CES Practical Course in Computer Graphics

Prof. Dr. Jan Bender, MSc. José Antonio

Contact: {bender, fernandez}@cs.rwth-aachen.de

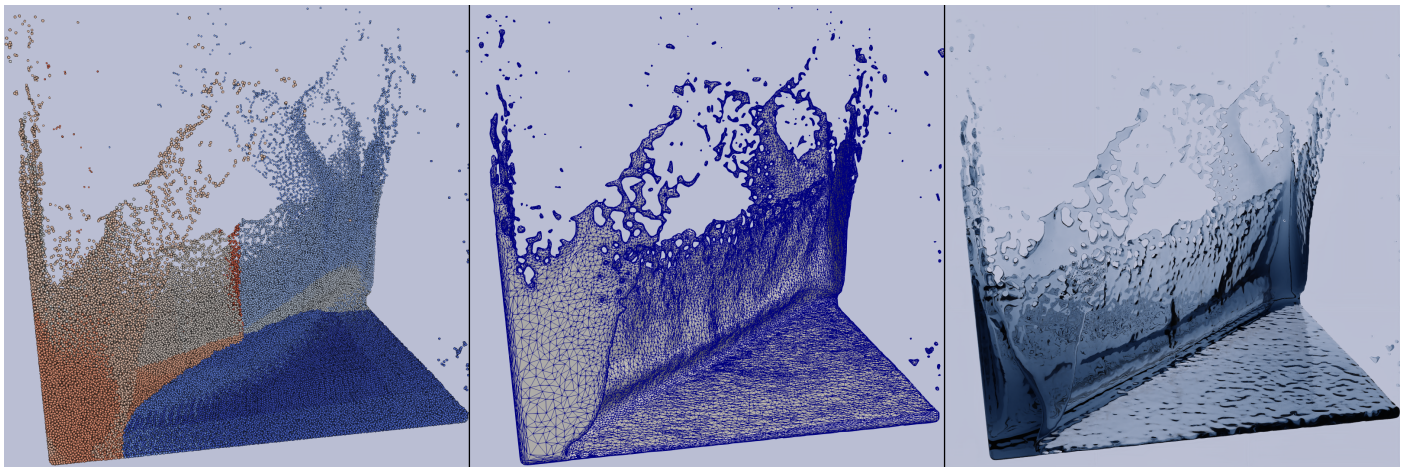

Figure 1: Transformation process from particle data to reconstructed surface mesh and ray-traced result.

1 Introduction

The need for realistic animations in computer generated movies has increased tremendously over the past decade. To capture the complex motion of animated materials, artists rely strongly on physically based simulation methods. This is especially true for fluids, since recreating their motion by hand is particularly challenging.

In computer graphics it is common to use particle methods to simulate fluids. Smoothed Particle Hydrodynamics (SPH) is one of the most popular frameworks, where particles are used to represent small amounts of fluid. Mechanical interactions between particles with themselves and with the environment are resolved using a particle discretization of the Navier-Stokes equation.

Despite particles being a convenient discretization for fluids, common rendering tools require a representation of the fluid surface for proper visualization. Surface reconstruction methods, such as Marching Cubes [LC87], are typically employed to create a surface mesh from the particle data. Since the generated surface mesh is only used for rendering and does not affect the simulation, surface reconstruction can be done in a post processing step.

In this practical course the students will implement a tool to reconstruct the fluid surface from fluid particle simulations. This tool will consist of a front end application with a graphical user interface and a different back end routines to handle the surface generation. The front end will allow the selection of files, the settings for the surface reconstruction and a preview of the particles and the reconstructed surface mesh. The back end routines will handle loading of the particle data, the surface extraction process using the popular Marching Cubes algorithm and exporting the resulting surface.

2 Problem description

The problem of surface reconstruction from fluid particle data can be decomposed in two main steps: Generation of a level set scalar field where the fluid surface is defined by an iso-value, and a surface reconstruction step to generate a triangular mesh on this level set iso-value. For the level set we will employ an SPH interpolated fluid density field that accounts for the spatial distribution of the fluid particles. This scalar field is sampled in a regular grid and the Marching Cubes algorithm is used to reconstruct the fluid surface. Lastly, several post-processing filters can be applied to the resulting triangular mesh to enhance its visual fidelity and to reduce the required memory to store it.

2.1 SPH Interpolation

Fluid is a continuous medium, however SPH uses a finite set of particle to represent it. Then, how is it possible to represent continuous and smooth properties of the fluid such as density, pressure or velocity? The answer is by using compact support weighted interpolations of discrete values. Think for instance in a randomly distributed cloud of points $\{\mathbf{x}_i\}$ inside a three dimensional unit cube. Let's assume that there is a scalar field $f(x)$ defined inside the cube and that each point has been assigned the exact value of f corresponding to their location $(f(\mathbf{x}_i))$. Then, we can define an interpolation of the scalar field at any point inside the unit cube by taking into account a small number of points very close to the query point and performing an weighted average such as:

$$f(\mathbf{x}) \approx \frac{1}{\sum_{j\in\mathcal{N}(\mathbf{x})} w_j} \sum_{j\in\mathcal{N}(\mathbf{x})} f(\mathbf{x}_j)\cdot w_j, \tag{1}$$

where $\mathcal{N}(\mathbf{x})$ is the set of neighbor points with known f close to the interpolation point $\mathbf{x}$ and w is the weight assigned to each neighbor point. In SPH, the neighborhood of a query point is typically defined by a maximum influence radius (*compact support*) and the weights are computed using a *kernel function* which has a *gaussian-like* shape that assigns higher influence to points closer to the query point (see Figure 2). Points are called *particles* in SPH, and each of them carry a certain amount of fluid mass and fluid properties. Additionally, in the context of SPH the value of w corresponding to a particle is proportional to its volume, which can be computed using the particle's mass and the fluid density.

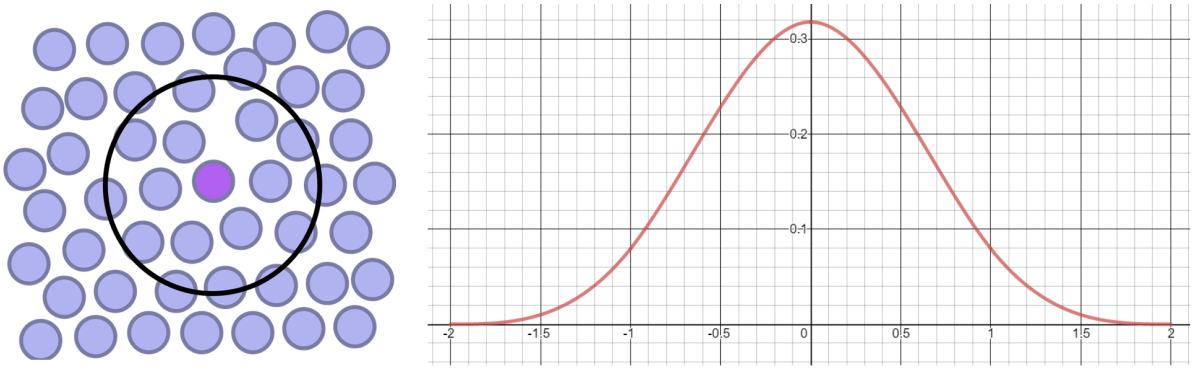

Figure 2: Illustration of SPH interpolation set up. On the left, a central point (purple) where a function must be interpolated, a set of points with values (blue) and the compact support of the interpolation (black circle). On the right, the commonly used cubic spline kernel function.

For the purpose of this course, we will make the assumption that the compact support radius is fixed. Then, the weighted average will always be performed accounting for neighboring particles inside a fixed sized sphere which center is the query point. Using this, it is possible to construct a pre-normalized kernel function W such that:

$$f(\mathbf{x}) \approx \sum_{j\in\mathcal{N}(\mathbf{x})} \frac{m_j}{\rho_j} f(\mathbf{x}_j)\, W(\mathbf{x}-\mathbf{x}_j; h), \tag{2}$$

where $\mathcal{N}(\mathbf{x}_i)$ is the set of particle neighbors within the compact support of $\mathbf{x}$, h is the smoothing length (which is related to the compact support) and m_j and ρ_j are the mass and density of particle j. The kernel function W is defined as:

$$W(\mathbf{x}-\mathbf{x}'; h) = W\left(\frac{\|\mathbf{x}-\mathbf{x}'\|}{h}\right) = W(q) = \frac{1}{h^3} s(q) \tag{3}$$

Where the cubic spline kernel function $s(q)$ is

$$s(q) = \frac{3}{2\pi}\begin{cases} \frac{2}{3} - q^2 + \frac{1}{2}q^3 & 0 \le q < 1 \\ \frac{1}{6}(2-q)^3 & 1 \le q < 2 \\ 0 & q \ge 2 \end{cases} \tag{4}$$

Note that the use of this pre-normalized kernel does not require a final division by the total volume of the compact support.

As well as with the scalar value of the interpolated function, it is possible to interpolate its gradient:

$$\nabla f(\mathbf{x}) \approx \sum_{j \in \mathcal{N}(\mathbf{x}_i)} \frac{m_j}{\rho_j} f(\mathbf{x}_j) \nabla W(\mathbf{x}_i - \mathbf{x}_j; h), \tag{5}$$

However, this is typically symmetrized in order to conserve momentum:

$$\nabla f(\mathbf{x}) \approx \rho_i \sum_{j \in \mathcal{N}(\mathbf{x}_i)} m_j \left(\frac{f(\mathbf{x}_i)}{\rho_i^2} + \frac{f(\mathbf{x}_j)}{\rho_j^2} \right) \nabla W(\mathbf{x}_i - \mathbf{x}_j; h), \tag{6}$$

Finally, the smoothing length h controls the extent of the interpolation. However, it does **not** change the shape of the kernel.

- High values of h increase the number of neighbors and therefore tend to increase smoothing.
- Low values of h decrease the number of neighbors and therefore tend to decrease smoothing.

In practice, it is chosen in relation to the "particle volume" or $\frac{m_i}{\rho_i}$. A good value is about 1.2 times the particle diameter, which will cover all immediate particle neighbors of a query point.

The compact support of a kernel depends on the extent of the function defining it. The compact support of the cubic spline kernel, is $2.0h$.

There are other common SPH kernel functions besides the cubic spline kernel. You can find those, together with a more extensive explanation of the SPH interpolation in [Pri10]. Additionally, in order to speed up the evaluation of the kernels, it is possible to precompute a set of values of W according to the squared distance between two points and either compute linear interpolation between two samples of just take the closest one.

2.2 Neighborhood Search

Compact support neighborhood search is a crucial step regarding runtime in SPH simulations. For each point in space where we need to perform any kind of computation related to the underlying continuous fields, we need to find all the particles that are within compact support radius. Therefore an efficient implementation of the compact support neighborhood search is imperative.

One of many approaches to this problem is to use spatial hashing neighborhood search (Figure 3) which can be outlined as

- Generate a grid with cell size equals to support radius.
- Classify all particles into the grid using a hash table to avoid storing empty cells.
- A neighbor particle can only lie in one of the 26 neighboring cells of the query cell (in 3D).

Other approaches to compute compact neighborhood searches involve KDTrees for better cache coherence or sorting the spatial hashing grid in the GPU.

2.3 Marching Cubes

Marching Cubes [LC87] is a well known algorithm to generate triangular meshes on an iso-value surface from a level set scalar field.

Let's call Φ the level set scalar function and assume we want to triangulate the iso-value $\Phi = 0$. Then, the steps to generate a triangular mesh using the Marching Cubes algorithm are (see Figure 4 as well):

1. Generate a voxel grid enclosing the implicitly defined surface. Each point and each cell in the grid must have a unique (i, j, k) integer coordinates relative to the number of points or cells in each direction. Given that the number of cells is finite, it is possible to device a mapping from the 3D integer indices to a unique single integer identifier. For the vertices in the grid, the mapping $(i, j, k) \rightarrow V$ could be defined as

$$V = i\, n_{v,y} n_{v,z} + j\, n_{v,z} + k \tag{7}$$

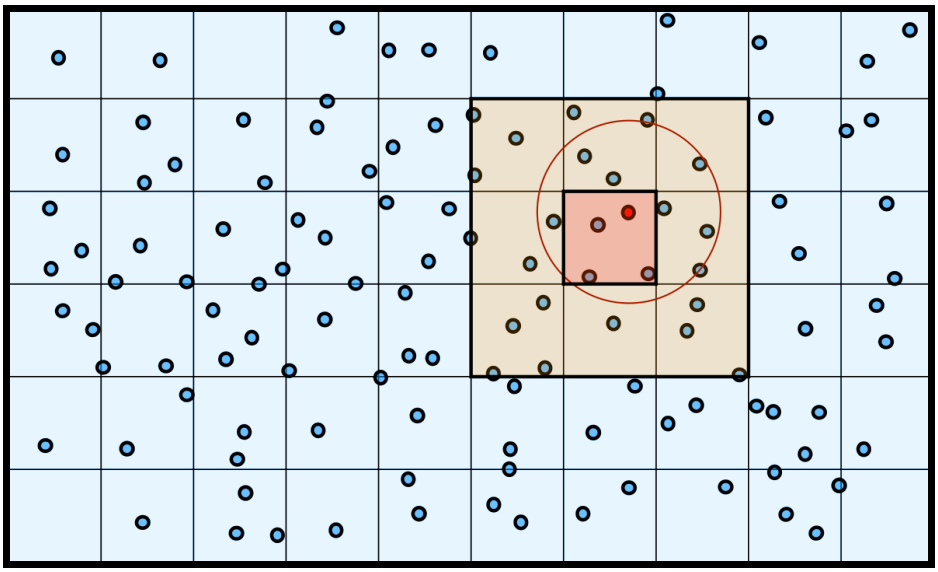

Figure 3: Example of a spatial hashing SPH query. The red point is the query point. The red circle is the compact support. The red cell contains the query point. The yellow cells are neighboring cells which must contain all particles within the compact support.

where $n_{v,y}$ and $n_{v,z}$ are the number of vertices in the y and z direction, respectively. Analogously, cells can be uniquely identified by

$$C = i\, n_{c,y} n_{c,z} + j\, n_{c,z} + k \tag{8}$$

where $n_{c,y}$ and $n_{c,z}$ are the number of cells in the y and z direction, respectively. Additionally, edges can be identified with the identifier of the vertex of origin and the direction they are aligned with:

$$E = 3V + dir \tag{9}$$

where dir is 0, 1 or 2 for x, y or z. Of course, one must handle the cases when the origin vertex is at the boundary of the grid and therefore it may not have some of the edges in the positive directions. It is suggested to use `uint64_t` as integer identifiers, instead of `int`, to avoid overflow.

2. Sample and store $\Phi_i = \Phi(\mathbf{x}_i)$ on each vertex i of the grid located at position $\mathbf{x}_i$.

3. Loop through the edges of the grid and check whether or not they intersect the target surface ($\Phi(\mathbf{x}_i) = 0$), which will be the case when the values of Φ at the edge vertices have different signs. For the edges that intersect the surface, compute the intersection point $\mathbf{x}_s$ by using linear interpolation in relation to the values at the edge vertices:

$$\mathbf{x}_s = (1.0 - \alpha)\mathbf{x}_a + \alpha \mathbf{x}_b \tag{10}$$

where

$$\alpha = \frac{\Phi(\mathbf{x}_a)}{-\Phi(\mathbf{x}_b) + \Phi(\mathbf{x}_a)} \tag{11}$$

and $\mathbf{x}_a$ and $\mathbf{x}_b$ are the edge vertices positions. Collect all the intersection points and use a map (probably a hash table) to keep track of the mapping between edge identifiers (E) and the actual intersection points found V_s.

4. Loop through the grid cells and use the marching cubes table (depicted in 5) to find the corresponding triangles of the intersection between the cell and the target surface. The different triangulations are uniquely defined by the signs of the function Φ at the cell vertices. Although there are 256 possible cases depending on 8 boolean values, they are only 15 unique cases if rotations are considered. Collect the triangles and use the mapping $E \rightarrow V_s$ such that the triangles connectivity refers to the vertices obtained in the previous step.

Naturally, one can compute the gradient of the level set scalar field at the marching cubes grid points and obtain the

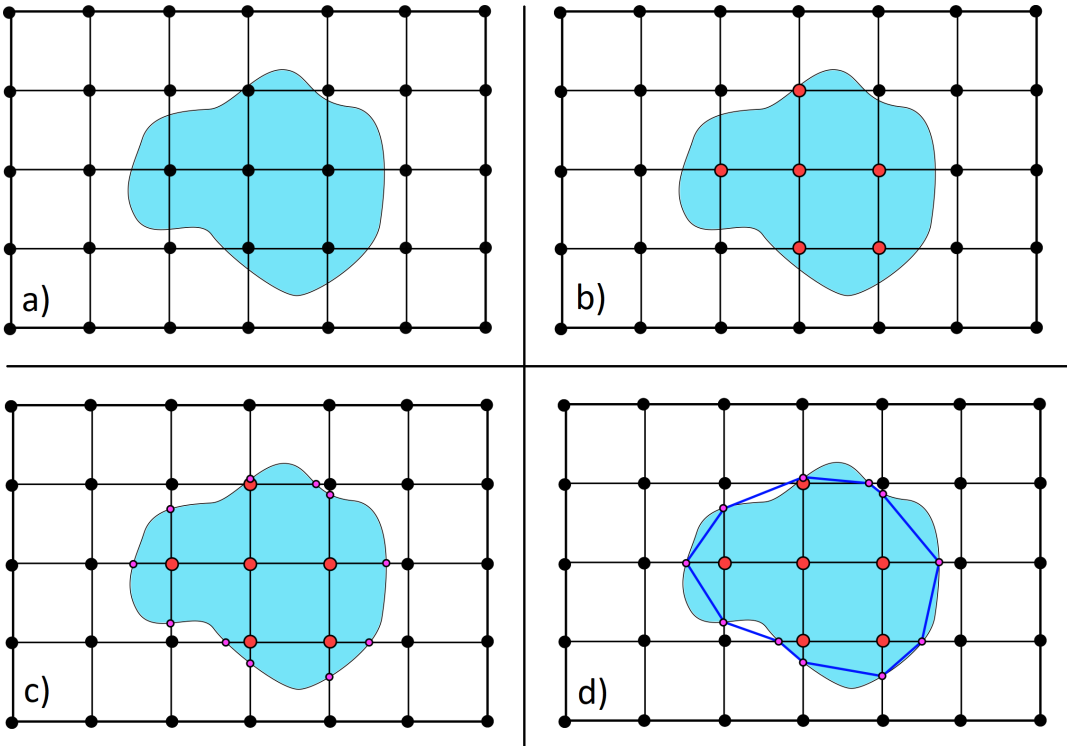

Figure 4: Marching cubes overview: a) Place a grid that encloses the target surface. b) Find the interior/exterior grid vertices. c) Find intersections between the edges and the surface. d) Connect the intersection points with surface simplices (segments in 2D, triangles in 3D) using a look up table.

gradient on the edge intersection by linear interpolation. Normalizing the resulting gradients one obtain a triangular mesh with normals on every mesh vertex which can can be used by most renderers to apply smooth shading when visualizing the results.

2.4 Fluid Surface Reconstruction

Now that we understand all the different components involved in surface reconstruction of particle fluid simulations, we proceed to put everything together. Although many different scalar fields can be used as level set for surface reconstruction, we will employ the following dimensionless expression

$$\Phi(\mathbf{x}) = -c + \sum_{j \in \mathcal{N}(\mathbf{x})} V_j \, W(\mathbf{x} - \mathbf{x}_j, h), \tag{12}$$

where c is the iso-value, $\mathcal{N}(\mathbf{x})$ is the set of neighbor particles within the compact support of $\mathbf{x}$ and V_j is the volume of each neighboring particle which can be computed as

$$V_j = \frac{1}{\rho_j} \tag{13}$$

where ρ_j is the *normalized density estimation* computed particle-wise using a SPH interpolation of the rest density of the fluid

$$\rho_i = \sum_{j \in \mathcal{N}(\mathbf{x}_i)} W(\mathbf{x}_i - \mathbf{x}_j, h), \tag{14}$$

where $\mathcal{N}(\mathbf{x}_i)$ is the set of neighbor particles within the compact support of the particle i, including particle i itself.

The scalar function Φ is continuous, differentiable and defines a surface for $\Phi = 0$, that for the right pick of the iso-value c, will tightly cover all the fluid particles. Additionally, $\Phi = -c$ far inside the fluid, $\Phi = 1 - c$ far outside the fluid and smoothly transitions between those values close to the fluid surface. A meaningful iso-value should range between 0 and

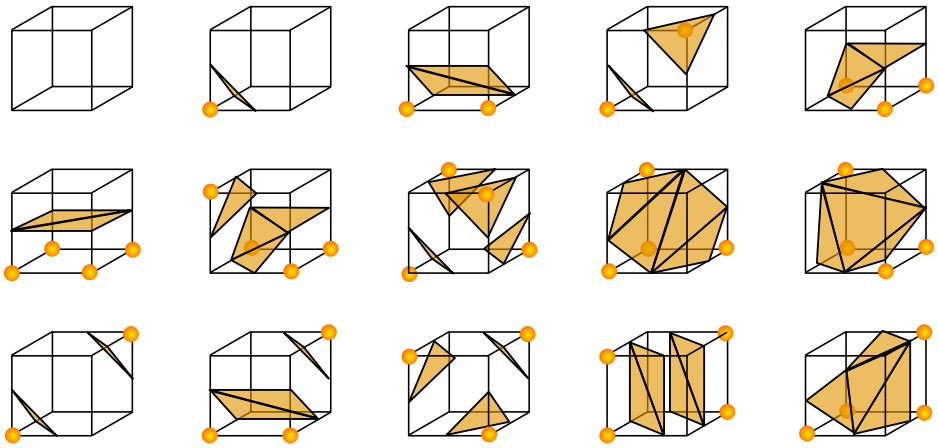

Figure 5: 15 unique reconstruction cases. Source: Wikipedia.

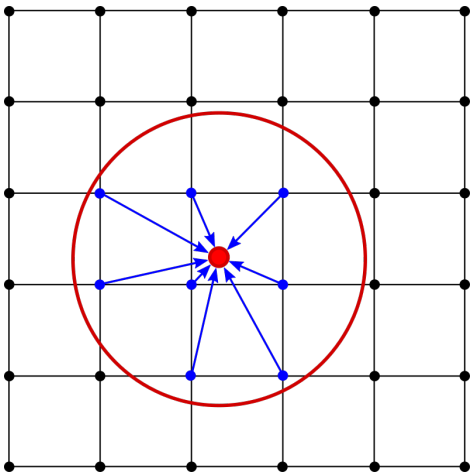

Figure 6: Fluid particle (red dot) contribution to the grid vertices (blue dots) within its compact support (red circle).

1. Finally, there is a fundamental detail to understand at this point: while both ρ_i and $\Phi(\mathbf{x})$ are defined at any point in space, we need to compute ρ_i for each fluid particle and $\Phi(\mathbf{x})$ for each Marching Cubes grid point.

A naïve strategy would to compute Φ at every grid node is to use a compact support neighborhood search to find the fluid particles neighbors of each grid point and then use Equation (12) at each grid point. However, this approach can be extremely inefficient since there is typically a huge number of grid points to account for, and most of them will have no fluid particles within their compact support. A more efficient strategy to obtain the same result is as follows:

1. Set $-c$ as initial value in all grid points.
2. For each fluid particle:
    i. Find the grid vertices that overlap with its compact support. This operation is direct given the structured distribution of the grid vertices (see Figure 6).
    ii. Use Equation (12) to add the contribution of the fluid particle to all relevant grid points.

This approach has linear complexity in the number of fluid particles and cubic with respect the the inverse of the grid cell size. No neighborhood search is needed.

With this last piece of information, we can outline the whole surface reconstruction procedure:

1. Generate a regular grid that fully contains the implicit surface.
2. Compute the normalized density estimation ρ_i for each fluid particle.
    i. Run a compact neighborhood search to find all the particle neighbors per fluid particle.
    ii. Use equation (14) to compute ρ_i.
3. Compute the level set values at every grid point using equation (12) and the described procedure to avoid a second neighborhood search.
4. Run the Marching Cubes algorithm to obtain a triangular mesh with normals from the sampled grid.

Finally, it is possible to apply some postprocessing to the reconstructed surface. Simple and very rewarding postprocessing techniques are *mesh smoothing* and *mesh decimation* or *mesh simplification*. The former makes the surface less bumpy and the second decreases the amount of triangles without sacrificing detail. Both have their own set of parameters that should be accounted for in the final application.

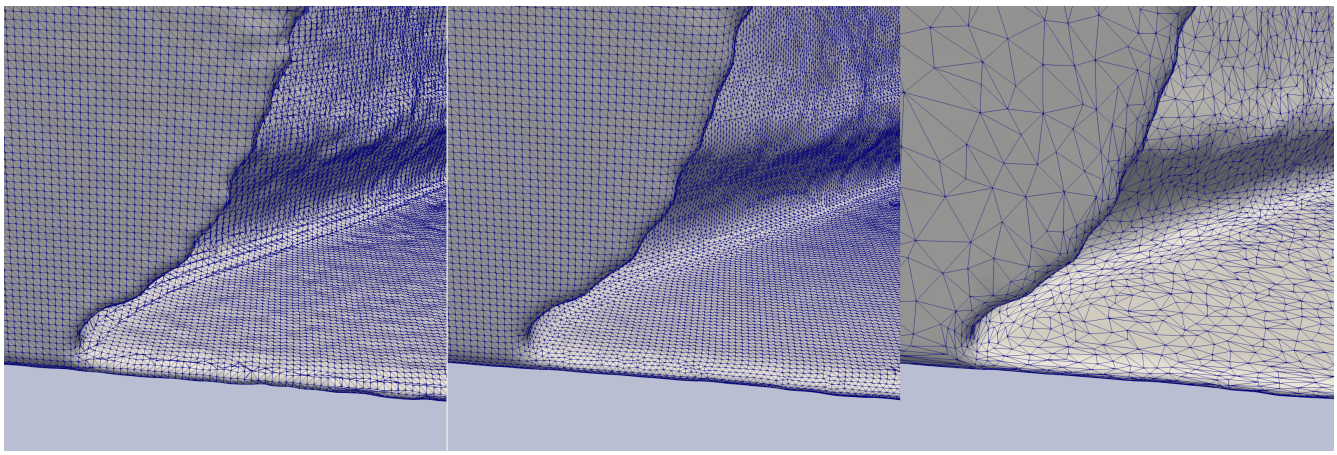

Figure 7: Effect of postprocessing: On the left, raw output from the Marching Cubes algorithm. In the middle, a smooth filter is applied. On the right, a simplification filter is applied to the smoothed mesh.

3 Task

Your task is to implement an application to compute the surface reconstruction of a particle fluid simulation. The application will have a Graphical User Interface (GUI) and will consist of two main use cases which should be handled separately.

The first case is that the should be able to specify a folder containing the particle data from a fluid simulation, which consists of a sequence of files. Then the user will indicate a particular frame number and a set of surface reconstruction parameters. The surface reconstruction explained in this document should be executed the result should be display in a 3D viewport. The user can then change the parameters, or even the frame they want to inspect, and the reconstruction must be refreshed. Every time a reconstruction is computed, it will be written in the same folder with a name that should be easy to identify.

The second use case is the possibility to run the surface reconstruction with a selected set of parameters on each and every frame in a folder. Different frames should be reconstructed in parallel. The GUI should give the option of how many threads or processes to use, but it should be defaulted to the number of available virtual cores. Finally, with cluster computing in mind, the program should be able to generate a batch script to compute the surface reconstruction of all the frame in the folder with the selected parameters.

The reason for this two parts of the application is to first allow the user to interactively generate and inspect surface reconstruction parameters and then, once they are satisfied, they can run the reconstruction for all the frames. Both uses of the application should be handled from the same GUI.

3.1 Use case 1: Interactive reconstruction of one frame

The main feature of this part of your application are the menus and the 3D renderer. While the structure and aesthetics of the GUI is up to you, the application should feature:

- Simple fields and menus to specify all the parameters.
- The folder and the number of available frames should be visible at all times.
- Proper error handling and meaningful error messages. For instance, the application should not crash if the user specifies a frame that doesn't exist. Instead, it should display an error message.
- Display of the following metrics for the current reconstruction:
  - Number of fluid particles.
  - Number of mesh vertices.
  - Number of mesh triangles.
  - Max RAM required.
  - Time required to run the surface reconstruction.
  - Progress bar or percentage when the reconstruction is running.
- Minimal 3D viewport:
  - Set camera position and orientation.
  - Render the reconstructed triangle mesh.
- Encouraged 3D viewport:
  - "Mouse and keyboard" interactive camera controls.
  - Toggle between surface or wireframe for the reconstructed triangle mesh.
  - Toggle between Gouraud/Phong and flat shading for the reconstructed triangle mesh.
  - Transparency of the reconstructed triangle mesh.
  - Show fluid particles as spheres or point gaussian (similar to Paraview).
- (Optional) The backend execution should not block the UI.

3.2 Use case 2: Reconstruction of a whole simulation

This use case is much less involved with respect to the GUI. A button should be featured in the GUI to launch the process of computing the surface reconstruction of all the frames in the folder. In addition, a tickbox next to the launch button should allow the user to create a cluster batch job instead of actually running the reconstruction. You can find a guide to the RWTH batch system in `https://doc.itc.rwth-aachen.de/display/CC/Using+the+SLURM+Batch+System`.

As already mentioned, the processing of the individual frames should be done in parallel. The minimal required implementation for this is to simply use OpenMP to run a frame per thread in a single machine. However, a better implementation would not be limited to a single machine and could take advantage of multiprocessing and use multiple machines in the cluster.

In the case of local execution (no batch job) a progress bar or percentage should be displayed in the GUI.

3.3 General remarks

Here you have a list of general remarks and requests to consider:

- The application should run in Windows and Linux.
- The application's code should be properly documented using a proper tool like Doxygen. The most relevant functionality and user interface should be explained. For instance, it is not necessary to document the function `cross_product(Vec a, Vec b)` but it would be necessary to document the function `find_particle_neighbors(...)`.

- The surface reconstruction should be reasonably fast and efficient in terms of memory. Keep in mind that a good surface reconstruction will need a Marching Cubes grid with a resolution not much larger than a particle radius to avoid losing spry particles.
- Different SPH interpolation kernel should be available to choose from. In addition, all kernels should be precomputed into discrete values for performance reasons.

Here there is a list of IO file formats that should be supported:

- Reading particle data: Partio (.bgeo), VTK (.vtk).
- Reading triangle meshes: VTK (.vtk).
- Writing triangle meshes: VTK (.vtk), PLY (.ply).

In addition, given that scientific research is a rapidly changing environment, we would like to have the possibility to expand or substitute the following components of the application:

- Compact support neighborhood search.
- Surface reconstruction algorithm (Marching Cubes).
- Level set computation.
- SPH interpolation kernels.
- IO formats.

This means that, for instance, the application should be prepared to support a different compact support neighborhood search without too much hassle.

3.4 Advices

You can use any external library at your disposal to build your application, except for the Marching Cubes algorithm which you will have to implement on your own. Here are a list of existing tools that can help you:

- You can CompactNSearch for the compact neighborhood search `https://github.com/InteractiveComputerGraphics/CompactNSearch`. Alternatively, you can take a look at nanoflann `https://github.com/jlblancoc/nanoflann`.
- Use OpenMesh for mesh smoothing and decimation `https://github.com/etlapale/OpenMesh`.
- Use SPlisHSPlasH to generate particle data from fluid simulations `https://github.com/InteractiveComputerGraphics/SPlisHSPlasH`.
- Use Partio to read BGEO files `https://github.com/wdas/partio`.
- Use vtkio to read and write VTK files (this library will be directly handled to you).
- Use happly to write PLY files `https://github.com/nmwsharp/happly`.
- You can use Paraview to interactively inspect VTK files.
- You can find many resources about OpenGL and Qt online. Here is a brief list of useful links:
  - `https://learnopengl.com/`
  - `https://doc.qt.io/qt-5/qt3d-overview.html`
  - `https://doc.qt.io/qt-5/qt3drender-qmesh.html`

References

[LC87] W.E. Lorensen and E. Cline, Harvey. Marching cubes: A high resolution 3d surface construction algorithm. *ACM Computer Graphics*, 21(4):163–169, 1987.

[Pri10] Daniel Price. Smoothed particle hydrodynamics and magnetohydrodynamics. *Journal of Computational Physics*, 231, 12 2010.