Universiteit Utrecht

Master Thesis

Fluid Simulation on Point Clouds

July 3, 2019

Student:
Jack van der Drift

ICA:
ICA-4098978

Supervisor:
Dr. A. Vaxman

Institution:
University of Utrecht

Contents

Abstract

With the effects of climate change becoming more and more apparent in the form of drought, heavy rainfall and the melting of the polar ice caps research into more efficient algorithms for fluid simulation become more important. For cases such as heavy rainfall and the melting of the polar ice caps it would be convenient to be able to use real life laser scans of landscapes to predict what damage they will do and where we need to take precautions.

The purpose of this thesis is to come up with a method to directly use fluid simulation on point clouds. We do this to skip the expensive reconstruction step which is normally done and simplify the collision detection between the marker particles and object.

Our algorithm combines the MLS algorithm with a Eulerian fluid simulation. For each grid point we compute the MLS values and use these to determine how solid a grid cell is and keep the marker particles from going into the point cloud.

Our experiments show that it is possible to directly perform fluid simulation on point clouds using our method, but that the performance of our algorithm is highly dependent on the type of scene and the used parameters.

List of Abbreviations

AHN	Actueel Hoogtebetand Nederland
BVH	Bounding Volume Hierarchy
GI Science	Geographical Information Science
DEM	Digital Elevation Model
SPH	Smoothed Particle Hydrodynamics
MPS	Moving Particle Semi-Implicit
AABB	Axis-Aligned Bounding Box
MLS	Moving Least Squares

1 Introduction

With climate change causing more extreme weather conditions than ever, the need to be prepared is becoming more prominent. This is especially true for countries which are prone to catastrophic disasters such as floods, for which the loss of life and property are preventable if the necessary precautions have been made beforehand. In order to be prepared for such disasters it will be beneficial if there exists a program that uses flood simulation to accurately predict which parts will be flooded and where flood defenses need to be raised, placed or, reinforced. This would especially benefit countries such as The Netherlands and cities such as New Orleans.

While current flood simulations, such as [Ghazali and Kamsin, 2008], manage to create realistic flood simulations, the scenes they use are sub-optimal and limited. One of the reasons why current flood simulations are problematic is that we need to use a geometric model for the area it covers, mostly in the form of triangle meshes.

A reason why the usage of solid meshes is the leading method in flood simulations is because flood simulations are derived from fluid simulations, that are in turn developed for meshes or grids. Another reason is the amount of research done with meshes. This makes it easier to find papers which have already done something of value and relevance compared to other methods.

Using point clouds for flood simulations has a few advantages over meshes. Normally the meshes used in such simulations are reconstructed from raw point clouds, so by skipping this step we save on costly reconstruction time. Furthermore, we avoid the artifacts created by the reconstruction that distort the geometry. And finally, we use a simpler data structure that makes it easier to create an efficient system for flood simulation. With real scans of a city, it is then possible to use our simulations for real-life applications such as flood prevention or damage estimation for a storm.

We propose a novel fast large-scale flood simulation algorithm that operates on huge point-cloud scenes. In order for our simulation to be interactive we use fast collision detection methods for point clouds, multi-resolution representations of point clouds, and develop methods to optimize the fluid computations for point cloud scenes. The point clouds scenes we use are a combination of landscapes, statues and self created objects. The landscapes are obtained from the AHN database over at `ahn.nl`, the statues come from the Stanford repository [Standford, 2014] and for the self created point clouds we used Blender. The AHN database is a database that consists of areal scans of The Netherlands divided into 31,25 km^2 pieces. These scans have been made by either a plane or helicopter equipped with a laser scanner. The scans from the Stanford repository have been made in the same manner but with a stationary and smaller laser scanner.

To summarize, our main contributions are:

- Introducing a novel way of using point clouds as scenes for flood simulation.
- A highly parallelizable algorithm based on distance fields, for collision checking between the liquid and the point cloud scenes.

2 Related Work

We explore the state-of-the-art research from two unrelated fields: flood simulation, and point cloud processing.

For the flood simulation part we are looking at state-of-the-art papers that introduce methods to simulate floods. This also includes flood simulations based on geographical data, such as height maps or sewer layouts.

For the point cloud part, we look at methods to scale down the amount of points and methods for fast collision detection, which both contribute to a real-time simulation.

2.1 Flood Simulation

When it comes to fluid simulations, there are two main approaches to simulate the fluid: Eulerian and Lagrangian. The first approach, Eulerian, tracks the fluid flow at fixed points in a scene. You can compare this method to standing on the ground and keeping track of the airplanes flying overhead. This approach usually incorporates a grid to keep track of the flow. The second approach, Lagrangian, tracks the fluid itself. This can be seen as being in the cockpit of the aircraft and being able to see its speed and all other variables, such as position. In order to represent the fluid this approach usually uses particles.

These approaches work the same for flood simulation, since it is subcategory of fluid simulations. Fluid simulation works on all kind of things that can be modeled as a fluid, such as fire, gasses and liquid, while flood simulations only work with liquids. This allows us to use simpler formulas, neglecting temperature and state changes. Another difference is that fluid simulations are usually small scale and thus have to account for viscosity. Flood simulations don't need a specific formula for viscosity since their impact is dwarfed by errors in the simulation. Another advantage is that in flood simulation we only perform the simulation on static scenes. This allows us to drop the formulas for One-way and Two-way coupling, which are used to model the influence of moving objects on the fluid and visa versa. A disadvantage of flood simulation compared to fluid simulation is the scale. For more approaches and a more detailed explanation on the basics of fluid simulation we refer to [Bridson, 2015].

As discussed before, both the Eulerian and the Lagrangian method can be used in a flood simulation, but each approach has their advantages and disadvantages when it comes to their usage in flood simulations. In the following sections we research which approach or combination of approaches works best for our application.

2.1.1 Particle-based simulation

There are two main approaches to particle-based flood simulation: Smoothed Particle Hydrodynamics (SPH) [Monaghan, 1992] and Moving Particle Semi-Implicit (MPS) [Koshizuka and Oka, 1996]. Both methods are similar, with the main difference being that MPS is based upon a local averaging process and SPH uses a kernel to smooth the results, which in turn gives SPH a smoother water surface. This is usually why SPH is used more than MPS when it comes to flood simulations, since we want a nice looking result in the end. For a better and more in depth explanation we refer to the original papers.

[Ghazali and Kamsin, 2008] uses SPH as a replacement for Water level simulations. These simulations only change the height of the water for the whole scene. By using SPH they get rid of the random puddles which form in the water level simulation. In addition, they use a satellite photo and a digital elevation model (DEM) created from LIDAR data to create their model of the city Kuala Lumpur. However, they do not use the LIDAR data (point clouds) directly, but rather have to reconstruct a mesh. This is costly and could introduce errors in the model.

[Chládek and Durikovic, 2010] presents a SPH based flood simulation that uses the distance to an object instead of the actual intersection to handle the collision between water and the scene. By switching to this method they increase their FPS by about 4 times, depending on the scene and time step. This paper suggest that it is beneficial to look at collision methods such as distance fields to increase the FPS, which is beneficial if you want to keep the fluid simulation interactive. We will be looking at distance fields and other collision detection methods in 2.2.2.

2.1.2 Grid-based simulation

The basic principle behind grid-based flood simulations is using a uniform grid encompassing the region where the simulation is performed with the same size as the scene in which you keep track of the flow of the fluid. This isn't very efficient since most of these grid cells could be empty or not change state at all, as in change from solid to a fluid, visa versa or from fluid to air and visa versa; consider the air above the water or the water at the bottom of the ocean.

[Irving et al., 2006] proposes a method that solves this to some degree. Instead of using a uniform grid with uniform cells they introduce tall grid cells, that can be seen as multiple uniform cells with the same values stacked on top of each other. This allows for scheduling processing power around areas of interest such as the edge of the water, or where it is in contact with moving objects.

[Chentanez and Müller, 2011] further improves this idea by introducing some constraints that reduce the complexity of the method of [Irving et al., 2006], while retaining the flexibility of the method. They do this by allowing only one tall cell per column and making sure it is the lowest water cell in the column. This allows them to perform all the computations on the GPU, which in turn gives them real-time performances in their simulations, which is something that can be of use to our application.

2.1.3 Geographical-based simulation

There are flood simulation methods on real-life scenes, but they use alternative approaches. All following papers use geographical data to perform their flood simulation.
[Shahapure et al., 2010] uses the Finite Element Method combined with a overland flow model created from a Digital Elevation Model (DEM) of the area. [Leandro et al., 2009] which is more focused on surface flows combined with sewer layout. For a better overview of surface flow methods combined with a map of a village we refer to [Hunter et al., 2008].

2.2 Point Clouds

Point clouds are becoming increasingly more popular as geometry representations due to advanced processing methods and widespread availability. This is largely due to the developments in scanners and computers. Scanners are getting cheaper, while they increase in precision and decrease in size. The size decrease allows them to be mounted on vehicles such as airplanes, helicopters, and even vans. The increased precision allows for more accurate and more meaningful simulations and other applications, such as digitizing objects and buildings.

However, increased precision comes at the cost of increased amounts of data. While this does give us highly detailed point clouds of large sceneries, they are not usable in fluid simulations without some form of pre-processing that reduces the amount of points and thus the amount of computations the fluid simulation needs to do. These computations mostly consist of the collision checks which need to be done between the fluid and all the points of the point cloud. So when we want to use large point clouds, like the landscapes scanned by AHN, we first need to scale them down or perform some sort of pre-processing in order for them to be useful in our application,

like they do in [Goswami et al., 2013] and [Klein and Zachmann, 2004]. In order to select the best method or combination of methods we need to look at the following methods: scaling down the amount of points through subsampling, using spatial divisions to make processing of the points clearer and more usable when it comes to fast interactions or multi-resolution methods, like entropy-based reduction or k-clustering, to get the desired amount of points, like in [Eckart et al., 2016].

However, none of the methods are perfect. For example when we sample the point cloud we get decreased accuracy which makes it impossible to run accurate simulations on the data, and when we use trees, like they use in [Klein and Zachmann, 2004], our computation time for simple calculations goes up. This in turn makes it harder to achieve an interactive application. Note that using a tree does decrease the time needed to interact with the point cloud, but thanks to the sheer amount of points in the tree, it still takes to long for interactive usage. That's why in most cases a combination of the two gets used, with on each level of a tree a sampled representation of the point cloud. This makes it possible to stop when we are spending too much time on one pass through the tree and also return an usable result.

In the following sections we will go more into detail on which methods are available and which are of use to us.

2.2.1 Scalability

As previously discussed there are two distinct ways to deal with massive amounts of data: subsampling and the use of data structures such as trees.

When it comes to subsampling there are many different methods that can be used, but not all of them are applicable to point clouds. In [Lee et al., 2001] two such methods get explained and tested against some other methods:

- Uniform grids: after creating a 2D uniform grid spanning the point cloud they pick the median point to represent all points in the grid square.
- Non-uniform grids: same principle as with the uniform grid except that they take into account the amount of points, in order to prevent the loss of corners and other details.

For a more detailed explanation and test results we refer to [Lee et al., 2001].

[Klein and Zachmann, 2004] proposes a method which combines multilevel sampling with an octree. On each layer of the octree we have a few point and a circle representing the original surface/point cloud. The lower we traverse the octree the more precise the representation will be. This layered system allows them to stop at any moment and generate an usable result. This is

very useful when we are trying to create an interactive application, since we can stop when have spend too much time checking for collisions.

In [Goswami et al., 2013], while mostly concerned with the rendering of point clouds, they did test some interesting methods to sample a point cloud. The methods they tested are normal deviation clustering, an Entropy-based reduction, and a k-clustering algorithm. They tested each sample method in combination with a multi-way Kd-tree, which uses the same idea as [Klein and Zachmann, 2004]. That means that each level of the Kd-tree has a different level of detail of the point cloud with the most detailed representation in the bottom nodes.

[Oehler et al., 2011] tries to represent parts of the point clouds as planes. They do this by creating an octree and for each level they extract the surface normals which they then group and use to fit a plane with RANSAC. While it does lower the processing time during each cycle because you cut away a lot of points, it removes details from the scene.

In [Elseberg et al., 2011] and [Elseberg et al., 2013] they propose a method that instead of compressing the point cloud, compresses the octree. However they didn't look at combining it with sampling the point cloud, but this can of course be introduced to the already memory efficient octree. This will make the octree use even less memory, since they now keep all points inside the nodes. [Schnabel and Klein, 2006] works similarly but uses a prediction method based on local surface approximations to compress the point cloud.

In [Eckart et al., 2016] they do something completely different than the papers above. Instead of using a spatial division or a sampling method to reduce the point cloud they use multiple Gaussian mixtures models, which can be re-sampled to recreate an approximation of the point cloud. The main advantages of these methods are: easy to generate multiple levels of detail and the possibility to scale the point cloud without much loss of detail.

Combining the above methods with the knowledge that we will need a tree for both the collision detection of the simulation and for scaling down the points even more, it is smart to combine these two to save on pre-computing times. The smart choice will then be the method proposed by [Klein and Zachmann, 2004] or any of the other spatial division papers. Another benefit this type of method adds is the possibility to stop traversing anywhere in the tree and still return a result, which is beneficial for a interactive application.

2.2.2 Collision Detection

A major demand of our application is that it has to run in real-time. However, when we have millions of points to check each frame, it will take a lot of time. Points themselves have a fairly fast and easy method to check for collisions which helps a bit, but we still need to take a look at collision detection methods which are fast, parallelizable and work on point clouds.

In [Figueiredo et al., 2010] they split the scene up in voxels and for each object in a voxel they create an R-tree of Axis-Aligned Bounding Boxes (AABB). This cuts down the amount of collision tests enormously. This approach could be applied to our application since we might be able to combine the AABB's with the grid we use for the flood simulation. [Schauer and Nüchter, 2014] instead opts to use a Kd-tree for the whole scene instead of an R-tree for each separate voxel, which allows for parallel operations who decrease the amount of time needed to test for collisions.

[Klein and Zachmann, 2004] combines a binary tree where each node stores a sample of the point cloud and a sphere which represents the surface of the object. This might work really well in our approach because it both subsamples the point cloud and decreases the time needed for collision tests.

In [Fuhrmann et al., 2003] they propose to use distance fields as a reduction of points clouds. This comes down to each cell knowing how far away it is from an object, to handle collision detection between deformable and rigid objects. [Bender et al., 2014] takes this idea and applies it to point clouds and combines it with a spatial octree to further decrease the amount of time spent on collision tests. Using a distance field usually doesn't make sense in large scenes where things are moving, but because we have a static scene we only need to create the distance field once.

3 Background

3.1 Fluid simulation

Each flood or fluid simulation, which simulates incompressible fluids, is governed by the Navier-Stokes equations. These equations are supposed to hold all throughout the fluid and are written as followed:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{g} + v \nabla \cdot \nabla \vec{u}, \tag{1}$$

$$\nabla \cdot \vec{u} = 0. \tag{2}$$

Here the letter $\vec{u}$ stand for the velocity of the fluid, as in (u, v) or (u, v, w) depending on whether the simulation is 2D or 3D. t stands for time. ρ stands for density of the fluid, which is 1000 kg/m^3 for water. p stands for the pressure the fluid exerts per unit area. The letter $\vec{g}$ is the acceleration due to outside forces. v stands for kinematic viscosity. However, as discussed before we can drop this from the formula, giving us the Euler equations:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{g} \tag{3}$$

$$\nabla \cdot \vec{u} = 0. \tag{4}$$

These will be the formulas (3, 2) we are going to approximate in our flood simulations. The Euler equation can be split in three parts:

- The material derivative ($\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u}$), which determines the flow as result of the existing flow and the change in velocity at the current location.
- The pressure gradient ($\frac{1}{\rho} \nabla p$), which represents the flow as result from the pressure in the fluid
- The external body forces ($\vec{g}$), which includes things such as gravity, wind and forces coming from objects.

When we implement a fluid solver using the Eulerian equations, we split them into three steps that are solved iteratively, and are individually easier than the full equation:

- Advection: which calculates the movement of the fluid.
- Body Forces: which applies the external forces working on the fluid.
- Pressure: which makes sure the fluid stays incompressable and consequently divergence free.

For a more detailed explanation of fluid simulations we refer to [Bridson, 2015].

3.2 Grids

In order to keep track of the values needed to solve these parts we use a staggered grid denoted as Marker-and-Cell (MAC) grid [Bridson, 2015]. This grid usually consist of uniform cells, which each keep track of the pressure in the middle of a grid cell and the velocity on either mid-edges of the respective axis (in 2D) or faces between cells (in 3D). When the scenes get too large to initialize a grid that encapsulates the whole scene from the start we use a sparse blocked grid, which dynamically adjusts its size to only accommodate the current state of the fluid including a few buffer cells around it. For an example of an 2D and 3D MAC grid cell we refer to Figures 1 and Figures 2.

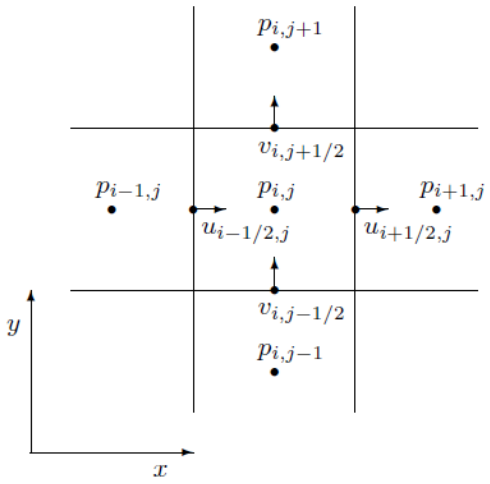


Figure 1: One cell from a 2D MAC grid

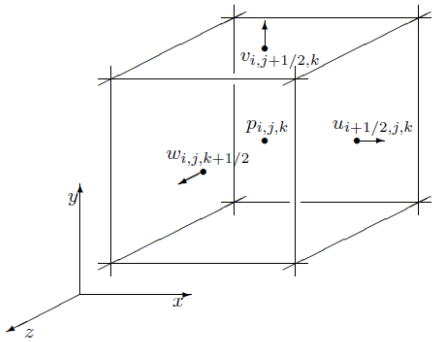


Figure 2: One cell from a 3D MAC grid

The tall cells from [Irving et al., 2006] and [Chentanez and Müller, 2011] are fairly similar to the standard MAC grid cell, as you can see in Figure 3. They store the velocities in the respective axis (in 2D) or faces between cells (in 3D) and keep track of the pressure. The main difference is that a tall cell is multiple uniform MAC cells tall and that the velocities and pressure values get stored for two locations, namely at the top and bottom uniform cells that the tall cell replaces. Interpolation can then be used to calculate the values for the rest of the tall cell.

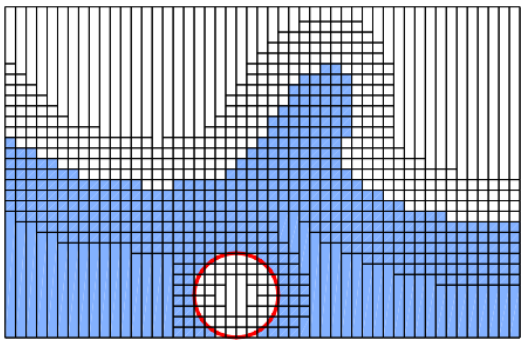


Figure 3: Tall grid cell example from [Irving et al., 2006]

3.3 Point clouds

A point cloud is a collection of unordered points in 3D Euclidean space, meaning each point has a x, y, and z coordinate relative to an origin. Together these points can represent the surface of an object. Point clouds can be created using laser scanners. Such a laser scanner takes measurements at a constant interval, which can be adjusted for the scale of the object. For example with small objects such as coins we want precise measurement and thus have multiple measurements per cm^2, but for surfaces such as marina's, we only need capture the outline and thus capture with only a few measurement per m^2.

In order to interact with the point cloud we need a spatial division to cut down the amount of points we need to check for operations such as collision detection. A spatial division splits the points based on their location and stores which point is on which side of the split in so called nodes. The structure keeps splitting the points, while maintaining the previous nodes. These subsequent splits get added to the structure as child nodes, to the nodes above them in the structure. This gets done iteratively until some criteria is met, this usually is the amount of points left in the node. When it comes to spatial divisions for point clouds there are two data structures which are the most common. They are the octree and the Kd-tree. These structures both are recursive subdivisions of space, with the difference being the amount of children each node has and how the subdivision of the point cloud is done. An octree subdivides around the center point of the current node to create eight child nodes, while a Kd-tree splits around an axis and creates two child nodes.

A distance field is a grid where each cell in the grid knows the shortest distance from its location to the nearest cell with another property, which in our case is a cell with a point from the point cloud in it. Normally each time an object moves this grid needs to be recalculated. Because we use the point cloud as a static scene we only need to calculate this grid once. A distance field can be calculated using different distance measures, some typical approaches are the Manhattan Distance and the Euclidean Distance.

4 Methology

Our method can be split up into two distinct parts, in one we do the pre-computations needed for the fluid simulation, and in the other the fluid simulation itself.

During our pre-computation part we take a point cloud with normals and transform it into something on which we can do fluid simulation. For this we want something which can simply tell us if we are inside or outside of the point cloud, for this we use the MLS algorithm. The MLS is a function which gives us values depending on how far from the point cloud or how far inside the point cloud we are. The zero set of the MLS thus means that we are on the edge of the point cloud. Furthermore we have chosen to think of negative values as being inside the point cloud and positive values as outside. More on this later in Section 4.1.

From these MLS values it is also possible to derive how solid each grid cell is. We use this to more accurately simulate the fluid flow. How solid a grid cell is will from now on will be referred to as the solid fraction of a grid cell. A more detailed explanation follows in Section 4.2.

Now during the fluid simulation itself we will be using both the MLS values and the solid fractions that we computed. First up, the solid fractions get used in the pressure projection to determine the correct amount of fluid flow through a grid cell and they get used to correctly determine how much influence each neighbouring cell's fluid values have on a grid cell. This will be explained in more detail in Section 4.3.

The MLS values get used each frame to check if all marker particles ended up in a correct location, meaning not inside of the point cloud. If a marker particle did end up in the point cloud we use the MLS gain to determine its last correct position. For more details we refer to Section 4.4.

All above sections combined give us a fluid simulation algorithm which can use point clouds to represent the scene. Our method works for both 2D and 3D, but we omit the explanation of the 2D part since this comes down to dropping the z coordinate from the 3D simulation.

We implemented our method in C++ using Visual Studio 2015 and used the following libraries:

- Eigen: used for matrix calculations
- laszip: used for loading point cloud with *.las or *.laz extension
- opengl32: used for basic visualizing the fluid and the point clouds
- PolyVox: used for better looking visualization of the fluid
- SDL2: used for window and keyboard input handling

4.1 MLS computation

We compute the MLS using a version closely related to "Interpolating and Approximating Implicit Surfaces from Polygon Soup" [Shen et al., 2004]. Meaning for each grid point we try and fit a polynomial of degree N. The mathematical notation for this polynomial can be found in Equation 5. Note that we compute a separate polynomial for each grid point ($\mathbf{q}$) and not one for all the grid points.

$$f(x,y,z) = \sum_{i,j,k \geq 0, i+j+k \leq N} a_{ijk} x^i y^j z^k \tag{5}$$

For 3D, the polynomial simplifies:

$$f(x,y,z) = a_{000} + a_{100}x + a_{010}y + a_{001}z + a_{110}xy + a_{011}yz + a_{200}x^2 + a_{020}y^2 + a_{002}z^2$$

Before we start the calculation we need to know the following: For each $\mathbf{q}$ we have a subset of points from the point cloud which are in range $\{\mathbf{p}_i\}$ and their corresponding normals $\{\mathbf{n}_i\}$. We also have a scalar parameter ϵ which we set to the width of a grid cell. For each point in $\mathbf{p}_i$ we create three points, one which is the original point, one where we add the normal of that point times ϵ to the original point and one where we substract the normal time ϵ, we will call this collection $\{\mathbf{c}_i\}$ from now on. These points will later be used to fill in some matrices.

Now in order to solve this: we first create a vector $\mathbf{a}$ which holds our unknown coefficients a_{000} to a_{002}.

Secondly we create a vector $\mathbf{d}$, which holds the ϵ for each of the point in $\{\mathbf{c}_i\}$. So for the first three entries in the vector we will have the values 0, $+\epsilon$ and -ϵ. This vector has the length of $\{\mathbf{p}_i\}$ times three.

Next up we create the a matrix called $\mathbf{C}$. For each point in $\{\mathbf{c}_i\}$ we add a row in which the values correspond to the x, y, z values in the polynomial, so for point (2,3,4) we will have a row of (1, 2, 3, 4, 6, 12, 4, 9, 16).

Finally we create a diagonal matrix $\mathbf{W}$ with on the diagonal $\theta(|\mathbf{q} - {}_i|)$ where θ is a weight function for which we took the Wendland weights function

(Equation 6). For h we picked the value of $(sr*cs)^2*3$, where sr is the search radius for the MLS and cs is the cell size of a grid cell.

$$\theta(r) = (1 - \frac{r}{h})^4(4\frac{r}{h} + 1) \tag{6}$$

We need to know all this, because what we are actually trying to do is compute a polynomial line which best represents $\{\mathbf{p}_i\}$. This means finding the parameters a_{ijk} which minimize the differences $|f(\mathbf{c}_m) - d_m|^2$. For this we use the following system:

$$\mathbf{a} = argmin(|\sqrt{\mathbf{W}}(\mathbf{Ca} - \mathbf{b})|^2)$$

,

where:

$$\mathbf{a} = \begin{pmatrix} a_{000} \\ a_{100} \\ \dots \\ a_{002} \end{pmatrix}$$

$$\mathbf{d} = \begin{pmatrix} d_0 \\ d_1 \\ \dots \\ d_{m-1} \end{pmatrix}$$

$$\mathbf{C} = \begin{pmatrix} (x_0)^0(y_0)^0(z_0)^0 & (x_0)^1(y_0)^0(z_0)^0 & \dots & (x_0)^0(y_0)^0(z_0)^2 \\ (x_1)^0(y_1)^0(z_1)^0 & (x_1)^1(y_1)^0(z_1)^0 & \dots & (x_1)^0(y_1)^0(z_1)^2 \\ \dots & \dots & \dots & \dots \\ (x_{m-1})^0(y_{m-1})^0(z_{m-1})^0 & (x_{m-1})^1(y_{m-1})^0(z_{m-1})^0 & \dots & (x_{m-1})^0(y_{m-1})^0(z_{m-1})^2 \end{pmatrix}$$

$$\mathbf{W} = diag[\theta(|\mathbf{q} - \mathbf{c}_0|), \theta(|\mathbf{q} - \mathbf{c}_1|), \dots, \theta(|\mathbf{q} - \mathbf{c}_m|)]$$

In order to solve this equation for $\mathbf{a}$ we turn it into Equation 7 which can be solved using standard techniques which are used to solve $Ax = b$.

$$\mathbf{C}^T\mathbf{WCa} = \mathbf{C}^T\mathbf{Wd} \tag{7}$$

After we solved Equation 7 we then use the found coefficients of $\mathbf{a}$ to fill in Equation 5. We then fill in this equation with the coordinates of the point for which we calculated the polynomial to find the corresponding MLS value. We repeat this until we went over all grid points. At the end we fill in the MLS values for the outer walls, to prevent fluids from going in. We don't need to do any calculation since we make the outer walls one grid cell wide. This gives us minus one grid cell width as MLS value for the outer most grid points, zero for the second layer of grid points and plus one grid cell width for the third layer of grid points.

4.2 Solid fraction computation

To get more detail in our fluid simulation we use fractions to tell the fluid simulation how solid a grid cell is, in this we differ from normal Eulerian fluid simulations, which normally only deal in absolutes so either fully solid or fully fluid.

We compute the solid fraction for each grid cell face, this way it can be used to determine the flow in and out of a grid cell face. The solid fraction can be calculated using the four MLS values of a grid cell face and a combination of triangle occupancy (Equation 8) and trapezoid occupancy (Equation 9).

$$0.5 * in * in/((out_1 - in) * (out_2 - in)) \tag{8}$$

$$0.5 * (-in_1/(out_1 - in_1) - in_2/(out_2 - in_2)) \tag{9}$$

There are five scenarios when it comes to computing the solid fractions:

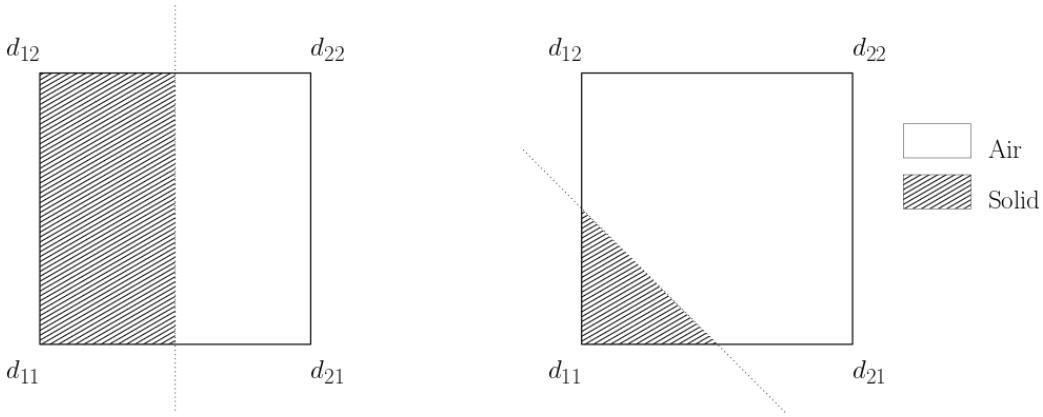
- All MLS values are positive, all corners are outside the object. This gives us a solid fraction of 0.
- 3 MLS values are positive, one corner is inside the object. We calculate this using triangle occupancy
- 2 MLS values adjacent are positive, two corners next to each other are inside the object. Here we use trapezoid occupancy
- 2 MLS values opposed are positive, two corners opposite of each other are inside the object. Here we use: triangle occupancy for one side and triangle occupancy for the other side.
- 1 MLS value is positive, three corners are inside the object. This can be calculated the opposite of before, so: 1 - triangle occupancy
- 0 MLS values are positive, meaning the whole grid cell is inside the object. This gives us a solid fraction of 1.

So in the case of Figure 4, the trapezoid occupancy for Figure 4a becomes:

$$0.5 * (-d_{11}/(d_{21} - d_{11}) - d_{12}/(d_{22} - d_{12})),$$

and the triangle occupancy for Figure 4b becomes:

$$0.5 * d_{11} * d_{11}/((d_{12} - d_{11}) * (d_{21} - d_{11})).$$



(a) Trapezoid occupancy (b) Triangle occupancy

Figure 4: Occupancy examples

4.3 Fluid simulation

For the fluid simulation part we made some changes to the original implementation which can be found in [Bridson, 2015]. We will be explaining the whole fluid simulation algorithm, so both the parts which we left untouched and the parts in which we made changes to make it work for point clouds.

To keep track of the variables in the fluid simulation, such as velocity and pressure we use the following variables:

- array u, the fluid velocity for each face in the x-axis
- array v, the fluid velocity for each face in the y-axis
- array w, the fluid velocity for each face in the z-axis
- array p, the pressure in each grid cell

We start the simulation with a divergence free velocity field, meaning there is no difference between flow in and flow out. So each previously mentioned array will start with all zero values.

Then for each step of the fluid simulation we first start with picking a time-step. This can be done according to the velocities in the simulation or by simply picking a constant time-step, which we did in our case. The Constant time-step depends on the size of the scene and the point cloud used, but in general we used a time-step of 0.01s.

After we picked a time-step we check if we have not yet reached the maximum amount of marker particles, this is a limit we set ourselves. If we have not reached the limit we spawn marker particles for each of the sources we set at the start. Setting sources simply comes down to specifying a location and velocity for each source. The velocity specifies at what speed the fluid flows from that source.

After we spawned the marker particles, we set each grid cell to one of the following types: solid, fluid or air. The solids can be determined from the solid fractions, as in when each face of a grid cell is completely solid the whole cell is completely solid. The water cells get determined by the marker particles, as in loop over all the marker particles and mark the grid cell they are in as fluid. This leaves us with determining the air cells, which are all the leftover grid cells.

4.3.1 Advection

The next step in the fluid simulation is advecting the velocity fields. We do this using "forward" Euler going backward in time. The equation for this is as follows:

$$\vec{x}_P = \vec{x}_G - \Delta t \vec{u}(\vec{x}_G),$$

where $\vec{x}_G$ is the location where we start, $\vec{u}(\vec{x}_G)$ the velocity at that location, Δt the time-step of the current frame and $\vec{x}_P$ the new location. In our case $\vec{x}_G$ corresponds to the center of a grid cell face, for which we know the velocity by looking at either u, v or w depending on which velocity field we are advecting. This however, is only the first part of the advection algorithm, since we now only have a location in the grid.

The next step is to check if we did not end up inside a solid. For this we use the MLS values. For $\vec{x}_P$ we calculate its MLS value by interpolating from the MLS grid. For this value we then check if it is positive or if it is negative. If this value is positive we don't have to do anything, but if it is negative we need to find the last viable position along its trajectory during this time step. We do this as following:

- First we determine the MLS value the particle had before we moved it.
- Then we use that value with the MLS value after moving to determine how much of the distance we moved this time-step was inside of the point cloud.
- We then multiply the distance we moved with this percentage to determine how much we need to move the particle back in order for it to be on the edge of the point cloud. This will then be its correct position. To prevent it from getting stuck in the wall, we move the marker particle a slightly more back than is necessary.

In Formula form this comes down Equation 10.

$$\overrightarrow{new} = \overrightarrow{current} + 1.01 * displacement * (MLS(\overrightarrow{current}) / MLS(\overrightarrow{before})) \quad (10)$$

4.3.2 External forces

Now in order to get the fluid flowing we need to add the external forces which work on the fluid. In our case this comes down to the forces of the sources and gravity.

For the source forces this comes down to adding the velocity of the source times the time-step to the faces of the grid cell in which the source resides. For gravity it is simply adding the gravitational velocity (-9.81) times the time-step to the y-faces of a grid cell, which is marked as fluid.

4.3.3 Pressure projection

The next step in the fluid simulation is the pressure projection, this part is for making sure that after the advection and adding the external forces we will have a divergence free velocity field again. In order to do this we will need to create a system of $Ax = b$, the same method we used to compute the MLS earlier. But before that we need to make sure that all the velocities in the walls are set to zero. This comes down to looping over all solid grid cells and setting the velocity of each face to zero.

Now that we have set the wall velocities, we can start making the A-matrix and the b vector, which we call rhs (right-hand side)) from now on. For the A-matrix each row represents a grid cell, but the row will only have values in it if the grid cell is marked as fluid. The same goes for the rhs, which will be as long as the total amount of grid cells.

The A-matrix gives us the influence our neighbours have on the current cell and in the rhs we put the flow through a grid cell. These get computed as follows:

The A-matrix is a large matrix which in both dimensions is the size of the amount of grid cells. Now in order to fill in the values for one grid cell, we check each of its direct neighbours. These values are than later mapped to one row of the A-matrix. Note that we only fill in values for rows which are marked as fluid, so we skip the grid cells which are air and solid. At max we will have 7 different values for each grid cell in a row, 6 for its neighbours and one for itself. The values for the neighbours depend on whether or not their are solid, and if they are not solid if they are air or fluid. If they are solid we leave their value at zero, if they are air or fluid we add $scale * FF(index)$ to our own value, where $scale = \Delta t/(\rho * cellSize^2)$ and FF is the fluid fraction of the face between the grid cell and the neighbour for which we are currently checking. If the neighbour is fluid we set its value in the current row to -$scale * FF(index)$.

The values for the rhs can be computed using the following equation:

$$\begin{aligned} -scale * ((uFF(i+1,j,k) * u(i+1,j,k)) - uFF(i,j,k) * u(i,j,k)) + \\ (vFF(i,j+1,k) * v(i,j+1,k) - vFF(i,j,k) * v(i,j,k)) + \\ (wFF(i,j,k+1) * w(i,j,k+1) - wFF(i,j,k) * w(i,j,k)))), \end{aligned}$$

where $scale = 1/cellSize$ and uFF stands for the fluid fraction in the x direction. This is simply $1 - uSF$ where uSF is the solid fraction in the x direction. Now that we have the A-matrix and the rhs we use the Conjugate Gradient Solver from the Eigen library to get us the correct pressures. Even though we now know what the pressure should be we still need to set the velocities accordingly.

We do this by looping over each grid cell and setting each face according to the following rules, note that for each grid cell we need to check each grid cell face to set the correct velocity:

- First, we check if either of the grid cells bordering the current grid cell face is marked as fluid. If none are marked as fluid we mark the velocity for this face as unknown and we will compute its velocity later when we extrapolate the known values.

- Second, if one is marked as fluid we check if the other one is marked as a solid, if this is the case we set the face velocity to zero.

- And last, if the other is not marked as a solid we compute the velocity of the face as follows (formula is for x-axis faces):

  $$u(i,j,k) = u(i,j,k) - scale * (p(i,j,k) - p(i-1,j,k)),$$

  where u is the velocity field in the x-axis, $scale = \Delta t/(\rho * cellSize)$ and p is the new pressure we got from the Conjugate Gradient solver.

Now that we have filled in the known velocity values we need to extrapolate these values in order to have correct values everywhere in the grid. We did this in a simple breadth first manner, where we set the unknown velocity to the known velocity. If the velocity could still not be set because it was for example surrounded by solids we left its velocity at zero.

In order to make sure that the walls can still not be penetrated by water we again set the velocities of grid cells in the wall to zero. Now that we have the correct velocities everywhere we can move the marker particles, more on this in the next Section 4.4.

4.4 Solid collisions

The last part of our method where we differ from normal Eulerian fluid simulation is how we handle collisions between marker particles and the point cloud. This happens after we have done all the calculations for the fluid itself, so after the advect and the pressure projection. Our method is as followed:

For each marker particle we calculate their velocity by interpolating from u, v and w according to their position. We then add this velocity times the time-step to its current position. Now in order to check if this is a correct location, we use Equation 10 from Section 4.3.1.

5 Experiments and Results

The following experiments were performed on an i7 870 with 16 GB DDR3 RAM and a GTX 1060 6GB. For each experiment we use the following default variable values, unless otherwise specified:

- Least amount of points needed to compute the MLS: 10
- Degree of MLS function: quadratic
- time-step for each step in the fluid simulation: 0.001s

5.1 Moving Least Squares search radius

5.1.1 Description

The purpose of this experiment is to show what impact the search radius for the Moving Least Squares algorithm has on load times and errors. Errors in this experiment are grid points which are wrongfully classified as either inside or outside and will be pointed out through the usage of images. The search radius is defined as follows: the radius of cubes around a given grid vertex, from which we use the points to compute the MLS value. The base case already searches with a radius of 1, because otherwise we won't have any grid cells from which we can take points to compute the MLS value.

Figure 5: Dragon model

We perform this experiment on the dragon point-cloud from the Standford 3d scanning repository [Standford, 2014], consisting of 99869 points. The model used to compute the point cloud can be seen in Figure 5. We have chosen this model, because it has a lot of extruding parts, such as the spikes on it's head and tail. These are relevant because they could introduces errors, when large search radii clip a small part of them.

As for the settings we gave the program a minimal grid resolution of 400 on the x-axis, which turned into a grid of 406 in the x-axis, 288 in the y-axis and 185 in the z-axis. For the MLS algorithm we set the minimum required points to 10, which is the least amount needed to compute a correct solution according to the algorithm itself.

5.1.2 Results

In Table 1 we can see the final timings for the computation of the MLS. In the search radius column we find in what range we looked for points. The search radius scales from 0 being the base case of the 8 (2x2x2) grid cells surrounding a grid point, to 4 looking in the surrounding 1000 (10x10x10) grid cells. The computation part is how long it took for the algorithm to compute all the MLS values, and the missing part of the total seconds comes from our method to set the values where we could not compute them due to no points or not enough points being in range.

In Figures6 through 10 we see 2 example slices of the MLS. Green means that the MLS thinks it is outside the object, Black means we are near an edge of and object and red means inside the object.

Search Radius	Total seconds (s)	Computation part (s)
0	6.90	2.89
1	28.95	24.97
2	79.57	75.69
3	177.80	173.98
4	334.95	331.15

Table 1: Time measurements for the Search Radius experiment

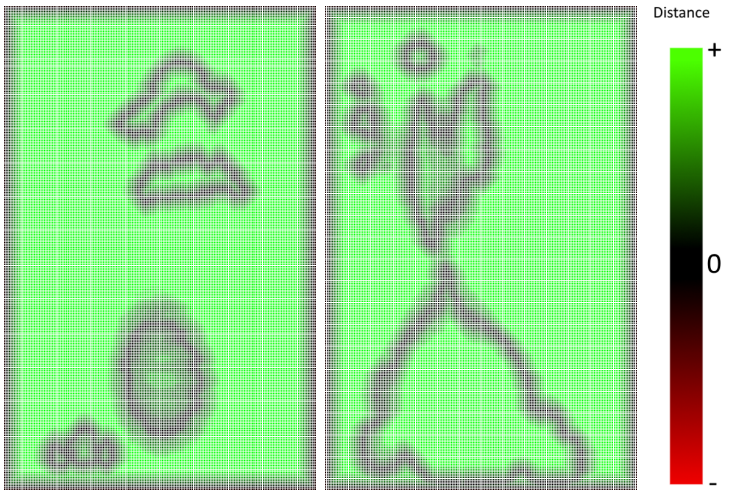


Figure 6: Slices 50 and 65 from the MLS when we set a search radius 0

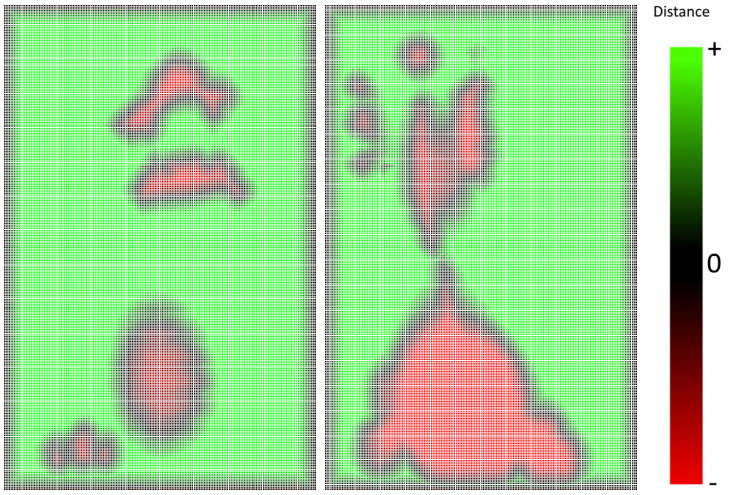


Figure 7: Slices 50 and 65 from the MLS when we set a search radius 1

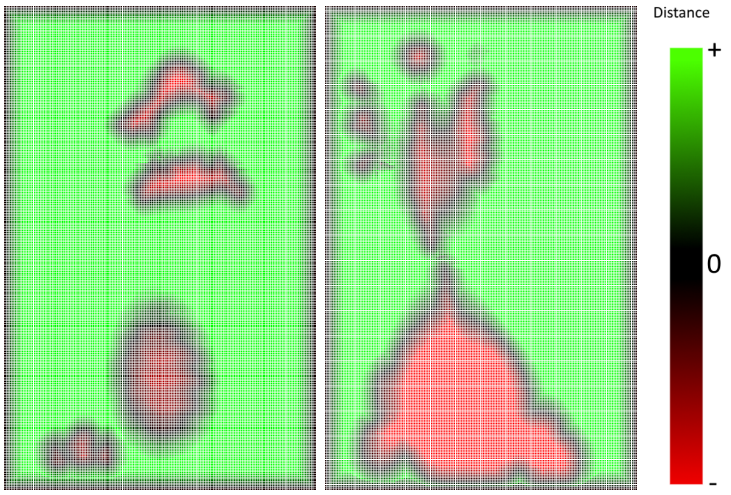


Figure 8: Slices 50 and 65 from the MLS when we set a search radius 2

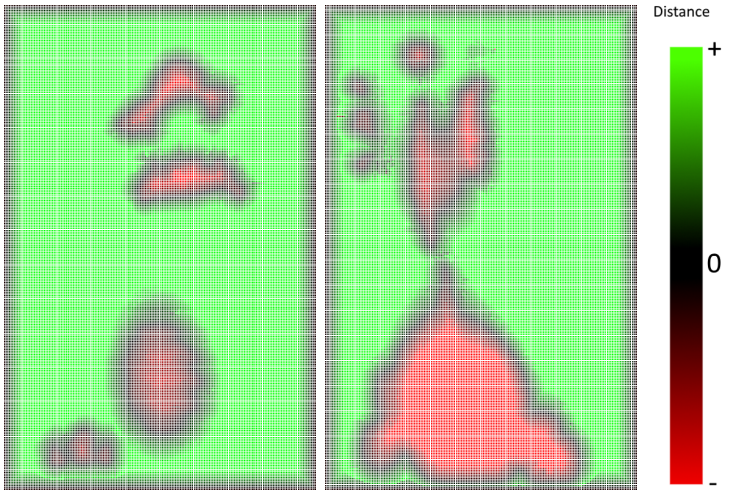


Figure 9: Slices 50 and 65 from the MLS when we set a search radius 3

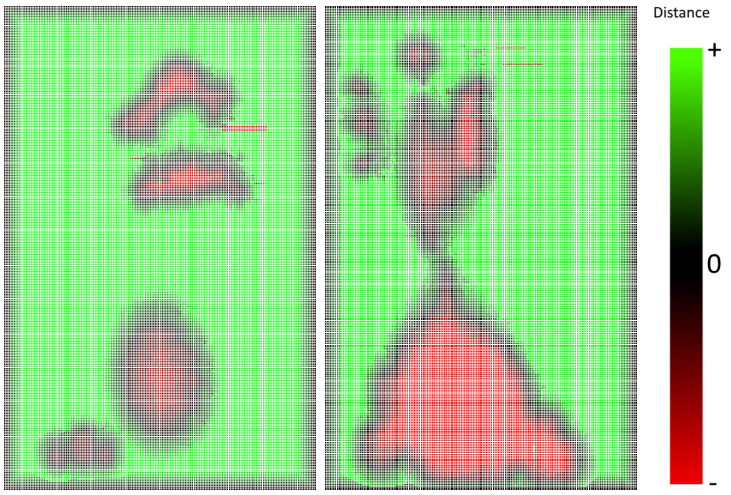


Figure 10: Slices 50 and 65 from the MLS when we set a search radius 4

5.1.3 Conclusion

Concluding from the load times of the MLS in Table 1, we can clearly see that the search radius has a big impact on it. The reason for this is because the algorithm finds more points to use in the calculation of the MLS value, which adds even more rows to the matrix which is used to compute the MLS value.

While the algorithm takes longer, the reason why we do this is to increase the band of known MLS values around the point cloud. This allows us to take bigger time steps in the fluid simulation without the risk of marker particles going through the point cloud. So taking a larger search radius has the most benefits if you have a lot of thin or small point clouds, which could get ignored by the marker particles if you also take a large time step. But in general you also want to take a larger search radius if you want be able to take larger time steps in the fluid simulation.

Note that the larger your search radius is the more chance you have to get wrong values during your MLS computation. One wrong value on its own is not that worrying, but combined with how we handle the missing values this introduces larger errors. This happens if the MLS finds just enough points on opposites of its search radius. These points could then contradict each other, which creates a wrong MLS value. These wrong values in turn add wrong behavior to the fluid simulation, because the marker particles are trying to evade solids which do not exist.

Taking all this into account we decide on using a search radius of 2 for the rest of our experiments, because this gives us the added benefit of having a larger band of known MLS values around the points, without the errors we get for a search radius of 3 and up. Your results might vary depending on the method you use to deal with unknown MLS values.

5.2 Effect of grid resolution on fluid simulation

5.2.1 Description

In this experiment we look at the impact of the grid resolution on the time it takes to compute a frame and what impact the grid resolution has on the result of the total fluid simulation.

For this scene we use a sphere (Figure 11) which consists of 1.2 million points. We have placed this sphere at the center of the bottom plane, such that no water could go underneath it. After the cell size is computed using the grid resolution and the point cloud, we double the dimensions and move the point cloud to the center of the floor of the new grid. We do this to have room for fluid to spawn unhindered.

We spawn the fluid from the left and right side, and give it a small velocity of 2.0 towards the center. As for the amount of fluid we continuously spawn fluid from the left and right wall for the first 25 frames.

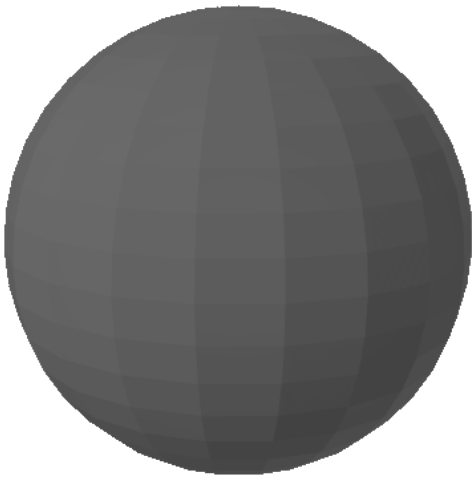


Figure 11: Sphere model

In this experiment we measure the following timings:

1) MLS computation time (cMLS)
2) Average total time used for one frame (frameT)
3) Average time the draw functions took (drawT)
4) Average total time of the simulation computation (fluidCStep)
5) Average time the advection took (advect)
6) Average total time the pressure projection took (peTotal)
7) Average time the pressure solve took (peSolve)
8) Average time the pressure application took (peApply)
9) Average time the extrapolation of the velocities took (velEP)
10) Average time it took for all particles to get moved (movePa)

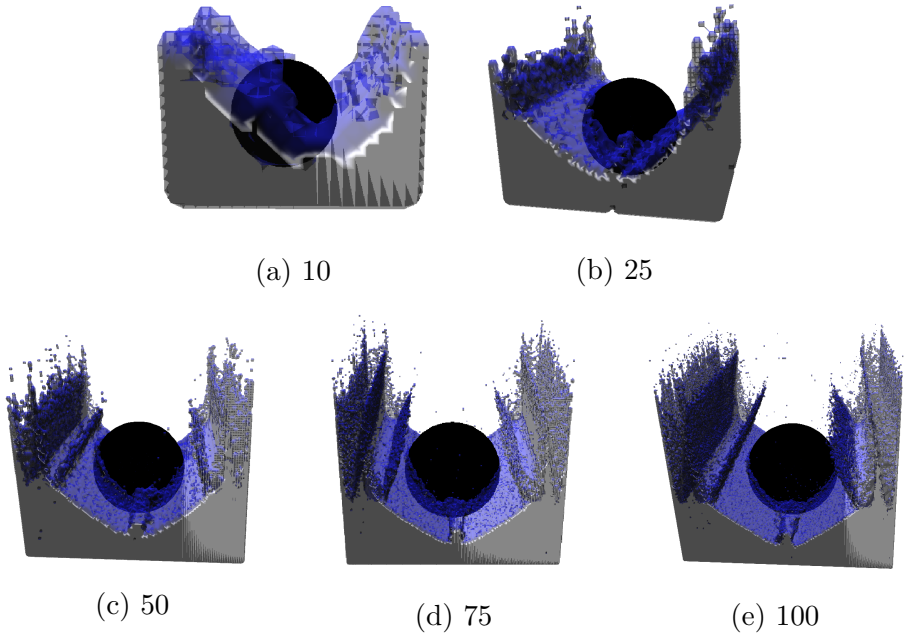
We do this for the grid resolutions (gridRes) 10, 25, 50, 75 and 100. In order to keep all the simulations at the same scale we scale the amount of grid cells which spawn fluid with the grid resolution. This makes it so that we have the same amount of visual fluid at each grid resolution.

5.2.2 Results

In Table 2 we can see the timings of our grid resolution experiment. The words in the top row of the table correspond to the list below the table. In Figure 12 we can see the visual impact of the grid resolution on the fluid.

gridRes	cMLS	frameT	drawT	fluidCStep	advect	peTotal	peSolve	peApply	velEP	movePa
10	202.143	107	16	105	3	61	52	0	7	40
25	174.270	1.521	16	1.499	56	1.230	1.140	13	73	230
50	149.838	25.428	18	25.269	364	24.106	23.854	85	145	1.345
75	96.393	104.349	20	103.813	781	101.058	100.387	137	468	4.394
100	86.504	238.038	16	238.031	1.563	231.519	229.990	315	1.078	10.518

Table 2: Grid Resolution timings in ms



(a) 10 (b) 25

(c) 50 (d) 75 (e) 100

Figure 12: The 50th frame for each tested grid resolution

5.2.3 Conclusion

When we look at the first column in Table 2, which corresponds to the computation or load time of the MLS we can see that increasing the grid resolution does the opposite with load time compared to increasing the search radius. This happens because when we increase the grid resolution we also decrease the raw range of the MLS search radius, which makes it so that the MLS uses less points per computation the higher the grid resolution is.

However the opposite can be said about the time it takes to compute one frame. As can be seen in column 2 from Table 2 none of the grid resolutions manages 30 frames per seconds, since in order for this to happen we need frame times of around 33.3 ms. The highest frame rate we got is around 9 fps in this experiment.

The largest amount of time for each frame is spend in computing the next step for the fluid simulation, as can be seen in column 4. If we then look at what takes the most time in this computation, we arrive at the pressure projection (column 6). We can then further split this timing until we finally arrive in column 7, the pressure solve itself. The main reason why this step takes so long is, because it is trying to solve a enormous matrix multiplication. This matrix is 7 times the amount of fluid cells big, because we need to know the new pressure for each fluid cell.

These huge frame times are largely due to the pressure projection needing to do a lot of processing, as seen in column 7. The second biggest bottle neck

is our method to check if each particle is still in a correct place according to the MLS. This also isn't that unusual because of the increased amount of fluid we also have an increased amount of marker particles.

So far increasing the grid resolution has mostly led to negative effects on the fluid simulation, but the main reason why one would increase the grid resolution is to create a more detailed fluid simulation, as can be seen in Figure 12. There we see that increasing the grid resolution gives us more detail in the fluid, especially in the areas around the object. While lower resolutions do look more smoother in the general areas, this is due to the marching cube algorithm we used for the visualization. The detail is more noticeable in the way the water wraps around the sphere in the higher resolution images.

5.3 Point Cloud Density

5.3.1 Description

In this experiment we look at the impact of the density of a point cloud on the accuracy of MLS and thus the whole simulation. For this experiment we use the Stanford bunny (Figure 13) from the Standford 3D scanning repository [Standford, 2014] sampled for different densities. These are computed using Cloud Compare's ([Girardeau-Montaut, 2019]) density sampling, which means given a density value, Cloud Compare will sample the object in order to give us a point cloud of x amount of points, which correspond to the specified density.

We used this function with values ranging between 10000 and 100000000. We chose these values, because the used bunny object is very small and Cloud Compare takes X amount of points from the surface of the object depending on the size of the object and the density sample value. The resulting amount of points for each sample value can be found in Table 3. For the Grid resolution we use 25, because this turned out to be the best combination from the Grid Resolution experiment (Section 5.2) for detail combined with performance.

Figure 13: Stanford Bunny model

Density value	Actual amount of points
10000	572
100000	5706
1000000	56937
10000000	571297
100000000	5712910

Table 3: Density values used and their resulting amount of points in the point cloud

For this experiment we are interested in the time the MLS takes to compute and how consistent, meaning no holes and following the shape of the object, the MLS is. We test the former using a timing and the lather through looking at slices of the MLS.

5.3.2 Results

In Table 4 and Graph 14 we can see how long the MLS took to compute. In Figure 15 we can see the solid marked grid cells for their corresponding densities. We have chosen the back side because here we could clearly see the holes which exist when the density of a point cloud is too low. Furthermore there are videos in which we can see fluid flowing towards the bunny and sometimes penetrate the point cloud due to the holes.

Amount of points	MLS computation time (ms)
572	196
5706	605
56937	5291
571297	86576
5712910	948862

Table 4: time to compute the MLS

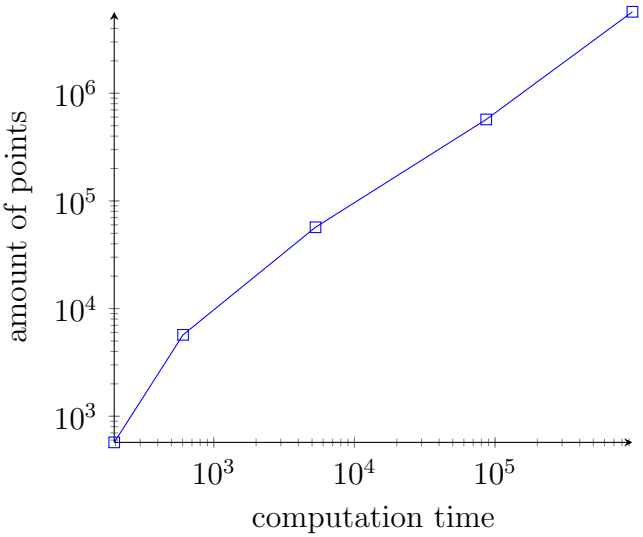
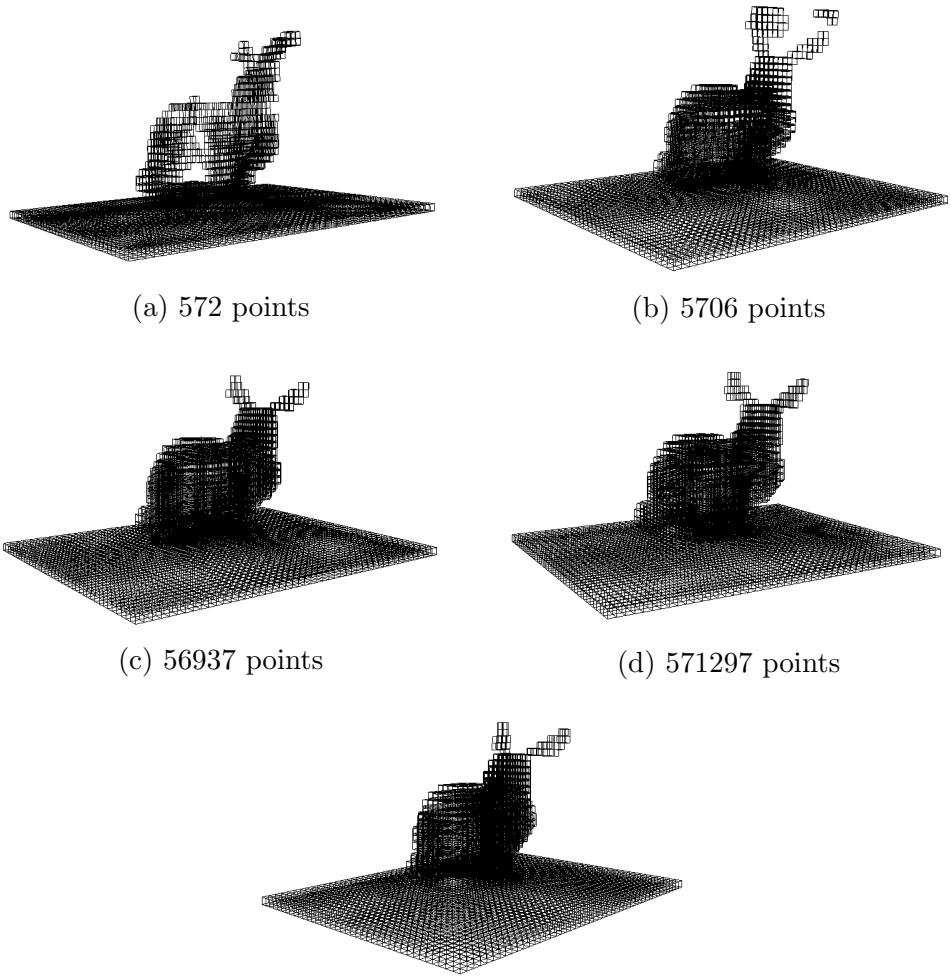


Figure 14: amount of points versus MLS computation time



(a) 572 points

(b) 5706 points

(c) 56937 points

(d) 571297 points

(e) 5712910 points

Figure 15: Solid grid cells according to the MLS for different density values

5.3.3 Conclusion

For Table 4 we see the same trend as the previous two experiments, the higher the amount of points we have in range of grid vertex the longer the MLS takes to compute. But since the previous experiments did already go into this, we will mostly focus on the resulting MLS and the behavior of the fluid in this experiment.

In Figure 15 we see the grid cells, which are fully solid according the MLS. What quickly becomes clear is that especially the lower density bunnies show signs of holes in their structure. This is best visible Figure 15a, where we can quite easily look through the model. We see the results of these holes in the videos which rotate the scenes to show the underside and thus the water which seeped through the holes.

Another thing which happened specifically in Figure 15a is that the fluid

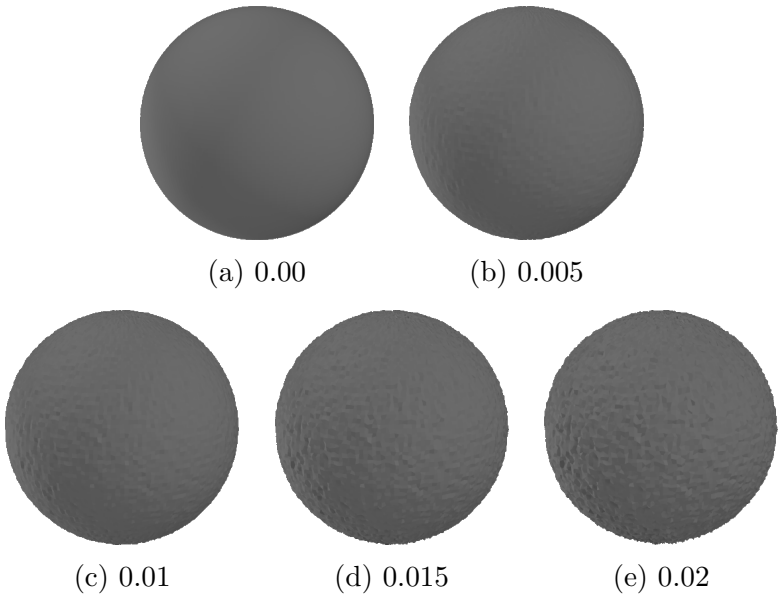
went flying out of the scene, and thus not allowing us to capture that specific simulation. For the other density values we did capture the first 50 frames of the fluid simulation. From these video's we can conclude the same as with the grid resolution experiment: the higher the density the higher the detail we can achieve. This is not only because we have more points, but also because we can increase the grid resolution without creating holes.

5.4 Random Noise

5.4.1 Description

In order to test how noise resilient the MLS is, we create a UV sphere in Blender with 100 segments and 100 rings. To this object we add noise using Blenders randomize option in the tools tab. We did this for: 0.005, 0.01, 0.015 and 0.02, these objects can be seen in Figure 16. We then used Cloud Compare to create points clouds of around 1000000 points from these object.

For this experiment we use a grid resolution of 200, because we want to see what kind of impact noise has on the MLS grid and this can only be visualized with a high grid size. The results for this experiment will consist of images of the MLS and videos showing the fluid simulation.



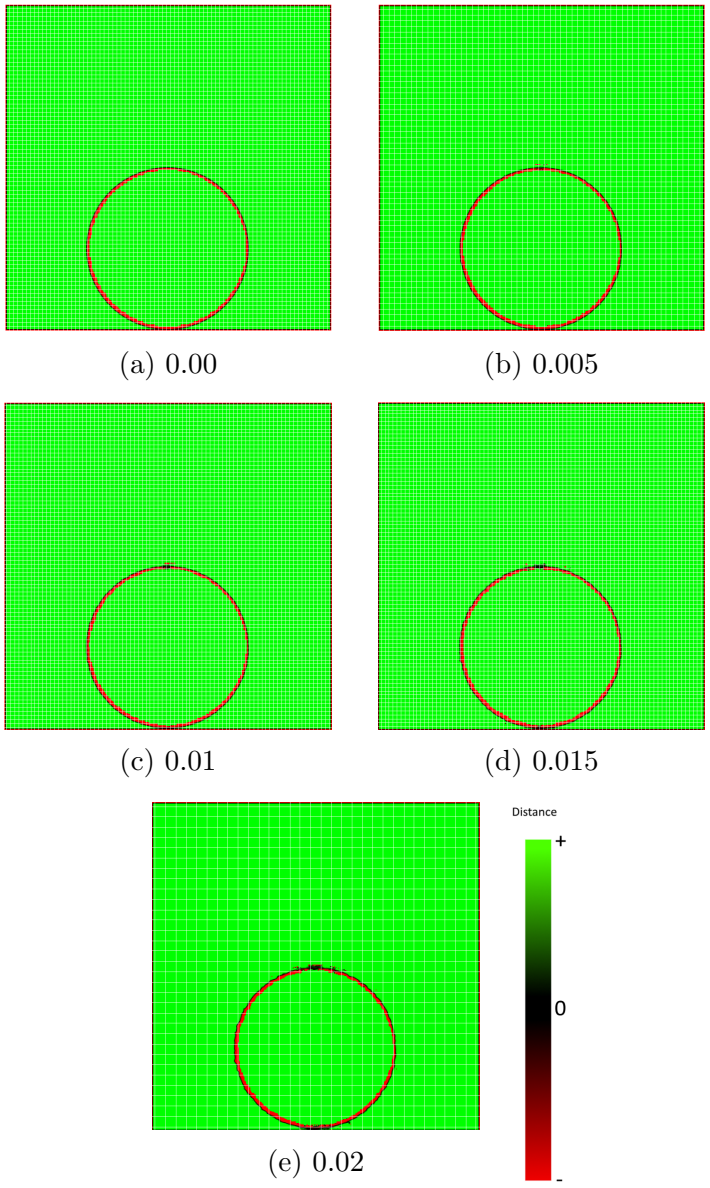
(a) 0.00 (b) 0.005

(c) 0.01 (d) 0.015 (e) 0.02

Figure 16: Models of the spheres which are used in this experiment. The values underneath the sub-figures are the values used in the randomize option of Blender

5.4.2 Results

In Figure 17 we can see the MLS values of the middle slice of the MLS grid (index 200), where green is what the grid knows is outside the grid and red is what the grid knows is inside the grid. The green inside the sphere is because the MLS values don't get extrapolated. This is to prevent large errors from forming from the small rounding errors, which can be seen in all noise experiments excluding the zero noise case.



(a) 0.00 (b) 0.005

(c) 0.01 (d) 0.015

(e) 0.02

Figure 17: The middle MLS slices for each noise value

5.4.3 Conclusion

As can be seen from Figure 16 we can see that MLS is not 100% noise proof. In Figure 17b, the one with the lowest noise value, we can see some errors forming at the top of the sphere. But despite this, we can still see that it is the section of a sphere. Another reason why noise is not a big problem in our case is because of their low quantity. Most high resolution, and thus preferred point clouds have thousands or millions of points, so a few points created by noise wont have that much of an impact on the total fluid simulation. And if the point cloud does not have a lot of points we probably don't want to use them anyway as we explained in the previous experiment.

5.5 Use Cases

5.5.1 Description

In this experiment we try out our method for different kind of models, to show that it can handle all types of point clouds, such as landscapes, cities and statues. Especially the landscapes and cities are interesting because if it produces accurate results, that means it could be used in practical cases such as floods. We also do a simple comparison scene in which we compare a cube point cloud with a cube object. This cube object is recreated from the point cloud by setting each grid cell which has a point in it to solid.

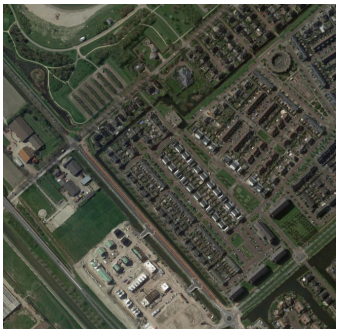
In Figure 18 we can see the aerial photo of the landscape, the statue model and the cube model.

- The landscape scene is created from data of AHN [AHN, 2018], which we cut up into pieces to be more manageable. Amount of points: 8067295.
- The Lucy statue comes from the Standford 3D scanning repository [Standford, 2014], which is the same place we took the bunny and dragon model from. Amount of points: 999454.
- The cube model is just a basic 2 by 2 by 2 cube made with blender. We recreate the normal method by setting each grid cell with a point to solid. Amount of points: 1000000.

For each scene we set the MLS search radius to two so we can safely set a time-step of 0.01s. This gives us more movement in each frame, without the particles ending up in the solids. The fluid in the scenes are configured as follows: in the landscape scene the water comes from the left, in the statue scene from above and in both cube scenes from left and right.

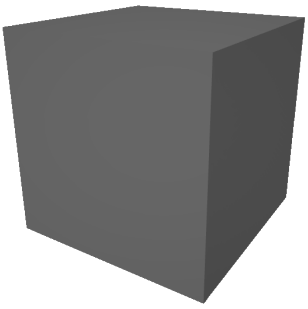
For this experiment we will report the grid resolution, which depends on the scene and how much detail we want to preserve. We also report the same

timings from Section 2 for each scene so we can show how feasible to run these scenes are.



(a) aerial photo

(b) Lucy statue



(c) cube

Figure 18: Models used as examples for the different use cases

5.5.2 Results

In Table 5 we can see the timings for each scene, the column headers correspond to the list in Figure 19. The reason why the last value in the cMLS column is missing is because we did not do any MLS computation for the reconstructed object. In Figure 20 and Figure 21 we can see the fluid flow from the landscape and statue scene. Figure 22 shows the results from the comparison between our method and the method which uses the recreated object.

scene	cMLS	frameT	drawT	fluidCStep	advect	peTotal	peSolve	peApply	velEP	movePa
Landscape	725.301	5.572	14	5.549	175	5.196	5.052	29	108	260
Statue	176.769	427	16	416	8	306	268	3	33	106
Cube	214.152	4.196	18	4.110	114	3.549	3.420	25	96	508
Cube Obj	-	5.160	17	5.078	140	4.602	4.470	28	98	431

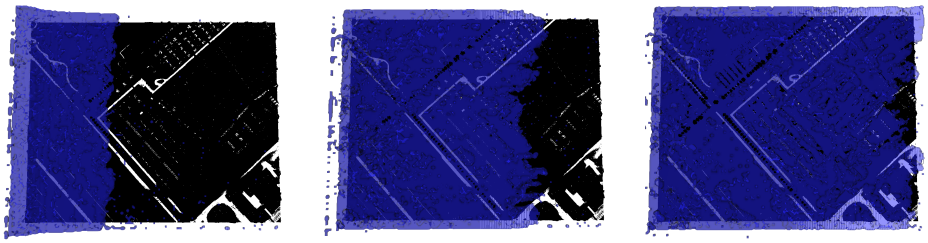
Table 5: The timings in ms for each of the experiments

- cMLS: MLS computation time
- frameT: average total time used for one frame
- drawT: average time the draw functions took
- fluidCStep: average total time of the simulation computation
- advect: average time the advection took
- peTotal: average total time the pressure projection took
- peSolve: average time the pressure solve took
- peApply: average time the pressure application took
- velEP: average time the extrapolation of the velocities took
- movePa: average time it took to move all particles

Figure 19: Table 5 column explanation

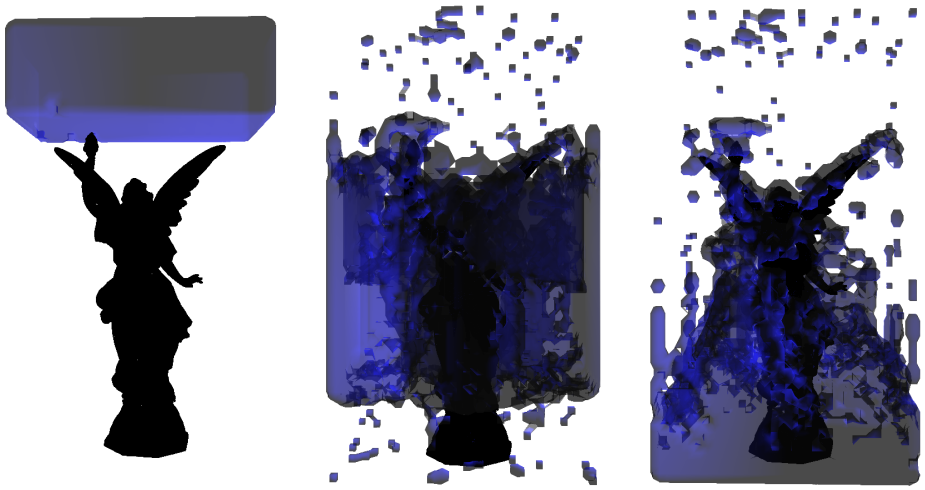
scene	x	y	z
Landscape	134	20	164
Statue	40	70	27
Cube	70	70	71
Reconstructed object	70	70	71

Table 6: Grid dimensions for each scene



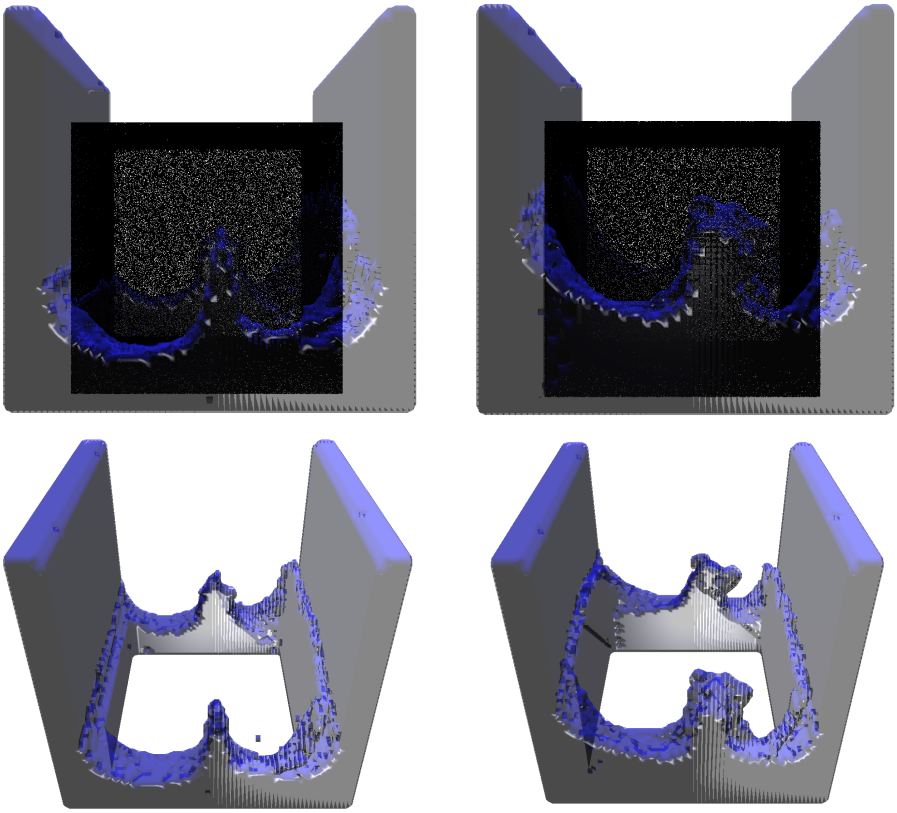
(a) frame 50 (b) frame 124 (c) frame 225

Figure 20: Fluid flow on the landscape scene



(a) frame 50 (b) frame 128 (c) frame 225

Figure 21: Fluid flow around the Lucy statue



(a) MLS (b) Recreated object

Figure 22: Comparison between the fluid flow on the MLS and on the recreated object.

5.5.3 Conclusion

When we look at the flow of water over the landscape scene in Figure 20, we see that the fluid does interact quite realistic with it. This shows by how far the fluid is in each frame, the more towards the start the fluid is the more building it met on its way. Of course it is not 100% detailed, but for predicting what will flood first it is already quite usable. As for frame rate it is not really interactive (30 fps), with a average fps of 0.18.

The Statue scene has a better fps of 2.3, which is mostly due to it's lower grid dimensions, as can be seen in Table 6. It does however also show the downside of having better fps, mainly the lack of details, which makes fluid get stuck behind some of the extruding parts of the statue.

In the comparison scene we see some small differences, but the most noticeable is that the scene with the recreated cube has a higher fluid level on the sides. This is linked to the water not being able to stick to the object on some locations, which happens because this scene only uses fully solid or fully empty cells. The MLS scene solves this by using partial cells for increased accuracy and detail.

6 Discussion

First we are going to summarize the conclusions of this thesis, after which we will explain the points on which to improve in the Future Work section.

6.1 Conclusion

The motivation behind this thesis is that there are no known methods of directly dealing with points clouds in fluid simulations. The reason why we would want to directly interact with point clouds in fluid simulations is so we skip the expensive step of recreating an object from a point cloud and the added benefit of having a simpler data structure since we are only dealing with points and not al kinds of shapes and objects. Therefore we created our own method.

Our method combines the MLS algorithm with Eulerian fluid simulations. We do this by calculating the MLS for each grid point of the fluid grid. We then use these values to determine how solid each face of a grid cell is using triangle and trapezoidal occupancy. We then use these values in the pressure projection to determine the flow of the fluid. We also used the MLS values to determine if the marker particles ended up inside the point cloud after the got moved and if the ended up inside the point cloud we used a linear interpolation between their start MLS value and end MLS value to place them at the edge of the point cloud.

While our method does become very robust if the density of the point cloud is high enough and we do have a lot of noise resistance the more points we use, we did notice during our experiments that our method has no general optimal setting for each type of scene. For example when we looked at the detail aspect of our method, it becomes very slow if we need a lot of detail, which is due to our MLS grid being the same resolution as the fluid grid and vice versa. Luckily our MLS grid does not always has to have a large resolution, because thanks to the usage of solids fractions it will preserve most of the shape of the points cloud at lower resolutions. This is mostly useful for when we want a fast fluid simulation.

So in summary, it is possible to create real time fluid simulations on point clouds, but it needs a lot of fine tuning for each scene to make it perfect and there are some mayor points on which we could improve our method.

6.2 Future Work

The first mayor improvement would be to port the algorithm to the GPU and have better multi-threading, but this holds for fluid simulation algorithms in general. Future research continuing or attempting something in the direction of this research should first and foremost look how the MLS is present in the scene and how to optimize the MLS computation.

As for the way the MLS is present in the scene we could have looked at disconnecting the MLS grid from the fluid grid. This way we could have a higher resolution for the MLS while maintaining the larger cells for the fluid simulation, so it can run in real-time. Another benefit to this method is that you can have multiple resolution for the MLS at the same time, so you can combine the precision of the MLS with faster lookup for the MLS values in areas where detail does not matter. A good example would be the open sea in front of a light house with rocks. Near the rocks and the lighthouse we want the details, but in the open ocean we don't need that precision. The final benefit from this method will be that it is trivial to add moving point clouds to the scene, since we only need to move the MLS values. This is easier to do in this method compared to our method, because the MLS is not tightly coupled to the fluid grid. The same goes for adding multiple point clouds to the scene.

References

AHN. 2018. Actueel Hoogtebestand Nederland. (2018). `http://www.ahn.nl/index.html` accessed 2018-02-15. pages 33

J Bender, C Duriez, F Jaillet, and G Zachmann. 2014. Continuous collision detection between points and signed distance fields. (2014). pages 7

Robert Bridson. 2015. *Fluid simulation for computer graphics*. CRC Press. pages 2, 8, 15

Nuttapong Chentanez and Matthias Müller. 2011. Real-time Eulerian water simulation using a restricted tall cell grid. In *ACM Transactions on Graphics (TOG)*, Vol. 30. ACM, 82. pages 4, 9

Michal Chládek and Roman Durikovic. 2010. Smoothed Particle Hydrodynamics in Flood Simulations. In *Proceedings of the Spring Conference on Computer Graphics-SCCG2010, ISSN*. Citeseer, 1335–5694. pages 3

Benjamin Eckart, Kihwan Kim, Alejandro J Troccoli, Alonzo Kelly, and Jan Kautz. 2016. Accelerated Generative Models for 3D Point Cloud Data.. In *CVPR*. 5497–5505. pages 5, 6

Jan Elseberg, Dorit Borrmann, and Andreas Nüchter. 2011. Efficient processing of large 3d point clouds. In *Information, Communication and Automation Technologies (ICAT), 2011 XXIII International Symposium on*. IEEE, 1–7. pages 6

Jan Elseberg, Dorit Borrmann, and Andreas Nüchter. 2013. One billion points in the cloud–an octree for efficient processing of 3D laser scans. *ISPRS Journal of Photogrammetry and Remote Sensing* 76 (2013), 76–88. pages 6

Mauro Figueiredo, João Oliveira, Bruno Araújo, and João Pereira. 2010. An efficient collision detection algorithm for point cloud models. In *20th International conference on Computer Graphics and Vision*, Vol. 43. 44. pages 7

Arnulph Fuhrmann, Gerrit Sobotka, and Clemens Groß. 2003. Distance fields for rapid collision detection in physically based modeling. In *Proceedings of GraphiCon 2003*. 58–65. pages 7

Jasrul Nizam Ghazali and Amirrudin Kamsin. 2008. A real time simulation and modeling of flood hazard. In *WSEAS International Conference. Proceedings. Mathematics and Computers in Science and Engineering*. WSEAS. pages 1, 3

Daniel Girardeau-Montaut. 2013–2019. Cloud Compare. https://www.danielgm.net/cc/. (2013–2019). pages 27

Abhinav Golas, Rahul Narain, Jason Sewall, Pavel Krajcevski, Pradeep Dubey, and Ming Lin. 2012. Large-scale fluid simulation using velocity-vorticity domain decomposition. *ACM Transactions on Graphics (TOG)* 31, 6 (2012), 148. pages

Prashant Goswami, Fatih Erol, Rahul Mukhi, Renato Pajarola, and Enrico Gobbetti. 2013. An efficient multi-resolution framework for high quality interactive rendering of massive point clouds using multi-way kd-trees. *The Visual Computer* 29, 1 (01 Jan 2013), 69–83. DOI:http://dx.doi.org/10.1007/s00371-012-0675-2 pages 5, 6

NM Hunter, PD Bates, S Neelz, G Pender, I Villanueva, NG Wright, D Liang, Roger Alexander Falconer, B Lin, S Waller, and others. 2008. Benchmarking 2D hydraulic models for urban flood simulations. In *Proceedings of the institution of civil engineers: water management*, Vol. 161. Thomas Telford (ICE publishing), 13–30. pages 4

Geoffrey Irving, Eran Guendelman, Frank Losasso, and Ronald Fedkiw. 2006. Efficient simulation of large bodies of water by coupling two and three dimensional techniques. In *ACM Transactions on Graphics (TOG)*, Vol. 25. ACM, 805–811. pages 4, 9, 10

Jan Klein and Gabriel Zachmann. 2004. Point cloud collision detection. In *Computer Graphics Forum*, Vol. 23. Wiley Online Library, 567–576. pages 5, 6, 7

Seiichi Koshizuka and Yoshiaki Oka. 1996. Moving-particle semi-implicit method for fragmentation of incompressible fluid. *Nuclear science and engineering* 123, 3 (1996), 421–434. pages 3

Jorge Leandro, Albert S Chen, Slobodan Djordjević, and Dragan A Savić. 2009. Comparison of 1D/1D and 1D/2D coupled (sewer/surface) hydraulic models for urban flood simulation. *Journal of hydraulic engineering* 135, 6 (2009), 495–504. pages 4

KH Lee, H Woo, and T Suk. 2001. Data reduction methods for reverse engineering. *The International Journal of Advanced Manufacturing Technology* 17, 10 (2001), 735–743. pages 5

Joe J Monaghan. 1992. Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics* 30, 1 (1992), 543–574. pages 3

Bastian Oehler, Joerg Stueckler, Jochen Welle, Dirk Schulz, and Sven Behnke. 2011. Efficient multi-resolution plane segmentation of 3D point clouds. In *International Conference on Intelligent Robotics and Applications*. Springer, 145–156. pages 6

Jia Pan, Sachin Chitta, and Dinesh Manocha. 2017. Probabilistic collision detection between noisy point clouds using robust classification. In *Robotics Research*. Springer, 77–94. pages

Johannes Schauer and Andreas Nüchter. 2014. Efficient point cloud collision detection and analysis in a tunnel environment using kinematic laser scanning and KD tree search. *The International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences* 40, 3 (2014), 289. pages 7

Ruwen Schnabel and Reinhard Klein. 2006. Octree-based Point-Cloud Compression. *Spbg* 6 (2006), 111–120. pages 6

SS Shahapure, TI Eldho, and EP Rao. 2010. Coastal urban flood simulation using FEM, GIS and remote sensing. *Water resources management* 24, 13 (2010), 3615–3640. pages 4

Chen Shen, James F. O'Brien, and Jonathan R. Shewchuk. 2004. Interpolating and Approximating Implicit Surfaces from Polygon Soup. *ACM Trans. Graph.* 23, 3 (Aug. 2004), 896–904. `DOI:http://dx.doi.org/10.1145/1015706.1015816` pages 12

Standford. 2014. The Stanford 3D Scanning Repository. (2014). `http://graphics.stanford.edu/data/3Dscanrep/` accessed 2018-11-23. pages 1, 20, 27, 33