Implicit Cut-Cells Implementation for Sub-Grid Liquids Simulation

Maxime Raafat

Bachelor Thesis
October 2020

Prof. Dr. Markus Gross and Vinicius Azevedo

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

cgl
computer graphics laboratory

Abstract

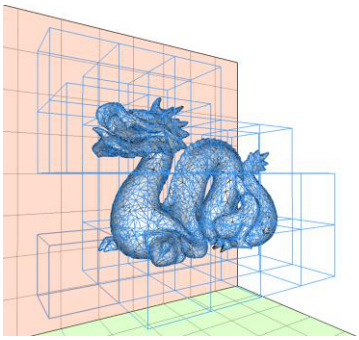
Cut-cell generation is a challenging task which, if done properly, can significantly simplify flow-object interactions for fluid animations. The method is practical for narrow-gaps or thin objects situations, where the grid resolution of the simulated flow is quite low. This work aims to extend and simplify an existing flow solver framework for future cut-cell creation by mainly looking into the rendering pipeline. We successfully implemented a particle rendering setup and described the mechanics behind the employed voxelized grid flow solver, while looking at some results generated in a smoke spreading simulation setting.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

computer graphics laboratory

Bachelor/Semester Thesis

Implicit Cut-Cells Implementation for Sub-Grid Liquids Simulation



Introduction

Accurately tracking sub-grid liquid interfaces for Eulerian fluid simulations is a challenging task. The Ghost-Fluid Method is the standard solution for dealing with grid unaligned pressure gradients that occur at liquid-air interfaces. However this method fails if the implicit interface tracking algorithm uses a higher dimensional grid, which is what is often assumed in practical liquid animation scenarios. Recently, a method for extending the Ghost Fluid Method for cut-cells was proposed to overcome this limitation. However, the generation of explicit cut-cells along with the liquids surface meshes is a computationally expensive problem, which make the extension of cut-cells to practical animation pipelines unfeasible. In this thesis we will explore a technique for generating implicit cut-cells, without the need for explicit liquids meshing. These cut-cells will form a topologically correct graph used by the pressure solving step, speeding-up the simulation of complex and adaptive liquid surfaces.

Task Description

The student must extend an existing simulation framework to support the creation of implicit cut-cells for complex geometries formed by the dynamic movement of liquids. The project will be firstly implemented in a simpler 2-D setup, which might be extended to the full 3-D simulation.

Skills

- Good C++ Skills
- Fluid simulation experience is desired
- Basic knowledge of vector calculus and algebra

Remarks

A written report and an oral presentation conclude the thesis. The thesis will be overseen by Prof. Markus Gross and supervised by Vinicius Azevedo.

Thesis Information

Thesis will be done by Raafat Maxime, from 12/06/2020 to 12/10/2020.

Acknowledgement

Throughout the writing of this thesis, I received a continuous support from my supervisor, Vinicius Azevedo, to whom I would like to show my greatest gratitude. His patience with me during many Zoom hours and his coding expertise helped me out in numerous debugging cases, even though contact in the current Covid-19 period was quite restricted.

I would also like to thank him for the last run of this thesis period, where he devoted a significant amount of his time to make the VoxelizedGridSolver2D class finally work, enabling me to present interesting results; without his precious input, the quality of this thesis would never have been achieved. Thank you.

Maxime Raafat

Contents

1

Introduction

In the last decades, the graphics industry has known several breakthroughs, especially with the incessant progress in hardware development, enabling computations to be executed faster and to be more flexible. Computational boundaries can now be pushed further on any personal computer and even on smartphones, allowing to render higher resolutions and frame rates, or simply supporting more expensive methods like fluid animations, be it in real time situations for games or in movie production. The ever increasing demand for visual quality accessible through cheap but performant hardware, or the environmental concerns of important energy consumption we are facing today is driving the research and development community to make significant software improvements, reducing the complexity of employed strategies. In this work, we focus on extending the cut-cell method, a method for low resolution flow simulations when interacting with high resolution bodies. Multiple previous research already addressed different approaches for flow simulations, such as the SPH method (Smoothed-Particle Hydrodynamics) or the PIC/FLIP methods (Particle-In-Cell and Fluid-Implicit-Particle). Although those approaches are very effective for visually realistic low cost fluid simulations, they do not handle liquid-solid interactions adaptively.

As mentioned above, our work relies on the existing cut-cell method and strive to extend it. Berger fabulously described the method in [Ber17]: the cut-cell approach computes the intersection between the Eulerian grid the flow is simulated on and the geometrical object on top of it. After having done so, it will extend the grid by clipping the new solid boundary to it, and removing the solid geometry parts that are not exposed to the flow field. The new generated grid can contain strange artefacts like floating edges or single vertices disconnected from any other vertex, or other bizarre situations. In an optimal case, the cut-cell implementation would be able to handle all those occurring events. The implementation should also be capable of handling narrow gaps and thin meshes, which is the whole point of the cut-cell method (simulating low resolution fluids against high resolution geometries, like very small objects). Simply explained, the goal of the cut-cell approach is to adaptively augment the grid resolution to the boundary

shape of an object interacting with the simulated fluid, but leaves the grid as it is away from the object boundary (as explained by Azevedo in [ABO16]).

While setting up the flow solver environment for future cut-cell usage, multiple challenges have been encountered during the period of this thesis. A tremendous amount of time has been invested in extending the rendering pipeline of the flow solver to be compatible with Magnum[1]. Although this not being the initial goal, refactoring the rendering pipeline with Magnum ended up taking the whole thesis duration. In a first step we will describe the Chimera and Magnum setup while going into detail about the rendering framework, followed by details about the challenges and implementation of the rendering pipeline. Finally, we will have a look at the voxelized grid flow solver and give the reader a few insights about the current status of our implementation.

[1] Magnum is an open-source graphics engine written in C++ and OpenGL which enables the use of multiple tools for graphics development. See their homepage here: `https://magnum.graphics`.

2

Related Work

Generating cut-cells for accurate sub-grid liquid tracking can be a challenging task. Although the concept of cut-cells might be relatively straight forward to grasp and understand (as discusses in section 1), constructing a robust implementation which fulfils the required conditions has as yet only been accomplished partially. In the following, we will discuss a few recent achievements in the quest for rigorous grid-geometry intersection for fluid flows and the requirements an ideal implementation should meet.

Usually, cut-cell implementations for flow simulations consist mainly of 3 steps

- Advection of the particles
- Generating the cut-cell mesh
- Performing a pressure projection on the mesh

(as well as additional in-between steps, like transferring particle velocities to the mesh, applying the forces to it and updating the velocities from the mesh. We will however not discuss those additional steps; further details are well described in the research paper by Azevendo et al. [ABO16] entitled *Preserving Geometry and Topology for Fluid Flows with Thin Obstacles and Narrow Gaps*.)

2.1 Geometry robustness

Ideally, generating cut-cell meshes should be a generic task, supporting multiple features and allowing any geometric input. Tao et al. [TBFL19] accomplished a remarkable implementation (titled *Mandoline*) fulfilling most of the desired geometric features, supporting split-cells, tunnels, adaptive grids or open and non-manifold meshes, thus enabling any polygonal geometry to split the grid into multiple distinct components or to cut a cell without intersecting its edges or vertices. Previous work by Zhou et al. [ZGZJ16] already enabled most of those features, but did not support as many as *Mandoline* such open meshes, neither allowed for other than triangular meshes.

Enabling generic geometric input for cut-cell implementations constitute an important asset, but remains only one half of the method. Almost more concerning is the way the solver handles the immersed boundary[1], and how particles pressures and velocities are stored in the grid.

2.2 Flow solver

As mentioned in the task description of this work, the standard solution for dealing with grid unaligned pressure gradients occurring at liquid-air interfaces is the Ghost Fluid Method (GFM) [FMOA99]. Widely used in graphics for its simplicity of implementation and robustness, the GFM enables pressure solving at liquid surfaces by placing non-existing values (called ghost values) in the air phase (thus considering the air as a liquid). Although very efficient, this method is limited by the grid resolution, that is even if a grid cell contains particles, as long as its center is outside the liquid phase, it will (wrongly) be considered as an air cell. This might lead to strange and non-desired floating artefacts for objects with small details or thin boundaries cutting through the liquid phase.

The cut-cell method solves the floating artefacts issue by storing the pressures and velocities in a clever way directly on the new grid generated at each time-step (after clipping the object-mesh to the grid cells). Colella et al. [CJ98] implemented a cut-cell method by storing the velocities in the centroid of each new partial face (the new cells generated after clipping); however, by doing so, the method yields non-symmetric complex stencils, which are defavorized for numerical operations. Azevendo et al. [ABO16] described and solved the latter issue by achieving a symmetric positive definite stencil system, allowing to simulate cut-cells in all kind of situations (narrow regions and thing gaps, small mesh details, etc.).

[1] Here we have considered the geometry properties and the flow solver as two unrelated concepts for the sake of classifying the previous works in cut-cell implementations. In reality, the geometric features one wants to enable need to be supported by the flow solver, and thus both geometry and flow properties are interlinked.

Despite having impressive properties and leading to very elegant results, the cut-cell method remains quite slow. Below (see table 2.1 and 2.2) we can observe the timings per frame for generating a cut-cell with 2 different input meshes (a bunny and a dragon), once for the method by Azevendo et al. [ABO16], and once for the *Mandoline* method [TBFL19]. Please note that those timings are the results from separate publications, i.e. the results cannot be compared directly in a one to one fashion. The grid dimension for both methods are for example not the same, neither are the used architectures (thus obviously leading to very different measurements). However, those numbers are provided for the sake of giving the reader better insights about the computational effort of the state of the art cut-cell implementations, and why optimizing for speed is of significance.

Method by Azevendo et al. [ABO16] with Grid Dimension **7x7x7**	
Loaded Mesh	**Time / Frame [seconds]**
Bunny	0.881
Dragon	1.986

Table 2.1: *Benchmarks for 3D cut-cell simulations measured with a single core Intel i7-2600 CPU at 3.4 GHz with 8GB RAM.*

Mandoline Method by Tao et al. [TBFL19] with Grid Dimension **10x10x10**	
Loaded Mesh	**Time / Frame [seconds]**
Bunny	3.225
Dragon	5.441

Table 2.2: *Benchmarks for 3D cut-cell simulations measured with a (single core) Intel Xeon E5-2630 at 2.4 Ghz with 64 GB of RAM. The Mandoline publication did not mention any parallel implementation, so single core run-times can be assumed safely.*

3

Core of the thesis

Cut-cell generations and more generally flow animations need to be integrated in a rendering framework, enabling visual output and feedback of how the flow behaves over time. This section is devoted to give the reader insights into the Chimera and Magnum rendering frameworks, the basis of our work, which will then be further detailed in the second part of this section. Finally, we will dive into the voxelized grid flow solver and look into results generated by a smoke simulation.

3.1 Chimera and Magnum rendering framework

3.1.1 Chimera structure

For the implementation of the rendering pipeline for future cut-cell usage, we were provided with the Chimera Application. Chimera is, at the time of writing, an in-development Software written in C++ by Vinicius Azevedo, designed for simulations of cut-cells. The objective of the Chimera framework is to ensure fast and simple usage of 2D and 3D cut-cells, while enabling multiple desired functionalities (see the Related Work section).

The Chimera implementation consists of multiple underlying directories, each one of them consisting of the major classes defining the complete framework. Among others, ChimeraSolvers, ChimeraAdvection or ChimeraParticles are responsible, as expected, for the flow solver and advection parts, as well as how particles are generated and distributed over the simulation grid. As we will describe in section 3.2, we mainly extended the ChimeraRendering part which takes care of, as the name hints, the rendering of the simulation. While we only extended the 2D pipeline, similar work can be performed in 3D.

In the following subsections, we will explain in details the important structure elements for 2D

rendering of the ChimeraRendering directory. Before starting, let us clarify that each Chimera subfolder (ChimeraRendering, ChimeraSolvers and so on...) are all split in 2 subdirectories, the declaration part (*include/*) and the definition part (*src/*). From this point on, we won't always specify which directory we are referring to, however it should be clear which subdirectory is meant (otherwise, this will be mentioned explicitly).

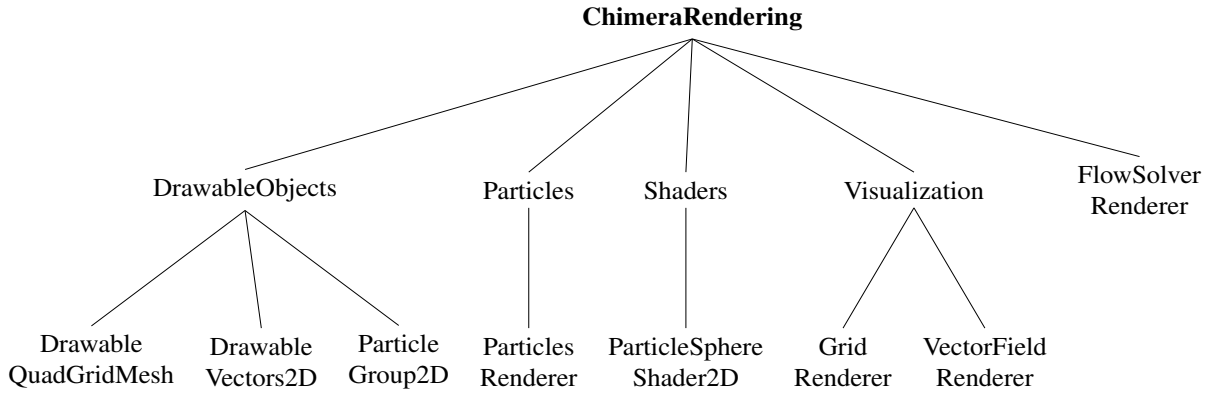


Figure 3.1: *Tree representation of the ChimeraRendering directory.*

The tree from Figure 3.1 presents the main layout of the ChimeraRendering directory, where each leaf is a class in a file, and each parent is a directory. Note that this figure is incomplete, some directories and files are missing. Nonetheless, for the purpose of simplicity, only the most relevant classes are part of this simplified representation.

3.1.2 Magnum particle rendering framework

ParticleRenderer

Extending the particle renderer in Chimera with Magnum has been the central achievement of this work. Details to the implementation and to the taken steps will be described in the second part of this section (3.2). The ParticleRenderer class takes several arguments : a ParticlesData[1] object, some rendering parameters (given in a struct format and consisting of the particle size and other parameters which might become handy) as well as scene and camera arguments. Notably in this class, one can find the class constructor as well as several functions, the *draw()* function being the most important one. Let us describe below how the class is constructed and how drawing the particles works.

The ParticleRenderer constructor is relatively straight forward and mainly consists of one instantiation, namely creating the drawable particles object (*m_pDrawableParticles*) from the ParticleGroup2D class (see details about the latter class in the ParticleGroup2D subsection). *m_pDrawableParticles* takes the particles positions as argument (those are stacked into a vector

[1]The ParticlesData class is located in ChimeraParticles. This directory takes care of sampling particles in either a uniform or a poisson distributed fashion. We will not go too deep into details about ChimeraParticles, since it is not part of the Magnum framework and wasn't an essential part of this work.

in the constructor as well), the particles size, some scene and camera arguments and an object from the DrawableGroup2D class (scene, camera and drawable group objects have only been added later, see 3.2 for explanations). Finally, we (optionally) set the color of the particles, although this is not necessary if the particle shader is set to a color gradient and not to a uniform color (more details in the ParticleShader2D subsection).

Regarding the *draw()* function, it only required the camera to make a call to the *draw()* function implemented in the ParticleGroup2D class with

```
m_pCamera->draw(*m_pDrawableGroup);
```

in order to display all objects added to the drawable group *m_pDrawableGroup* (i.e. in this case, only the particles). The drawing of the particles was not achieved in this way in a first place, since we didn't need the ParticleGroup2D class to be a child of the Drawable2D class. Details to this choice are, as already mentioned, explained in the second part.

ParticleGroup2D and ParticleShader2D

The particles created in the ParticleRenderer constructor are objects from the ParticleGroup2D class with a shader implemented in the ParticleSphereShader2D class. Since the core of the rendering happens there, we want to give the reader a detailed overview of those classes.

As discussed above, the ParticleGroup2D class takes several arguments : a vector of positions, a floating point number (for the particle radius), a Magnum::Scene2D scene object, a Magnum::Object2D object-transformation object and a Magnum::SceneGraph::DrawableGroup2D drawable group object. The two first arguments are necessary for the rendering, whereas the remaining objects take care of making the class a public child of the Drawable2D class. Most functions in this class are responsible for enabling rendering parameters adjustments, such as tweaking the shader's color, the size of the particles or the color mode (whether the particles have a uniform color or follow a color gradient). We again specifically pay attention to the *draw()* function, where all the magic happens.

Since ParticleGroup2D is a child of the Drawable2D class, our *draw()* function needs to override its parent function. We start by allocating our particles positions to an OpenGL Mesh object *_meshParticles* (more precisely a GL::MeshPrimitive::Points object) initialized in the constructor by adding our vector of positions to an OpenGL Buffer *_bufferParticles*. Once this is done, the ParticleSphereShader2D object (*_particleShader*) instantiated in the constructor is created (by calling numerous functions from the shader class) and we can finally apply the *draw()* function from the AbstractShaderProgram class (an OpenGL class we will not describe here), which draws our particles with the desired shader properties. Note that one important function from the ParticleSphereShader2D class consists in setting the camera projection matrix (*setViewProjectionMatrix()*) and its call (see below) is responsible for aligning the camera view (*camera.projectionMatrix()*) to the particles no matter how the camera is moved (*transformation*).

```
(*_particleShader).setViewProjectionMatrix(
  ↪  camera.projectionMatrix() * transformation);
```

In contrast to the ParticleGroup2D, the ParticleSphereShader2D is very straight forward to understand. In its constructor, it only reads in a vertex and fragment file (of OpenGL types GL::Shader::Type::Vertex and GL::Shader::Type::Fragment) and verifies for their correctness by checking whether they compile. In a second and last part, the class contains several functions which are responsible for performing parameter adjustments such as color or particle size tweaking. One can for example decide to opt for a uniform color for the particles by setting the integer *colorMode = 0* (in this case, the color can be changed with the *setColor()* function) or select a gradient color transition (*RampColorById*) by setting *colorMode = 1*.

3.2 Extending the rendering pipeline : challenges and implementation

Since the work has completely been performed on MacOS, a few in-between steps had first to be taken in order to assure compatibility of the Chimera application with the Clang compiler (since initially written for Visual Studio). Those necessary steps comprised the installation of required libraries (*corrade*, *magnum* and *magnum-integration*), adapting the main CMakeLists file by adding some Clang specific compiling flags, and fixing some compiler related syntax issues dispatched among the whole application. Once set up, an important part of the research was to get the particle renderer to work properly, which involved a lot of errors and unexpected hurdles. Below are described the encountered obstacles and how those have been tackled and overcome.

3.2.1 Temporary particles in the GridRenderer class

The central part of the rendering of our simulation happens in the FlowSolverRenderer class (located in ChimeraRendering), which again makes use of several underlying classes; the GridRenderer class being among those. Since the ParticleRenderer class had a few issues at first, the first particles have temporarily been implemented and rendered inside the GridRenderer class constructor using Magnum instantiations.

We started by initialising ParticleGroup2D objects and assigning them a fix position *particle_positions* and a constant radius *particleSize* (defined in the constructor as well)

```
m_pDrawableParticles =
  ↪ make_unique<Magnum::Examples::ParticleGroup2D>(
  ↪ particle_positions, particleSize);
```

and by drawing them in the following way[2] (inside the GridRenderer *draw()* function)

```
m_pDrawableParticles->draw(m_pCamera,
  ↪ Magnum::GL::defaultFramebuffer.viewport().size().y(),
  ↪ DomainDisplaySize.y());
```

(where we took care of setting the *DomainDisplaySize* object properly).

The particles' position were first chosen (arbitrarily) to be each grid-cell's vertex position (i.e. one particle on each grid-cell corner). However, assigning the particle object those positions led to crashing the simulation (we assume this is an overflow issue : if we set one particle on each of the 4 vertices of a grid-cell, we obviously get several particles per vertex, since every interior vertex[3] shares at least 4 grid-cells - see Figure 3.2). To overcome this issue, we decided to abandon the from-scratch implementation, and initialized the particles in the same way as

[2] You may notice that the drawable particles object is neither instantiated, nor called (with *draw()*), in the same way as described in 3.1.2. The initial Magnum ParticleGroup2D instantiation works as described in this part, but we implemented a few changed we will explain later in 3.2.2.

[3] An interior vertex is a vertex which is not on the grid-edge (or grid-boundary), i.e. a vertex with less than 4 neighbour vertices.

in the Magnum 2D Fluid Simulation example (see `https://doc.magnum.graphics/magnum/examples-fluidsimulation2d.html`) by using the APIC (Affine Particle-In-Cell) fluid solver from the example.

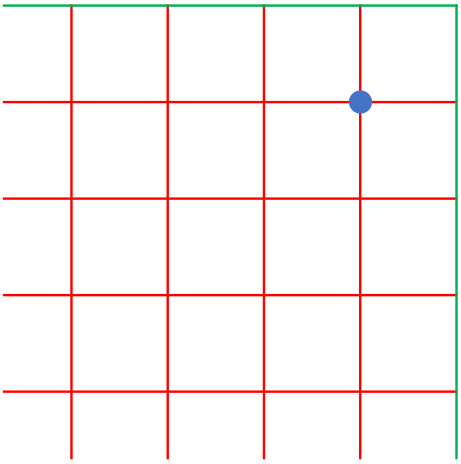


Figure 3.2: *The interior part of the grid-mesh is represented by the red edges, whereas the grid-boundary is represented in green. The blue point corresponds to an interior vertex, and has 4 neighbours in 2D.*

After having introduced the temporary APIC solver, we finally managed to render the particles and get them to behave in a meaningful way. Despite normally behaving particles, we encountered some issues with their non-changing and very little size[4], and ran into a lot of scene and camera view issues. Since for the loaded grid (see the RealtimeSimulation class in ChimeraApplications), the grid centroid does not coincide with the scene origin, a few camera projections and transformations had to be performed. This was no issue inside of the FlowSolverRenderer class, however for underlying initialized class objects (like the *m_pGridRenderer* object from the GridRenderer, or later the *m_pParticlesRenderer* from the ParticleRenderer), those transformations seemed to vanish (for no explainable reason) when declared. In other words, for

```
m_pGridRenderer = make_unique<GridRenderer<GridMeshType>>(
↪  m_pFlowSolver->getSimulationGrid(), m_pScene, m_pCamera);
```

the *m_pCamera* object was set back to its initial projection matrix (not centred around the origin), even though the camera projection had been performed previously. This strange behaviour led to a few difficulties, but were later resolved by re-centring the particles inside of the respective class (ParticleRenderer). Although this bypass trick was not the most sophisticated solution, it still enabled to observe satisfying results[5] (see Figure 3.3).

After having resolved most rendering issues, we could finally deviate from the APIC solver (and remove it) in order to construct our particles with the desired ParticlesSampler class (in ChimeraParticles), which makes use of the ParticlesData class (see Figure 3.4). Here, we could

[4]This issue which was later resolved by including a missing Magnum header as a preprocessor directive.
[5]A cleaner solution was later implemented and will be explained soon.

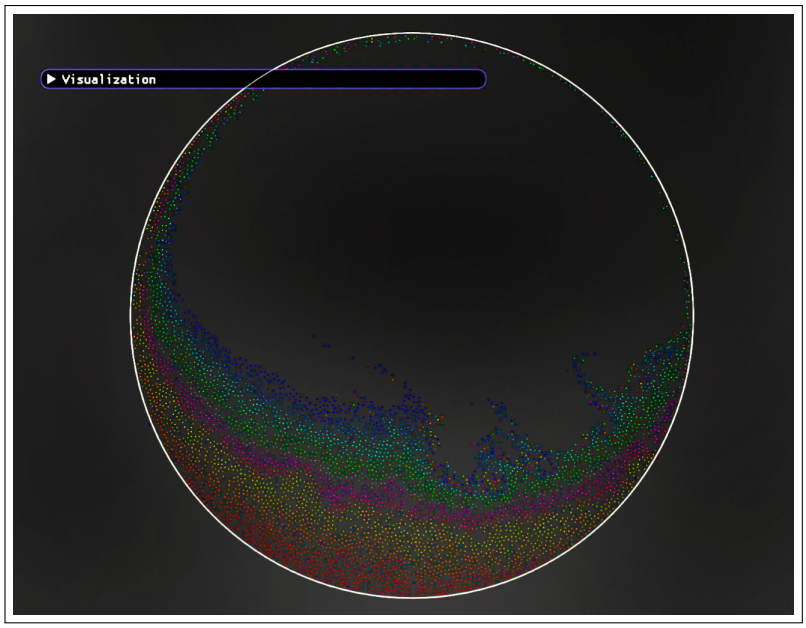


Figure 3.3: *An early screenshot from the Chimera Application with the APIC solver (particle radius still unchanged and erroneous).*

generate the particles in a relatively straight forward manner with the *createSampledParticlesPoisson()* function (or other equivalent function creating particles distributed in a uniform fashion), and finally move the whole rendering region to its dedicated class (ParticleRenderer).

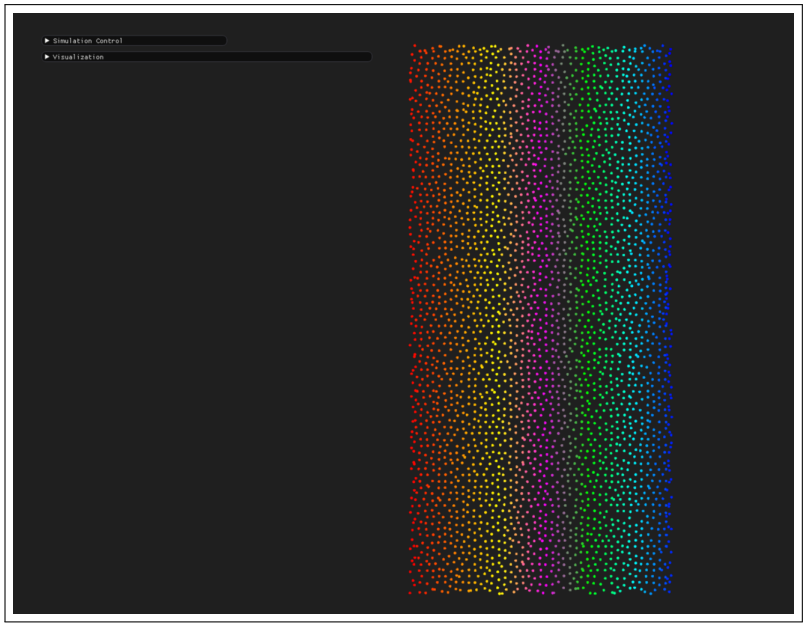


Figure 3.4: *Screenshot from the Chimera Application with particles generated from the ParticlesSampler class (with radius = 5).*

3.2.2 Particle rendering issues

As discussed in the previous subsection, one encountered issue was the particles not being centred to the scene centroid. When switching from the Magnum 2D Fluid Simulation with the APIC solver to the ParticleSampler, we realised that our particle data was generated from the fluid solver simulation grid (which was not centred). Instead of re-centring the particles in the ParticlRenderer, we wrote a *translateParticles()* function inside the ParticlesData class, which allowed to move the particles wherever desired, thus presenting a handy solution for translating the particles when the flow solver grid is not positioned correctly in a first place.

When running the simulation and using a computer mouse, one could observe an issue with the dragging of the scene objects performed with the middle mouse button. A user who would try to move the particles around would observe a strange event : the particles would be zoomed out and get infinitesimally small (since moving far away from the camera), but the grid and vector field would move as expected. We figured out that this issue was due to all visible objects (grid, vector field and scalar field) being child objects of the Drawable2D class, but not the particles. We thus transformed the ParticleGroup2D from an independent class to a child of the Drawable2D class, and adjusted the *draw()* function from

```
ParticleGroup2D& draw(std::shared_ptr<SceneGraph::Camera2D>&
 ↪  camera, Int screenHeight, Int projectionHeight);
```

to

```
void draw(const Magnum::Matrix3& transformation,
 ↪  Magnum::SceneGraph::Camera2D& camera) override;
```

where the *screenHeight* and *projectionHeight* are just window parameters for the rendering.

3.3 Additional and further achievements in the VoxelizedGridSolver2D class

The FlowSolverRenderer class takes as main argument a flow solver, in particular a voxelized grid solver from the VoxelizedGridSolver2D class in our example. Even though not being the central part of our work, this section consists in documenting the VoxelizedGridSolver2D class which is responsible for updating the behaviour of the simulated flow over time. For this part, note that instead of the particle renderer, we used the scalar field renderer from the ScalarFieldRenderer class.

When running the simulation, one can update the flow by one time-step with the *"Step simulation"* button, or continuously updating it with *"Run Simulation"*. In a first time, let us describe how the voxelized grid solver is constructed, and then we will discuss how the update step works when the time-step is incremented.

3.3.1 VoxelizedGridSolver2D constructor

The VoxelizedGridSolver2D constructor is relatively straight forward and starts with declaring all the necessary grid variables : velocity, auxiliar velocity (which serves as a sibling of the velocity and is practical for incremental steps, where both previous and new velocity might be handy. Both attributes are added to the edge), pressure and divergence (both cell attributes), as well as vorticity (vertex attribute). The *initializeGridVariables()* function takes also care of detecting the smoke and obstacle objects on the grid.

The next important step is to identify which cells are voxelized with *updateVoxelizedCells()*, where at time 0, only the boundary and the obstacle are marked as voxelized cells. Once this is done, we finally run over all grid vertices and update the poisson metrix with a laplacian discretization. If the neighbour vertices of the considered vertex are voxelized (set to true with the *updateVoxelizedCells()* function), then the value of the current vertex is incremented whereas the neighbour vertex is set to *-1*. More formally, if we currently look at the vertex with coordinates *(i,j)*, we will increment the poisson matrix at this position if the left neighbour *(i-1, j)* is voxelized. In this case, the left neighbour poisson matrix element is set to *-1*, and we repeat this for all neighbour vertices (top, bottom, left and right). This results in a sparse poisson matrix, and thus those operations are performed with triplets for the sake of efficiency. In a last step, the pressure variables are initialized (divergents and pressure) with vectors of length equal the number of grid cells.

3.3.2 VoxelizedGridSolver2D *update()*

The *update()* function of our voxelized grid solver is the most important part of this class and represents the core of the simulation. We start with doing an extra step (which will be performed only once at the very beginning of the simulation) necessary to fully interpolate the velocities from the grid and solve the system (more to the system solving below). We then update the particle velocities one first time and advect our system. We then add external forces to our

system with *applyForces()*, which applies a hot smoke source (by spreading the smoke attributes from the previously smoke-labeled vertices) and add buoyancy (basically adds to the auxiliary velocities a vertical force proportional to the velocity difference of temperature and smoke).

After having applied external forces, we can finally solve the pressure. The *project()* function is achieved in 3 steps and requires a few insights : note that the whole pressure projection is performed with multithreading (OpenMP) in order to exploit all local cores and achieve maximal performance. The first step of solving the pressure is to compute the updated divergences for all cells (*updateDivergents()*) by taking the gradients from the current edge to next edge in x and y direction. We then solve the pressure system (*solvePressure()*) with the divergent attributes by calling the *solve()* function from the PoissonSolver class. Finally, in a similar spirit than for the *updateDivergents()* function, we run over all cells and compute the pressure gradients from the left and bottom cell (*(i-1, j)* and *(i, j-1)*) to the current cell (*applyPressureGradients()* function).

In a last step, we can finally update the particles a second time from the grid mesh (post projection update step), and all this is repeated every time the *update()* function is called.

3.3.3 Results

In this final subsection, we want to present a few visual results of the current status of the VoxelizedGridSolver2D class and give the reader an idea about what was achieved so far. The following screenshots show a smoke simulation (with emission in the bottom center of the grid) interacting with a solid object in the grid centroid and being subject to buoyancy. For better visualization, we adjusted the arrow sizes of the vector field for every taken image. All images have been generated with a MacBook Pro 2015, dual core Intel i5 CPU at 2.7 GHZ with 8GB RAM.

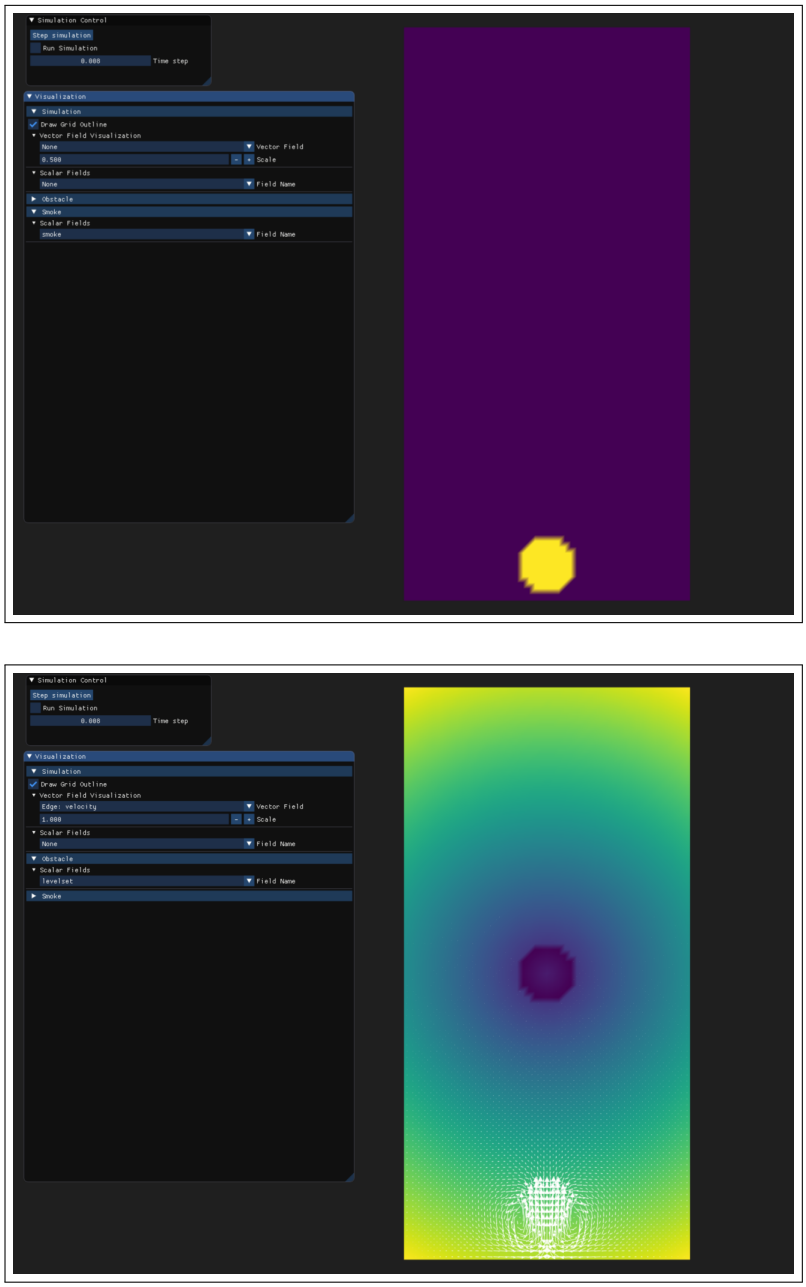


Figure 3.5: *Screenshot from the Chimera Application smoke simulation after 1 time-step (top : smoke scalar field / bottom : obstacle scalar field with edge velocity vector field).*

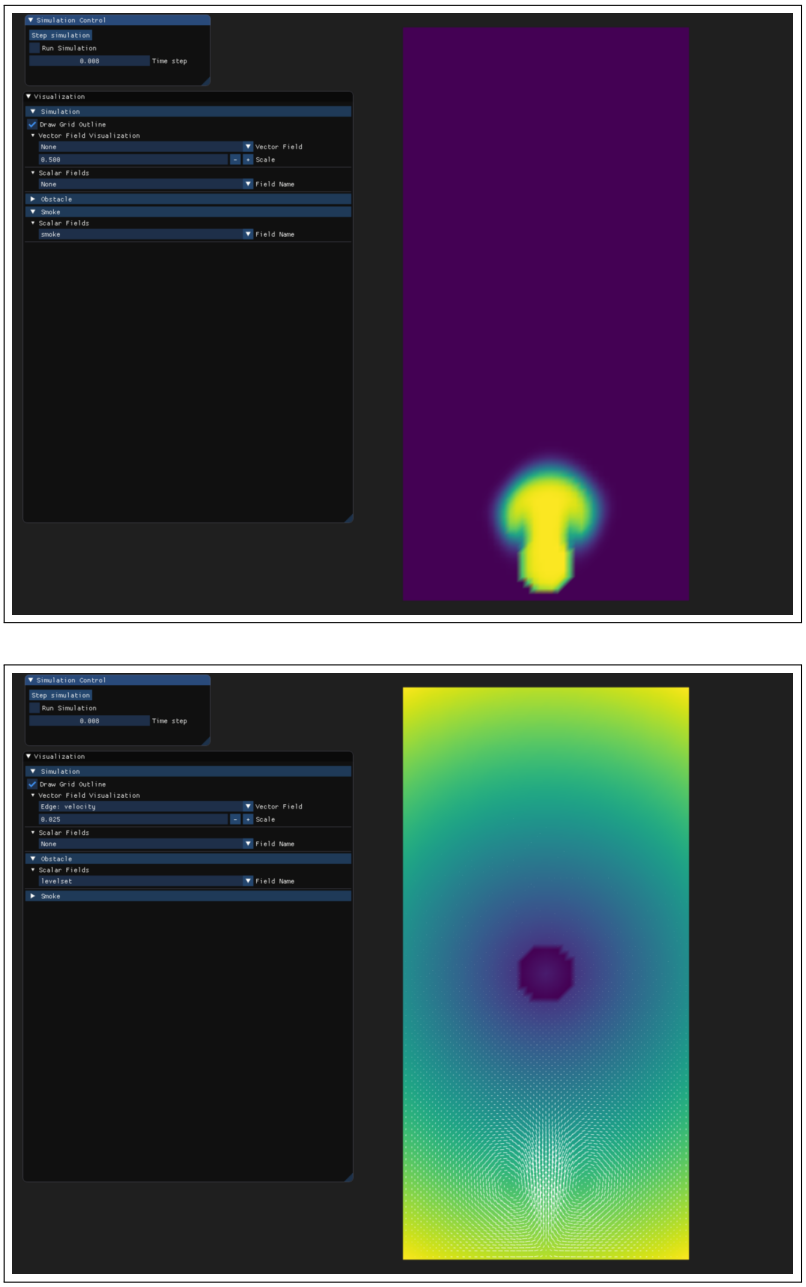


Figure 3.6: *Screenshot from the Chimera Application smoke simulation after 60 time-steps (top : smoke scalar field / bottom : obstacle scalar field with edge velocity vector field).*

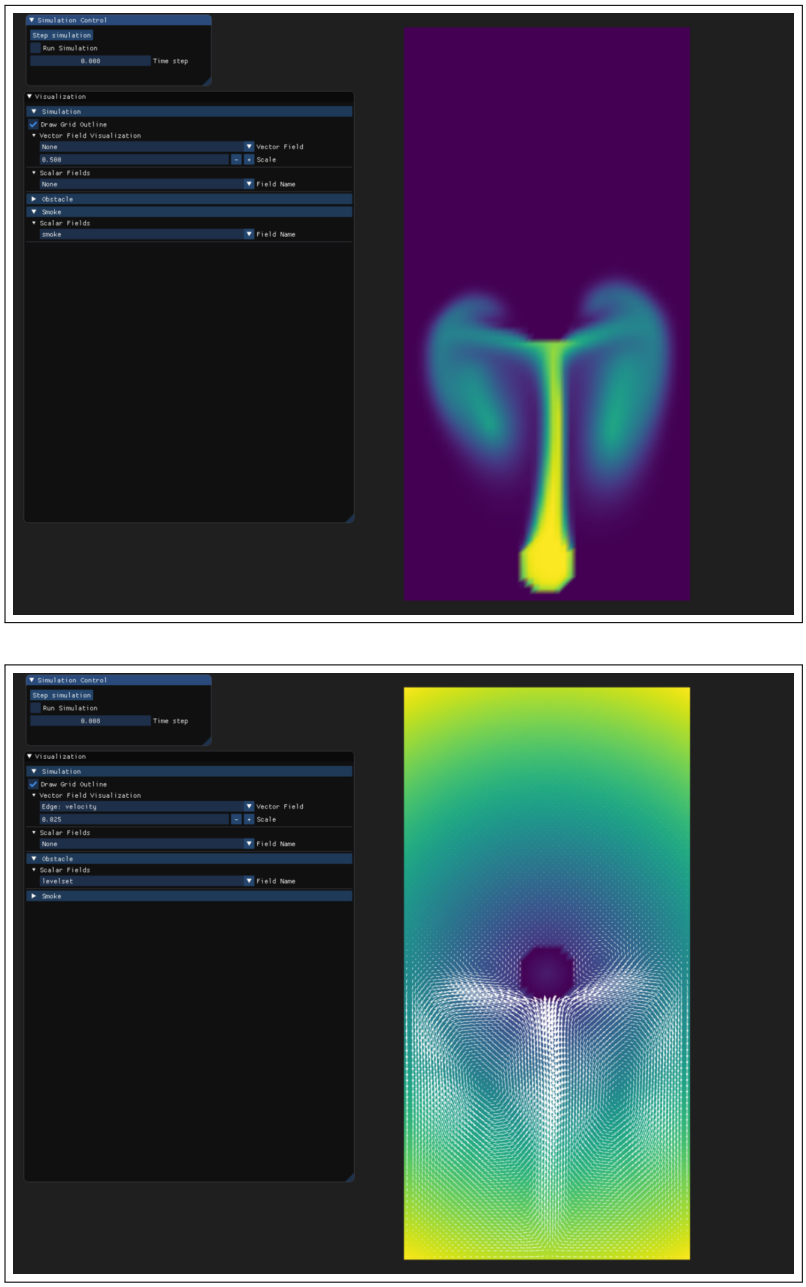


Figure 3.7: *Screenshot from the Chimera Application smoke simulation after 160 time-steps (top : smoke scalar field / bottom : obstacle scalar field with edge velocity vector field).*

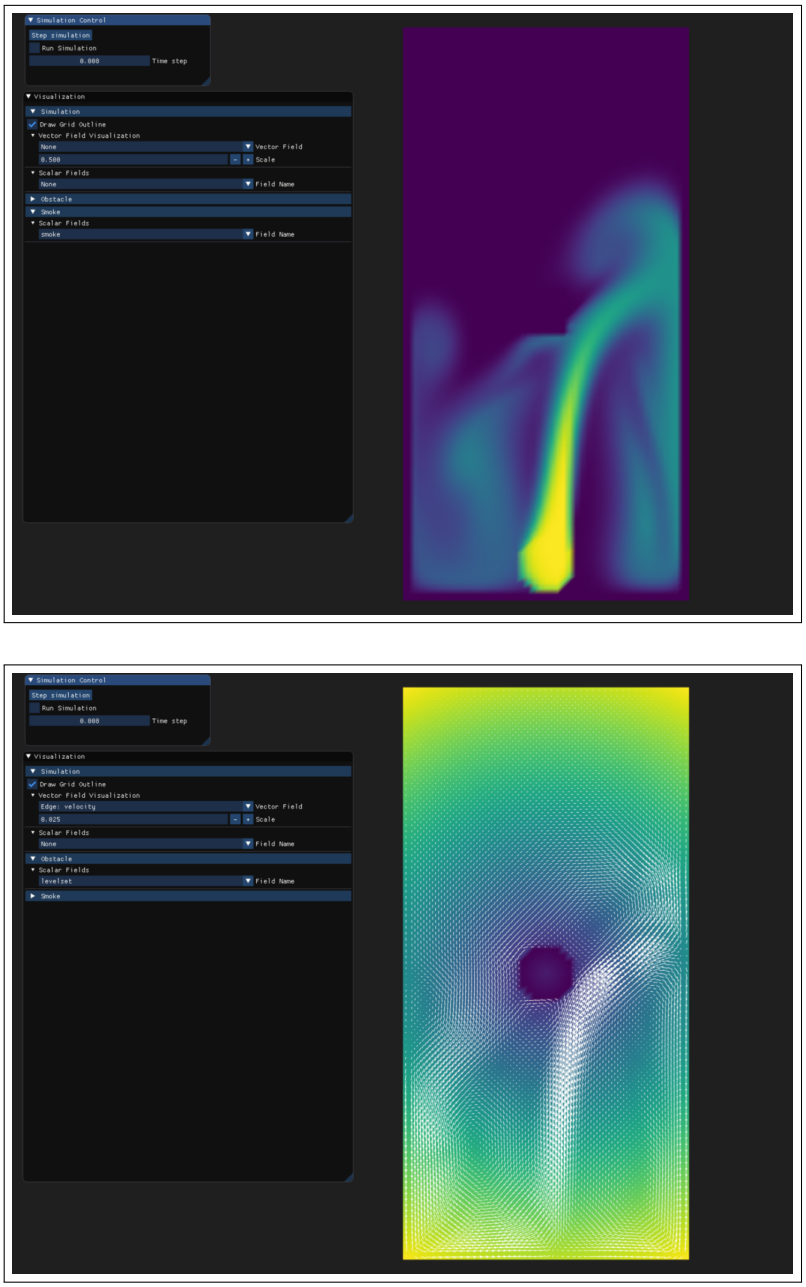


Figure 3.8: *Screenshot from the Chimera Application smoke simulation after 260 time-steps (top : smoke scalar field / bottom : obstacle scalar field with edge velocity vector field).*

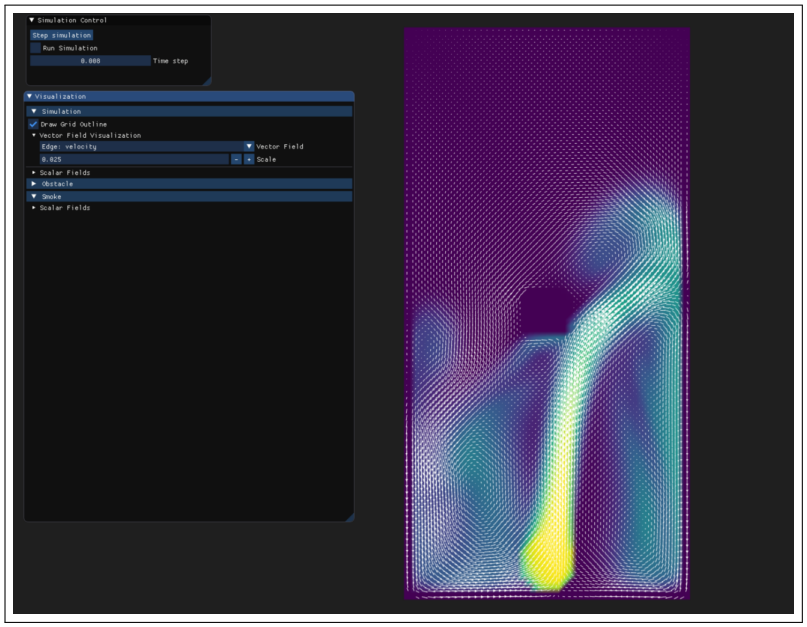


Figure 3.9: *Screenshot from the Chimera Application smoke simulation after 260 time-steps (smoke scalar field with edge velocity vector field).*

4

Conclusion and Outlook

In this thesis, we mainly looked into the ChimeraRendering pipeline and simplified it with the Magnum framework. While doing so, we encountered multiple implementation and debugging challenges, as well as obstacles which slowed down the progress and delayed the initial cut-cell extension objective.

We started with detailing the ChimeraRendering structure and how rendering in Magnum is done, preceding our implementation approach and insights about the VoxelizedGridSolver2D class. Although not achieving the fixed goal, we however managed to get a working particle rendering setup, and further dived into the voxelized grid flow solver of our simulation. Now that the flow solver is functional, it might be interesting to further extend it to the particle renderer for liquid simulation, and the subsequent step would naturally be to further develop the cut-cell solver in the CutCellSolver2D class.

Bibliography

[ABO16] Vinicius C. Azevedo, Christopher Batty, and Manuel M. Oliveira. Preserving geometry and topology for fluid flows with thin obstacles and narrow gaps. *ACM Transactions on Graphics*, 35(4), 2016.

[Ber17] M. Berger. Cut cells: Meshes and solvers. In *Handbook of Numerical Analysis*, volume 18, chapter 1, pages 1–22. New York University, New York, NY, United States, 2017.

[CJ98] Phillip Colella and Hans Johansen. A cartesian grid embedded boundary method for poisson's equation on irregular domains. *Journal of Computational Physics*, 147:1:60–85, 11 1998.

[FMOA99] R P Fedkiw, B Merriman, S Osher, and T Aslam. A non-oscillatory eulerian approach to interfaces in multimaterial flows (the ghost fluid method). *Journal of Computational Physics*, 152:2:457–492, 7 1999.

[TBFL19] Michael Tao, Christopher Batty, Eugene Fiume, and David I.W. Levin. Mandoline: Robust cut-cell generation for arbitrary triangle meshes. *ACM Trans. Graph.*, 38, 6, Article 179 (November 2019), 2019.

[ZGZJ16] Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson. Mesh arrangements for solid geometry. *ACM Transactions on Graphics*, 35(4), 2016.