faces matrix
120     facesFinal = zeros(faceIdx,3);
121     facesFinal(1:end,1:3) = faces(1:faceIdx,1:3);
122
123     % scale vertex positions (m to mm)
124     vertex = vertex./1000;
125
126     %call write to PLY
127     write_ply(vertex,facesFinal,'Mesh.ply');
128 end
```

Bibliography

[Ber13] Adrien Bernhardt. *Modèles pour la création interactive intuitive*. Theses, Université de Grenoble, 2013.

[Bro11] André R. Brodtkorb. Efficient shallow water simulations on gpus. SIAM Conference on Mathematical & Computational Issues in the Geosciences, March 2011. Presentation Slides. Retrieved from `http://babrodtk.at.ifi.uio.no/files/publications/brodtkorb_gs11.pdf`.

[CH14] Paul Conolly and Robin Hogan. Shallow water model, October 2014. Computer Practical. Retrieved from `http://personalpages.manchester.ac.uk/staff/paul.connolly/teaching/practicals/material/shallow_water/swe_notes_pc.pdf`.

[DM08] Clint Dawson and Christopher Mirabito. The shallow water equations, 2008. Retrieved from `http://users.ices.utexas.edu/~arbogast/cam397/dawson_v2.pdf`.

[EBC+15] Even Entem, Loïc Barthe, Marie-Paule Cani, Frederic Cordier, and Michiel Van de Panne. Modeling 3d animals from a side-view sketch. *Computers & Graphics*, 46:221–230, 2015.

[GWS13] M. Guan, N.G. Wright, and P.A. Sleigh. A robust 2d shallow water model for solving flow over complex topography using homogenous flux method. *International Journal for Numerical Methods in Fluids*, 73(3):225–249, 2013.

[Har04] Mark Harris. Fast fluid dynamics simulation on the gpu. In Randima Fernando, editor, *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, chapter 38. Addison-Wesley Professional, March 2004.

[Hol75] John Holland. Adaptation in natural and artificial systems. University of Michigan Press, 1975.

[Kos02] R. John Koshel. Enhancement of the downhill simplex method of optimisation. *2002 International Optical Design Conference Technical Digest*, 2002.

[Man10] Wolf Mangelsdorf. Structuring strategies for complex geometries. *Architectural Design*, 80(4):40–45, 2010.

[MCG03] Matthias Mueller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. *Proceedings of ACM SIGGRAPH Symposium on Computer Animation*, pages 154–159, 2003.

[MF08] Matthias Müller-Fischer. Fast water simulation for games using height fields. GDC2008, 2008. Retrieved from `http://matthias-mueller-fischer.ch/talks/GDC2008.pdf`.

[Mol] Cleve Moler. Chapter 18: Shallow water equations. Experiments with MATLAB. MathWorks. Retrieved from `http://www.mathworks.com/moler/exm/chapters.html`.

[NISA07] Andrew Nealen, Takeo Igarashi, Olga Sorkine, and Marc Alexa. FiberMesh: Designing freeform surfaces with 3D curves. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, 26(3):article no. 41, 2007.

[NM65] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965.

[PTC+15] Jesús Pérez, Bernhard Thomaszewski, Stelian Coros, Bernd Bickel, José A. Canabal, Robert Sumner, and Miguel A. Otaduy. Design and fabrication of flexible rod meshes. *ACM Trans. Graph.*, 34(4):138:1–138:12, July 2015.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Numerical recipes in c: The art of scientific computing. chapter 10.4, pages 408–412. Cambridge University Press, 1992.

[Qui12] Michael Quinten. *Appendix D: Downhill Simplex Algorithm*, pages 199–200. Wiley-VCH Verlag GmbH & Co. KGaA, 2012.

[Rob11] Colin Richard Robinson. Shallow water equations, December 2011. MATLAB Project for Syracuse University. Retrieved from `http://homes.civil.aau.dk/jen/VisualizationExercise3/ShallowWaterEquations.pdf`.

[Sta99] Jos Stam. Stable fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 121–128, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[TCG+14] Bernhard Thomaszewski, Stelian Coros, Damien Gauge, Vittorio Megaro, Eitan Grinspun, and Markus Gross. Computational design of linkage-based characters. *ACM Trans. Graph.*, 33(4):64:1–64:9, July 2014.

[TH] Nils Thuerey and Peter Hess. Shallow water equations. Chapter 11 of course notes. Retrieved from: `http://liris.cnrs.fr/alexandre.meyer/teaching/m2pro_charanim/papers_pdf/coursenotes_shallowWater.pdf`.

[TSS94] E.F. Toro, M. Spruce, and W. Speares. Restoration of the contact surface in the hll-riemann solver. *Shock Waves*, 4(1):25–34, 1994.

[UKSI14] Nobuyuki Umetani, Yuki Koyama, Ryan Schmidt, and Takeo Igarashi. Pteromys: Interactive design and optimization of free-formed free-flight model airplanes. *ACM Trans. Graph.*, 33(4):65:1–65:10, July 2014.

[WJT11] Yun Wu, Hai-xin Jiang, and Wei Tong. Generalized lax-friedrichs schemes for linear advection equation with damping. In Baoxiang Liu and Chunlai Chai, editors, *Information Computing and Applications*, volume 7030 of *Lecture Notes in Computer Science*, pages 305–312. Springer Berlin Heidelberg, 2011.