MSc Masters Project
3D Particle in Cell / Fluid Implicit Particle Fluid Solver using OpenMP directives

Ioannis Ioannidis
MSc CAVE 11-12
August 20, 2012

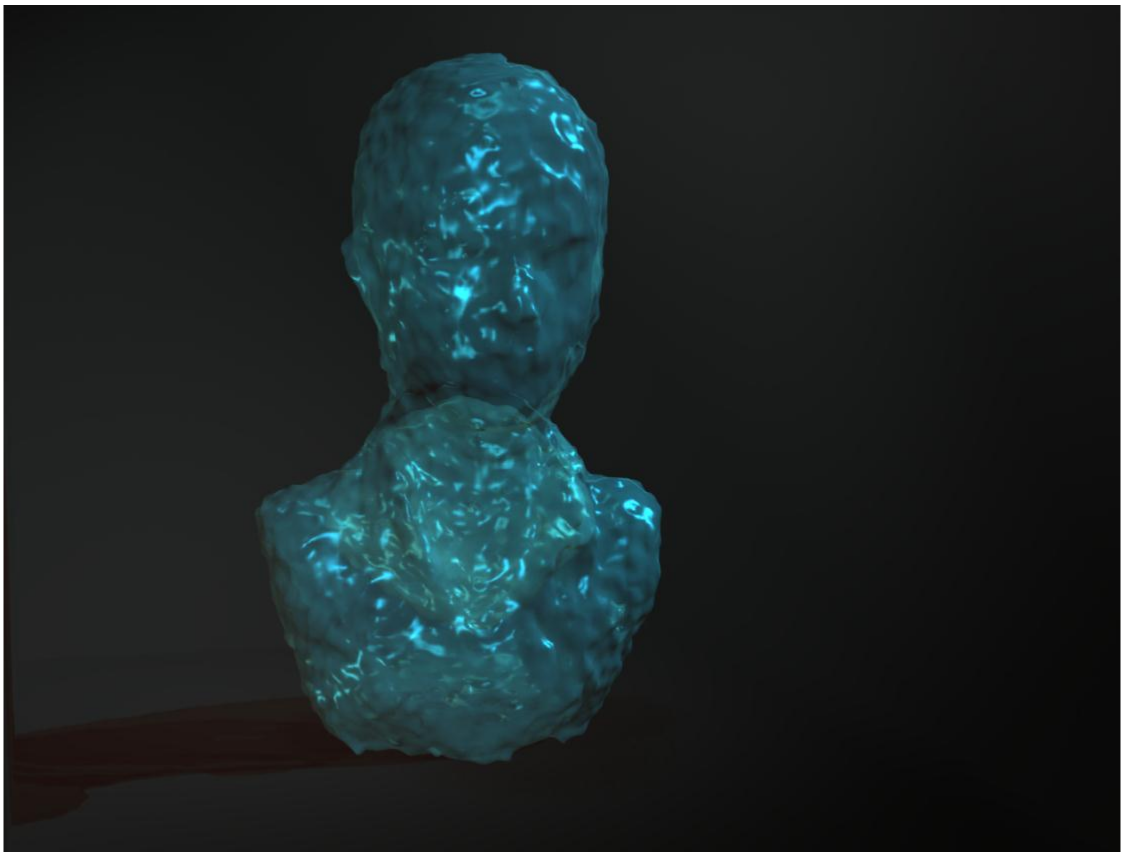


TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Abstract

This master's project paper deals with the implementation of a Pic Flip fluid solver implemented with OpenMP as a plugin for SideFX Houdini 3D Package. First and foremost, a standalone pic/flip solver is created in C++, using OpenGL and NGL graphics library (Bournemouth University's graphics library). Secondly this fluid solver's execution is parallelized using OpenMP compiler directives so as to exploit all available CPU cores of the current running system. In addition, as simulation is controlled from inside Houdini, 2 HDAs (Houdini Digital Assets) are created, which set and store all available fluid geometry data and settings for the custom external fluid solver. Last but not least, an interface is attached to the solver for parsing an .xml which contains all settings needed for the solver, plus a .geo file writer of point position data to a Houdini compliant file format is attached as the means to communicate the simulation data back to Houdini scene space. Additionally, a new feature regarding the rotational motion of the fluids being simulated has been added to the solver as well, which introduces an artificial vortex-like behaviour during the fluid flow.

1 Introduction

Fluid simulations for generating realistic animations of water, smoke are extremely useful in the field of visual effects especially in films but also in commercial projects such as advertising. This is mainly owed to the fact that are fully controllable (because they are pre-programmed to behave in a certain way), but most importantly because it is much more easier, cheaper and safe to create in ex. fake huge ocean waves compared to filming in the actual ocean, or huge explosions and so on.

1.1 Topic

In this thesis report the implementation of a custom 3D fluid solver as a plug-in for SideFX Houdini 3D package using the particle-in-cell / fluid implicit particle method is covered and analyzed. It is mostly based on the Siggraph paper “Animating Sand As Fluid” by Zhu and Bridson (2005) [2] and on the “Fluid Simulation for Computer Graphics” book by Robert Bridson [1] from which someone can form a deep and meaningful background in the two fluid simulation methodologies which are combined under Pic/Flip model. A Pic/Flip custom fluid plug-in is investigated, designed and implemented so as that it is ready to be used iteratively through Houdini 3D package and offer the opportunity for technical directors or artists to create and direct sequences of fluid simulations using their custom set of preferences inside Houdini.

1.2 Project Objective

The project uses C++ and OpenGL to implement the fluid solver having this is as it's initial objective as it's a programming project. Secondly, Houdini Digital Assets are designed and created to load pre-created fluid object geometry data inside Houdini and plug their data into a custom fluid solver so to be solved. Of course, a custom c++ solver has been implemented to solve the fluid flow using the Pic/Flip method. Furthermore, an interface between Houdini and the custom Pic/Flip solver is implemented using an xml file format parser in C++ to import and read a configuration settings file created inside Houdini using Python scripting embedded as one of the various functionalities of HDAs. Moreover, point position data are written per frame in .geo file format so as to pass the fluids' particle positions back to Houdini to a readable and compliant format. Finally, the custom fluid solver's execution time gets boosted by parallelizing it using OpenMP compiler directives.

1.3 Report Structure

- Chapter 2 introduces the area of research through presenting previous work related to the project.
- Chapter 3 will analyze the main theory behind particle-in-cell / fluid implicit particle (Pic / Flip) methodology, plus some general basic fluid dynamics theoretical concepts that will allow the reader to form the proper theoretical background and carry on reading this thesis.
- Chapter 4 will present the design of this custom plug-in for Houdini 3d Package as an external application.
- Chapter 5 will exhaustively analyze the details of the implementation of the plug-in as a whole.
- Chapter 6 will present the overall pipeline and several issues and steps that are needed to be followed so as for the custom solver to work as an external application.
- Chapter 7 will discuss the results of the implementation
- Chapter 8 will form a conclusion for the project and suggest possible future work that could improve the existing solution in different aspects.

2 Previous Work

Pic/Flip methodology has its roots back to 1963 by Harlow [8] and then it was improved in the 1980's and more specifically in 1986 when Brackbill et. all [6] in the journal "*FLIP: A method for adaptively zoned particle-in-cell calculations in two dimensions*" presented the fluid implicit particle fluid methodology based algorithm theory of particle in cell (PIC) method. They also addressed and almost completely solved the problem of numerical dissipation in PIC method.
Later on, in 1988 they developed the FLIP (Fluid-Implicit-Particle) method that uses

fully Lagrangian particles to eliminate convective transport which was the main and largest source of in calculations of computational diffusion in fluid flow (Brackbill, 1988). In FLIP method here the Lagrangian particles were used and contained all necessary information to describe the fluid and through using these particle data, the Lagrangian moment equations were solved on a grid. The results of those equations then used as solutions to advance the particle variables at each simulation time step. The real strength of FLIP method which conserves the angular momentum and the absence of numerical dissipation is presented in this journal.
There has been a lot of improvements the main PIC/FLIP methodology since then with culminating in the Siggraph paper *"Animating Sand as Fluid" by Zhu and Bridson (2005) [2]*, in which they described a method for converting an existing fluid solver into one capable of plausibly animating granular materials such as sand. In their paper they developed a new fluid algorithm that combines the strengths of both particles and grids, offering enhanced flexibility and efficiency while offering a new method for reconstructing implicit surfaces from particles; this paper is considered as a milestone to the PIC/FLIP methodology due to its excellent conservation of energy without the spurious oscillations associated with similarly undissipative central difference schemes.
In 2008, R. Bridson in his book "*Fluid Simulation for Computer Graphics*" presents and analyses extensively the similarities and differences as well as the equations of the Eulerian and Lagrangian methods used in fluid simulations. Its contents form a deep and meaningful background in those two fluid simulation methodologies which are combined under PIC/FLIP model. It helps clarifying the theoretical structure of algorithmic approaches related to these kind of fluid methods which are then fitted together to create the hybrid PIC/FLIP method itself.
Furthermore, Ando et. Al (2012) in their technical paper *"Preserving Fluid Sheets with Adaptively Sampled Anisotropic Particles"* present a particle-based model for preserving fluid sheets of animated fluids with an adaptively sampled Fluid-Implicit-Particle (FLIP) method. In addition, preserving fluid sheets by filling the breaking liquid sheets with particle splitting in the thin regions, and by collapsing them in the deep water is extensively analysed in this paper as well.
Finally, , in 2011 Boyd and Bridson in their Siggraph paper "*MultiFLIP for Energetic Two-Phase Fluid Simulation*" presented the MultiFLIP method to graphics, which is an extension of the Fluid Implicit Particle (FLIP) method [Brackbill and Ruppel 1986] , and adapted it to incompressible flow by Zhu and Bridson [2005]. They developed a new particle-based surface tracking scheme which identifies sub-scale bubbles and droplets (Boyd, 2011) . MultiFLIP method incorporates particles into the grid-based fluid solve so they are subject to the same physical parameters as the rest of the simulation.

3 Basic Fluid Dynamics and Pic / Flip Theory

3.1 Fluid Dynamics Research Approaches

There has been a variety of technical methods researched to simulate 3D fluids properly already with the most exceptional ones to be the Eulerian or (Grid-Based), the Lagrangian or (Particle-Based), the Lattice Boltzmann and the Vorticity-Based researched methods. All these methods have gradually been adapted in Computer Graphics industry and the world of Visual Effects so as to produce realistic pieces of water and smoke animation mostly. Since the method used in this project is the Pic / Flip and since its based at a combination of two of the aforementioned (Eulerian & Lagrangian) there will be only a brief description on those two related ones.

3.1.1 Eulerian Grid Based Methods

The Eulerian description of a system, generally considers the range of change of system variables fixed at a particular point in space. So, as time evolves, all the physical term properties associated with the fluid particles such as pressure, density or velocity that pass through a fixed region of space are being measured and examined. This could be further interpreted as points sticked in a grid which measure any quantitative attributes of particles that pass by those points over time. All the particle data information such as velocity, pressure, fluid surface indices and so on are kept usually on a grid.

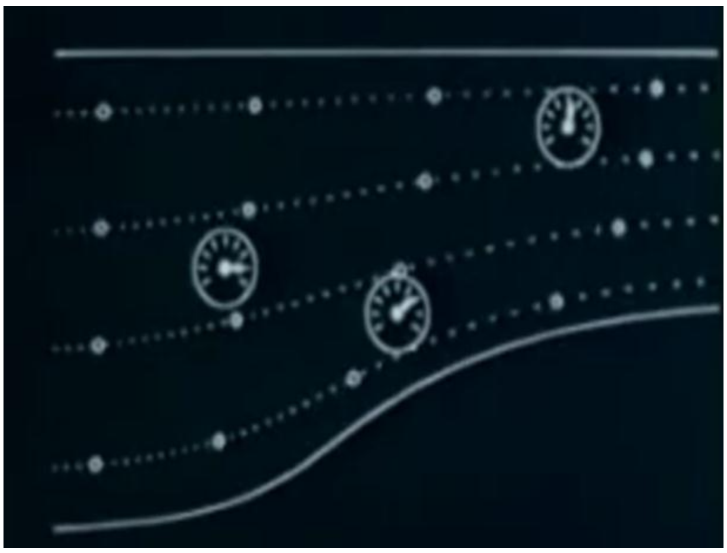


Figure 1 : Eulerian Particles Passing through fixed points in space (by the National Committee for Fluid Mechanics)

3.1.2 Lagrangian Particle Based Methods

The Lagrangian system on the other hand, considers individual particles which correspond to individual fluid parcels and which the system tracks and follows them along their trajectories in time and space. Each of those particles carry all the fluid information such as acceleration, viscosity, density, pressure or velocity and those quantitative material terms are measured in motion.

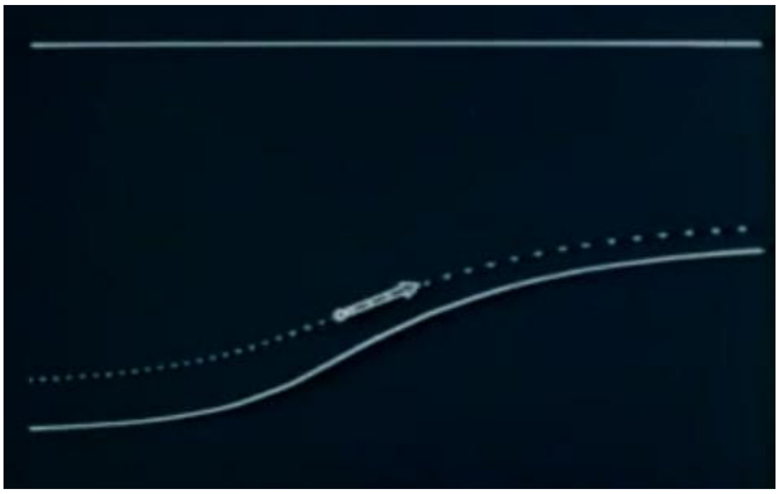


Figure 2 : Lagrangian Particle moving through space and time with a velocity vector attached to it.
(by the National Committee for Fluid Mechanics)

3.1.3 Theoretical Comparison Eulerian Vs Lagrangian

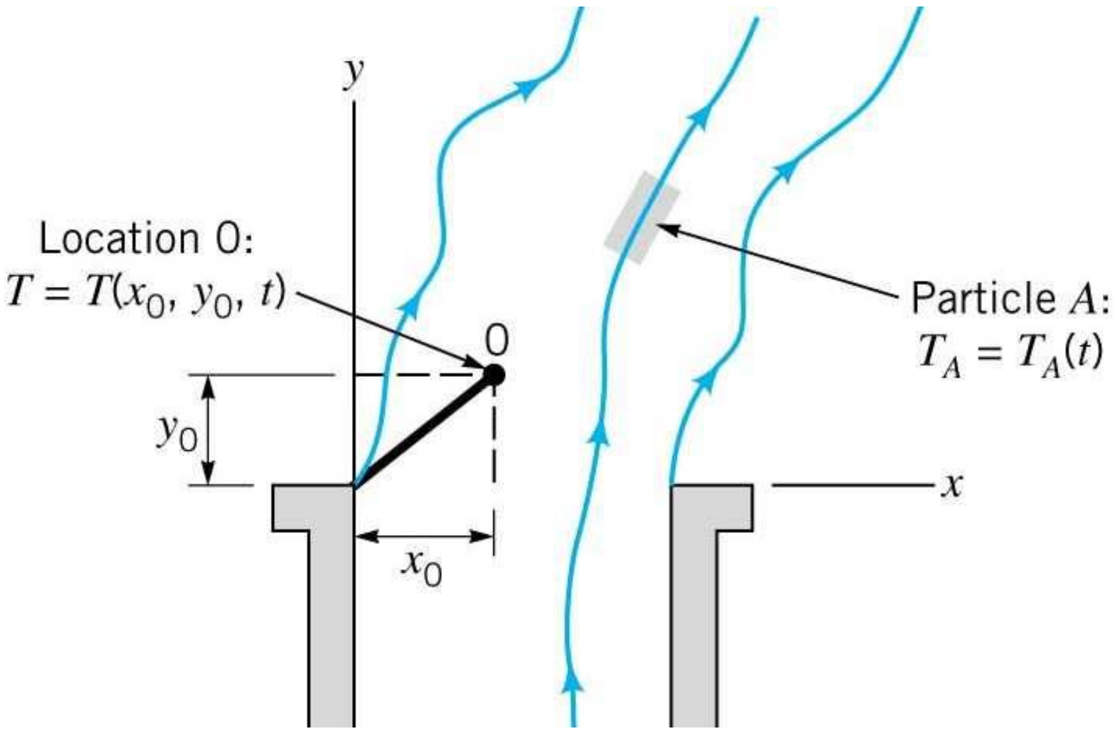


Figure 3 : Eulerian (left) – Lagrangian (right) system visual comparison

(by www.cs.cdu.edu.au/homepages/jmitroy/eng243/sect05.pdf)

To start with, while the grid-based methods provide impressive results, the particle-based methods could provide an alternative for simulation and animation of variety of fluids with different physical properties while allowing user control and fast feed-back at coarse resolutions.

Listing the advantages of Particle based approaches we could say:

- Overall complexity of the simulation is reduced
- Mass conservation is ensured
- Convection terms are dispensable: and so pressure and density can be computed from weighted contributions of neighbouring particles rather than by solving linear systems of equations
- Particles can be used directly to render the fluid surface
- Large deformations are treated relatively easier because particles have no fixed connectivity
- Particle refinement is way easier than mesh-refinement(in Eulerian Grid-based methods)

One of the most important disadvantages of particle-based methods (as usually stated) is the need for large numbers of particles to produce simulations of equivalent resolution as grid-based methods.

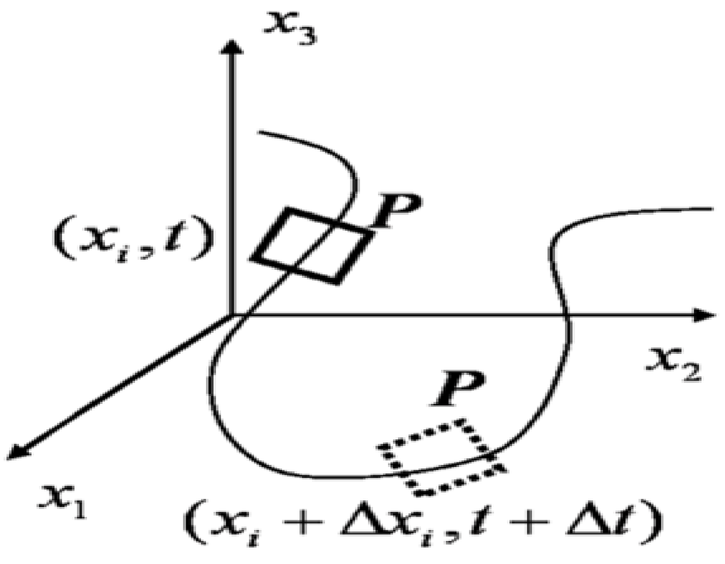


Figure 4 : Individual Lagrangian Particle moving through space and time (by http://en.wikiversity.org/wiki/Fluid_Mechanics_for_MAP_Chapter_4._Fluid_Dynamics)

Listing the advantages of Grid based approaches we could say:

- Higher visual accuracy is ensured generally
- They maintain better Surface Representation
- Performance is independent of the particles' number

One of the most important disadvantages though is the numerical dissipation because of the fluid domain sampling, plus it has a large memory footprint. Moreover, the detail of the simulation is largely dependent on the grid resolution hence the size of the simulation depends on the grid's size.

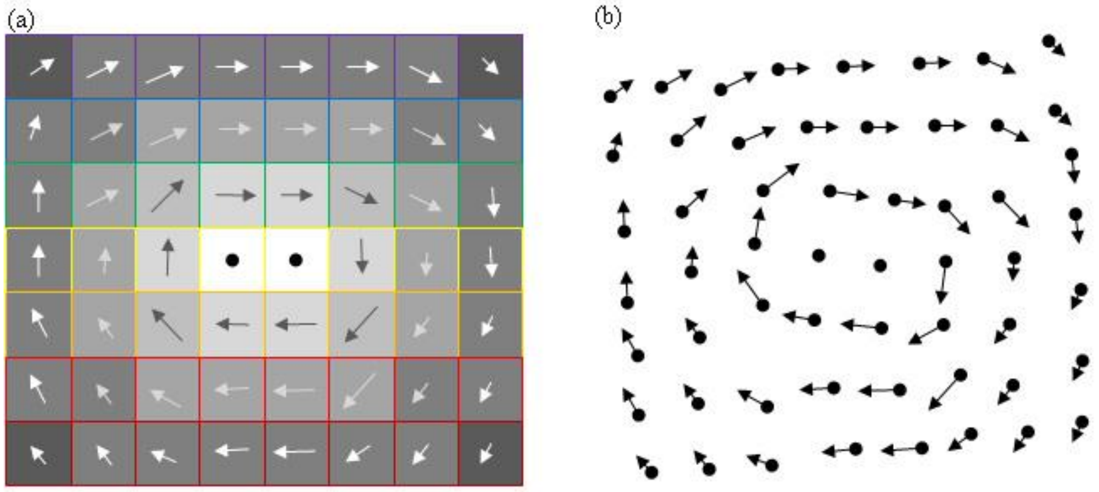


Figure 5 : Field-based and particle-based views of a fluid. (a) Grid based: Each point has fluid properties like velocity (arrows), density (box fill), pressure (arrow color), and temperature (box outline), and the grid points never move. (b) Particle based: Each particle has fluid properties in addition to position*, and each particle can move*

(by http://software.intel.com/en-us/articles/fluid-simulation-for-video-games-part-1/)

3.2 Staggered MAC Grid

In this thesis a staggered MAC grid is used so as to store different quantative variable terms in different locations because it's a simple model to avoid the "Odd-Even" numerical issue when trying to compute pressure on the grid [21]. If we were to look at a simpler form of the MAC grid in 2 dimensions we can see the pressure in grid cell (i, j) is sampled at the center
of the cell, indicated by Pi,j and the velocity is split into its two components regarding each direction of the system (grid's cell faces). The horizontal u component is

sampled at the center of the vertical cell's faces, and is indicated by Ui+1/2,j for the horizontal velocity between cells (i, j) and (i+1, j). The vertical v component is sampled at the centers of the horizontal cell faces and is indicated by Vi,j+1/2 for the vertical velocity between cells (i, j) and (i, j +1). This way, having acquired each grid cell's velocity at the center of each of its 6 faces we are able to calculate the amount of fluid flowing into and out of the cell (Bridson, 2007).

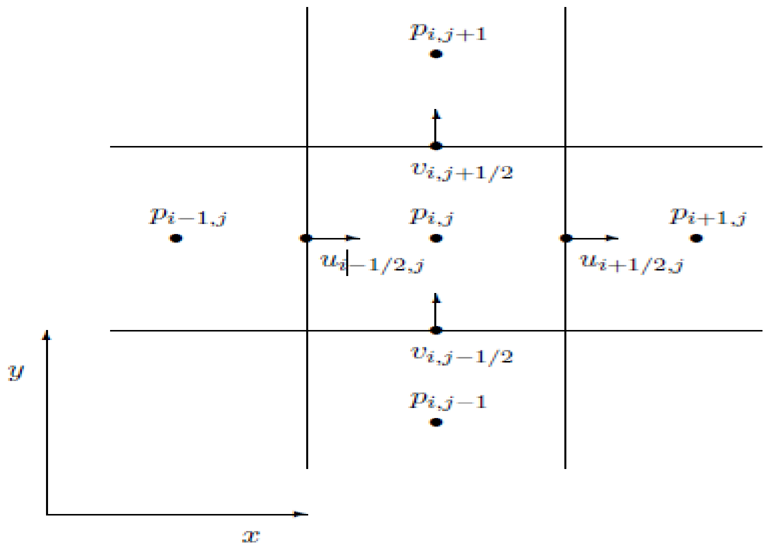


Figure 6 : 2D staggered MAC grid showing pressure values stored in the center of each cell and velocities on the center of each face of the cell (part 1) (by http://www.cs.ubc.ca/~rbridson/fluidsimulation/fluids_notes.pdf)

The same goes for the 3D representation of a MAC grid:

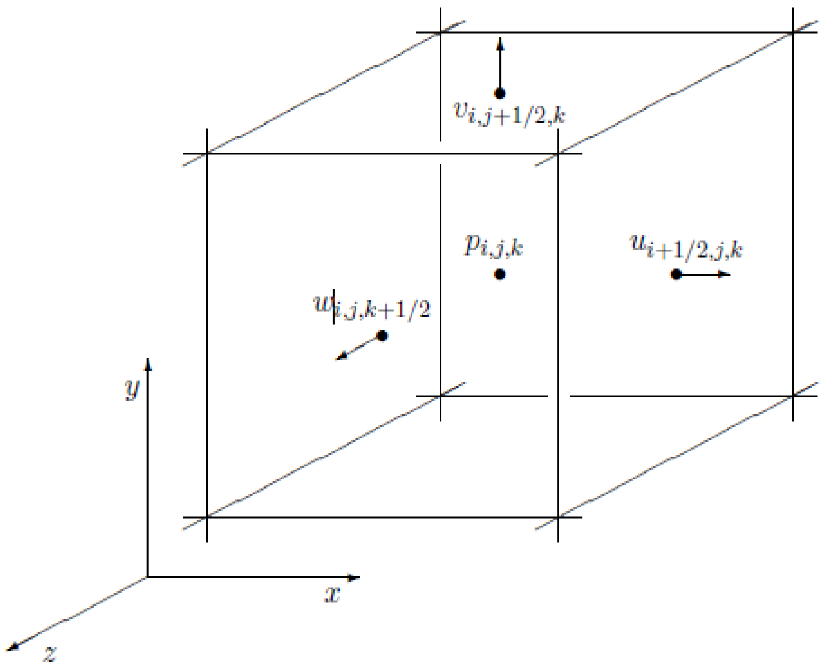


Figure 6: 3D staggered MAC grid showing pressure values stored in the center of each cell and velocities on the center of each face of the cell (part 2) (by http://www.cs.ubc.ca/~rbridson/fluidsimulation/fluids_notes.pdf)

3.3 Equations of Fluid Motion

Navier Strokes Equations:
The governing equations that are used to describe the motion of a fluid are called the Navier-Strokes equations and result from applying Newton's second law to fluid motion:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{g} + \nu \nabla \cdot \nabla \vec{u} \qquad (3.1)$$
$$\nabla \cdot \vec{u} = 0$$

Figure 7 : Navier Strokes Equations (by http://en.wikipedia.org/wiki/Navier%E2%80%93Stokes_equations)

In the above figure u denotes the velocity , ρ the density, p the pressure, ν the kinematic viscosity and f the external forces acting on the fluid. The symbol ∇ represents the spatial derivative vector in 3 dimensions:

$$\nabla = \left(\frac{\delta}{\delta x}, \frac{\delta}{\delta y}, \frac{\delta}{\delta z} \right) \tag{3.2}$$

The first differential equation of the two of the Navier Strokes equations is called the momentum equation and it practically describes that the fluid accelerates and consequently moves when external forces are applied onto it.
The second equation deals with what in computational fluid dynamics we call incompressibility condition. That basically means, that the mass of an infinitesimal fluid parcel has to remain constant throughout the fluid flow and so the mass is conserved.
Although, in reality fluids are not incompressible and indeed change their volume (ex. supersonic flows, sound under water), in fluid simulations especially for computer graphics we tend to ignore this parameter for the sake of simplicity in terms of coding and performance; This is because we are mostly interested in an acceptable visual outcome which is subject to a little deviation from the strict physical laws.

$$\overbrace{\rho\Big(\underbrace{\frac{\partial \mathbf{v}}{\partial t}}_{\substack{\text{Unsteady}\\\text{acceleration}}} + \underbrace{\mathbf{v}\cdot\nabla\mathbf{v}}_{\substack{\text{Convective}\\\text{acceleration}}} \Big)}^{\text{Inertia (per volume)}} = \overbrace{\underbrace{-\nabla p}_{\substack{\text{Pressure}\\\text{gradient}}} + \underbrace{\mu\nabla^2\mathbf{v}}_{\text{Viscosity}}}^{\text{Divergence of stress}} + \underbrace{\mathbf{f}}_{\substack{\text{Other}\\\text{body}\\\text{forces}}} . \tag{3.3}$$

Figure 8 : Analytical Fluid Form of Navier Strokes Equations
(forces acting on the fluid (right side) cause the acceleration of the fluid (left side))
(by http://en.wikipedia.org/wiki/Navier%E2%80%93Stokes_equations)

Analysing the terms of the momentum equation gives us a better clarification on the whole fluid motion and so we can describe each term as follows:

- $\theta v/\theta \tau$: is the unsteady acceleration of each of the individual particles of the fluid
- $v\nabla v$: is the convective acceleration caused by a (possibly steady) change in velocity over position, for example the speeding up of fluid entering a converging nozzle. It's also known as the advection term that moves the fluid

- $-\nabla p$: is the pressure gradient which describes how the volume of the fluid moves with respect to the pressure attribute field and is responsible for maintaining the fluid incompressible (solved in the step called Projection).

- -$\mu\nabla\nabla v$: is the kinematic viscosity which is also called diffusion as well, representing the whole fluid volume's internal resistance to motion. For example, types of fluids like water, blood, honey and molasses have different amounts of viscosity with the first having the lowest and the latter the highest.

- At Last, f represents the external forces that act on the fluid with the most common and simple one be the gravity itself with approximately 9.81 m/sec^2.

3.4 PIC / FLIP Methods

3.4.1 Particle In Cell Method

The Particle-in-Cell (PIC) method which was early approached by Harlow (1963) [8] was a method for simulating fluid flows using particles for the advection step. All the other simulation steps were computed on a grid. All particles' quantative terms were replaced by the weighted average of neighbour particles' quantative terms' values and then were updated on the grid without the doing the advection step. An interpolation of the newly found particle values on the grid and the previous particle values was undertaken and in the end the particles were moved based on the grid's velocity field. What this method was suffering from was the numerical diffusion caused by the continuous averaging and interpolating of the fluid variables (Bridson, 2005).

3.4.2 Fluid Implicit Particle Method

Later, in 1986 Brackbill and Ruppel in their paper "FLIP: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions " [5] achieved almost no numerical dissipation while at the same time having many variations of fluid shapes by using particles as the main fluid volume representation and by using the grid so as to increment the particle quantative terms based on the change calculated on it (Bridson, 2005).

3.5 PIC / FLIP Combination Algorithm

By adapting PIC and FLIP methods to ensure an incompressible fluid flow as we have the following generic algorithm (following Bridson's 2005 "Animating Sand as Fluid" paper):

- We initially set all particle positions & velocities
  For each time step:
- For each MAC grid cell we calculate the weighted average of neighbouring particle velocities
- For FLIP: We store the grid velocities
- Perform all the non-advection steps of a standard water simulator on the grid
- For FLIP: Subtract the new grid velocities from the previously stored velocities, then add the interpolated difference to each particle's velocity.
- For PIC: Interpolate the new grid velocities to the previous particles' velocities.
- Mix Pic and Flip velocities
- Move particles through the grid velocity field with an ODE solver, making sure to push them outside of solid wall boundaries
- Write the positions of the particles to an output.

3.5.1 PIC / FLIP Algorithm Step Analysis

1. **Set initial Positions & Velocities**

   Initially, we actually place the particles inside the MAC grid's space and in this thesis we practically match Houdini's scene space particle positions to the custom solver's corresponding ones.(See section 5.2.3)

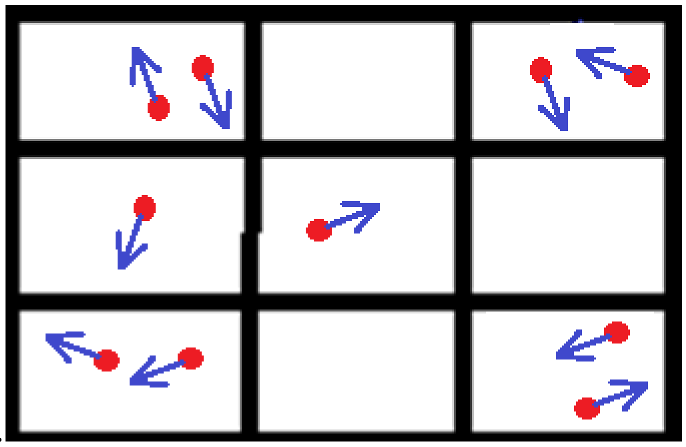


Figure 9 :Particle positions and velocities initial set up

2. **Calculate the weighted average of neighbouring particle velocities**

   In this step the particle velocities are mapped onto the grid and each of the grid points are taking a weighted average of nearby particles by performing a trilinear (horizontal, vertical and depth) 3d interpolation.(See chapter 5)

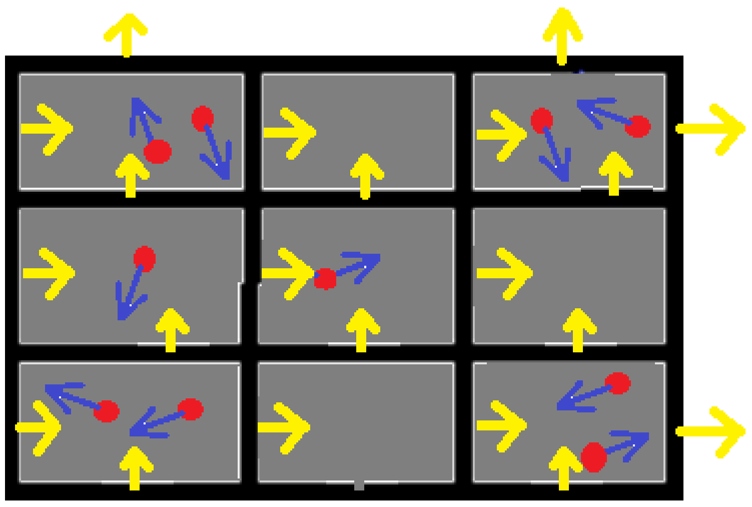


Figure 10 :Particle velocities are mapped on the grid

3. **Store grid velocities**

   In this step, we just temporarily store the grid velocities that have just been mapped to it from particles for future reference as a part of the FLIP method.

4. **Perform all the non-advection steps of a standard water simulator on the grid**

   In this step we set indicators and mark each MAC grid's cell as liquid, solid or empty cell, we set boundary cell velocities to zero, we perform the projection step to enforce the incompressibility of the fluid (see Chapter 5) and extrapolate velocities (define the fluid velocity in regions that might not be defined).

5. **For Flip: Subtract new from stored grid velocities , then add the interpolated difference to each particle velocity.**

   This step is essential for the FLIP method in which we subtract the newly calculated grid velocities (by the projection step) from the previous stored grid velocities and store them (as X). Then we temporary save the already existing particle velocities (as Y) and calculate new particle velocities (Z) by mapping back to particles the change (X) in stored grid velocities from previous and current time step . As soon as we have these new particle velocities (Z)we interpolate the difference (Z) to each particle's previously saved velocity (Y) and we call that flip velocity. (See Chapter 5)

X =	new_grid_vel – prev_grid_vel
Y =	cur_particle_vel
Z =	X
Flip =	Z + Y

Table 1 : Flip velocity step explanation

6. **For PIC: Interpolate the new grid velocity to the particles' velocities**

In this step, we calculate new particle velocities (PIC) by mapping and interpolating the new grid velocities (X) found by the projection step (and not the difference between the new grid velocities and the previously stored ones) back to particles current velocities (Y). We call theses ones PIC velocities (See Chapter 5)
.

X =	new_grid_vel
Y =	cur_particle_vel
PIC =	Y + X

Table 2 : Pic velocity step explanation

7. **Interpolate Pic and Flip velocities**

At this step, we just interpolate a value between pic and flip velocities computed so as to finally apply it to the particles velocities. A weighted average of the two is used to result the preferred numerical viscosity. Flip is mostly preferred for inviscid flows such as water whereas Pic for more viscous flows such as molasses or honey etc. (See Chapter 5).

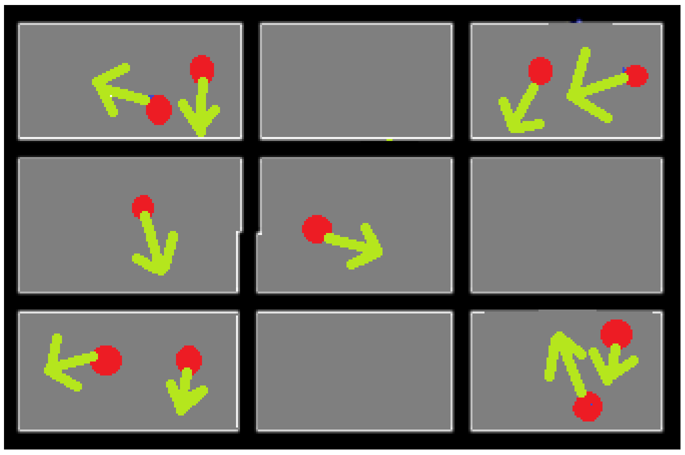


Figure 11 : New Particle velocities calculated through steps 5,6,7

8. **Move particles through the grid velocity field with an ODE solver, making sure to push them outside of solid wall boundaries**

   In this step, we simply advect the particles to new positions by using a simple and accurate ordinary differential equation solver and more specifically using a Runge-Kutta 4th order integration method (See Chapter 5).

9. **Write the positions of the particles to output.**

   This is where we take all new particle positions found based on the integration method performed at the previous step and write them to the output so as to update the fluid motion. In fact, we write them to a Houdini compatible file format so as to be loaded as separate frames when then simulation finishes (See Chapter 5).

4 Design of the custom plug-in

In this chapter all the aspects around the design of the external pic/flip solver as a custom Houdini plugin are discussed. More specifically all the initial general design modules needed till the actual implementation considerations are presented.

4.1 Aims and Objectives

The main key features needed for the actual success of the application are listed as follows:

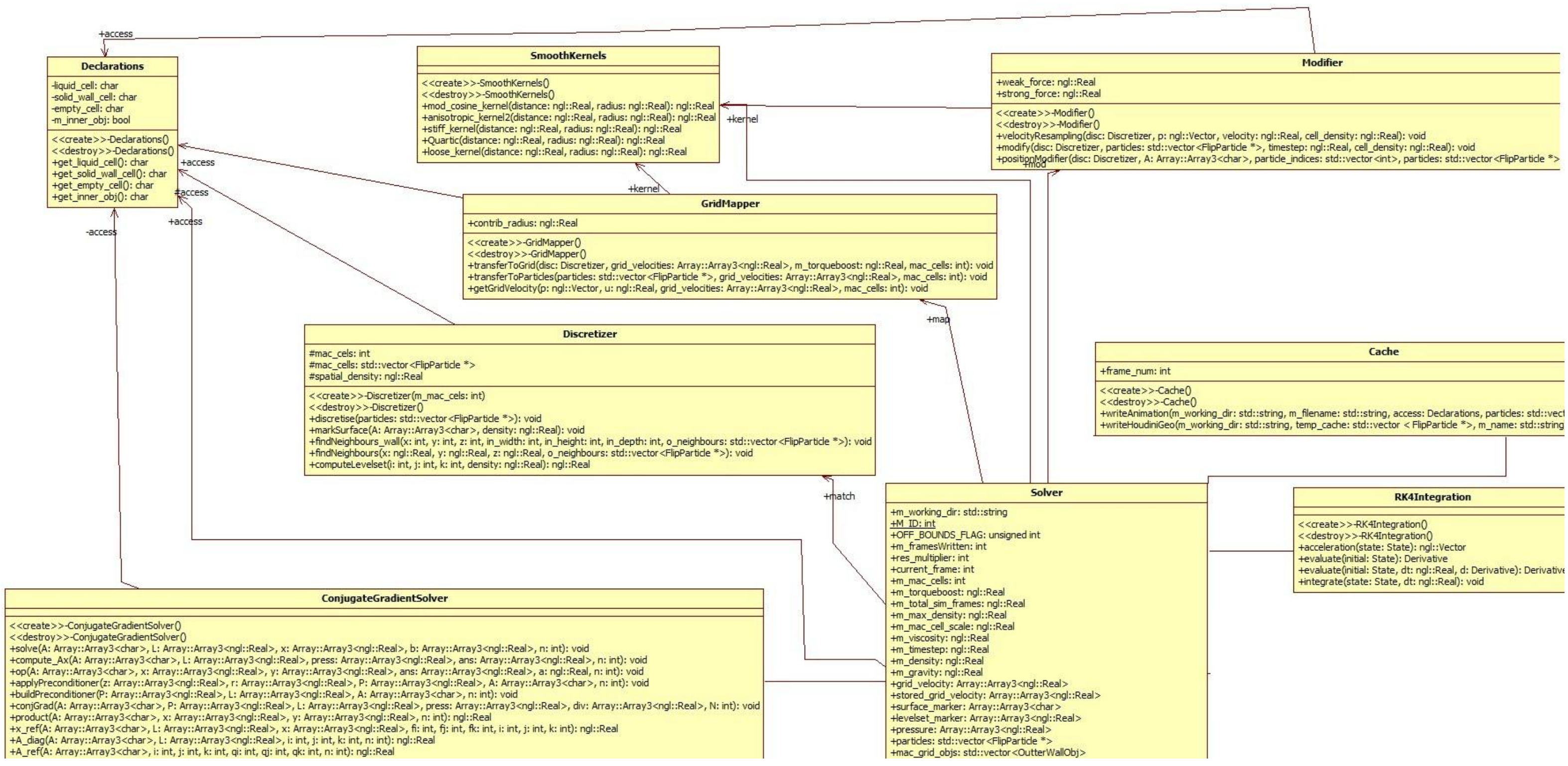
- A number of more than one of fluids should be simulated within one scene
- There should be a capability to simulated a variety of the fluids with different properties as well as the support for different set up topologies in the pre simulation stage inside Houdini.
- It should provide an interaction with static collision obstacles and bounding box areas
- It should be capable of handling large amounts of simulation data (number of particles)
- There should be an interface for importing .obj models pre-created in Houdini
- It should include a proper point position exporter for particle positions to a Houdini compliant file format (.geo) for later loading and rendering of the final simulation
- There should be a visualization window of the current simulation running in the custom solver

All if the above bullet points are further discussed, analyzed in detail and associated with their contribution to the overall design of the custom plugin.

4.2 Application Class Diagram

In figure 10 and 11 a class diagram splitted in two comprising all the classes/entities needed for the solver to work properly. Additionally, it sums up all the relationships amongst the different functionalities that each one of the classes provides to the application. To briefly guide the reader through, we should indicate that the most crucial part of the following class diagram is the Solver class itself which is the heart of the pic/flip solver and encapsulates all the needed algorithmic functionality in terms of methods which are called to solve the fluid/s. Furthermore, an XML file parser is provided to read and import a settings file created inside Houdini and stored in our application's home directory. Finally, the 3rd most important part of the whole application is controlled through the GLWindow class in which all the needed steps of the simulation are provided and called one by one at each time step.

CLASS DIAGRAM PART 1



Declarations
-liquid_cell: char
-solid_wall_cell: char
-empty_cell: char
-m_inner_obj: bool
<<create>>-Declarations()
<<destroy>>-Declarations()
+get_liquid_cell(): char
+get_solid_wall_cell(): char
+get_empty_cell(): char
+get_inner_obj(): char
SmoothKernels
<<create>>-SmoothKernels()
<<destroy>>-SmoothKernels()
+mod_cosine_kernel(distance: ngl::Real, radius: ngl::Real): ngl::Real
+anisotropic_kernel2(distance: ngl::Real, radius: ngl::Real): ngl::Real
+stiff_kernel(distance: ngl::Real, radius: ngl::Real): ngl::Real
+Quartic(distance: ngl::Real, radius: ngl::Real): ngl::Real
+loose_kernel(distance: ngl::Real, radius: ngl::Real): ngl::Real
Modifier
+weak_force: ngl::Real
+strong_force: ngl::Real
<<create>>-Modifier()
<<destroy>>-Modifier()
+velocityResampling(disc: Discretizer, p: ngl::Vector, velocity: ngl::Real, cell_density: ngl::Real): void
+modify(disc: Discretizer, particles: std::vector<FlipParticle *>, timestep: ngl::Real, cell_density: ngl::Real): void
+positionModifier(disc: Discretizer, A: Array::Array3<char>, particle_indices: std::vector<int>, particles: std::vector<FlipParticle *>
GridMapper
+contrib_radius: ngl::Real
<<create>>-GridMapper()
<<destroy>>-GridMapper()
+transferToGrid(disc: Discretizer, grid_velocities: Array::Array3<ngl::Real>, m_torqueboost: ngl::Real, mac_cells: int): void
+transferToParticles(particles: std::vector<FlipParticle *>, grid_velocities: Array::Array3<ngl::Real>, mac_cells: int): void
+getGridVelocity(p: ngl::Vector, u: ngl::Real, grid_velocities: Array::Array3<ngl::Real>, mac_cells: int): void
Discretizer
#mac_cels: int
#mac_cells: std::vector<FlipParticle *>
#spatial_density: ngl::Real
<<create>>-Discretizer(m_mac_cels: int)
<<destroy>>-Discretizer()
+discretise(particles: std::vector<FlipParticle *>): void
+markSurface(A: Array::Array3<char>, density: ngl::Real): void
+findNeighbours_wall(x: int, y: int, z: int, in_width: int, in_height: int, in_depth: int, o_neighbours: std::vector<FlipParticle *>): void
+findNeighbours(x: ngl::Real, y: ngl::Real, z: ngl::Real, o_neighbours: std::vector<FlipParticle *>): void
+computeLevelset(i: int, j: int, k: int, density: ngl::Real): ngl::Real
Cache
+frame_num: int
<<create>>-Cache()
<<destroy>>-Cache()
+writeAnimation(m_working_dir: std::string, m_filename: std::string, access: Declarations, particles: std::vect
+writeHoudiniGeo(m_working_dir: std::string, temp_cache: std::vector < FlipParticle *>, m_name: std::string
Solver
+m_working_dir: std::string
+M_ID: int
+OFF_BOUNDS_FLAG: unsigned int
+m_framesWritten: int
+res_multiplier: int
+current_frame: int
+m_mac_cells: int
+m_torqueboost: ngl::Real
+m_total_sim_frames: ngl::Real
+m_max_density: ngl::Real
+m_mac_cell_scale: ngl::Real
+m_viscosity: ngl::Real
+m_timestep: ngl::Real
+m_density: ngl::Real
+m_gravity: ngl::Real
+grid_velocity: Array::Array3<ngl::Real>
+stored_grid_velocity: Array::Array3<ngl::Real>
+surface_marker: Array::Array3<char>
+levelset_marker: Array::Array3<ngl::Real>
+pressure: Array::Array3<ngl::Real>
+particles: std::vector<FlipParticle *>
+mac_grid_objs: std::vector<OutterWallObj>
RK4Integration
<<create>>-RK4Integration()
<<destroy>>-RK4Integration()
+acceleration(state: State): ngl::Vector
+evaluate(initial: State): Derivative
+evaluate(initial: State, dt: ngl::Real, d: Derivative): Derivative
+integrate(state: State, dt: ngl::Real): void
ConjugateGradientSolver
<<create>>-ConjugateGradientSolver()
<<destroy>>-ConjugateGradientSolver()
+solve(A: Array::Array3<char>, L: Array::Array3<ngl::Real>, x: Array::Array3<ngl::Real>, b: Array::Array3<ngl::Real>, n: int): void
+compute_Ax(A: Array::Array3<char>, L: Array::Array3<ngl::Real>, press: Array::Array3<ngl::Real>, ans: Array::Array3<ngl::Real>, n: int): void
+op(A: Array::Array3<char>, x: Array::Array3<ngl::Real>, y: Array::Array3<ngl::Real>, ans: Array::Array3<ngl::Real>, a: ngl::Real, n: int): void
+applyPreconditioner(z: Array::Array3<ngl::Real>, r: Array::Array3<ngl::Real>, P: Array::Array3<ngl::Real>, A: Array::Array3<char>, n: int): void
+buildPreconditioner(P: Array::Array3<ngl::Real>, L: Array::Array3<ngl::Real>, A: Array::Array3<char>, n: int): void
+conjGrad(A: Array::Array3<char>, P: Array::Array3<ngl::Real>, L: Array::Array3<ngl::Real>, press: Array::Array3<ngl::Real>, div: Array::Array3<ngl::Real>, N: int): void
+product(A: Array::Array3<char>, x: Array::Array3<ngl::Real>, y: Array::Array3<ngl::Real>, n: int): ngl::Real
+x_ref(A: Array::Array3<char>, L: Array::Array3<ngl::Real>, x: Array::Array3<ngl::Real>, fi: int, fj: int, fk: int, i: int, j: int, k: int, n: int): ngl::Real
+A_diag(A: Array::Array3<char>, L: Array::Array3<ngl::Real>, i: int, j: int, k: int, n: int): ngl::Real
+A_ref(A: Array::Array3<char>, i: int, j: int, k: int, qi: int, qj: int, qk: int, n: int): ngl::Real
+access
+access
#access
+access
-access
+kernel
+kernel
+mod
+map
+match

Figure 12 :Class Diagram 1 Shows the relationship of the heart of the pic flip the Solver class with the rest of the solver's classes functionality (part 1)

CLASS DIAGRAM PART 2

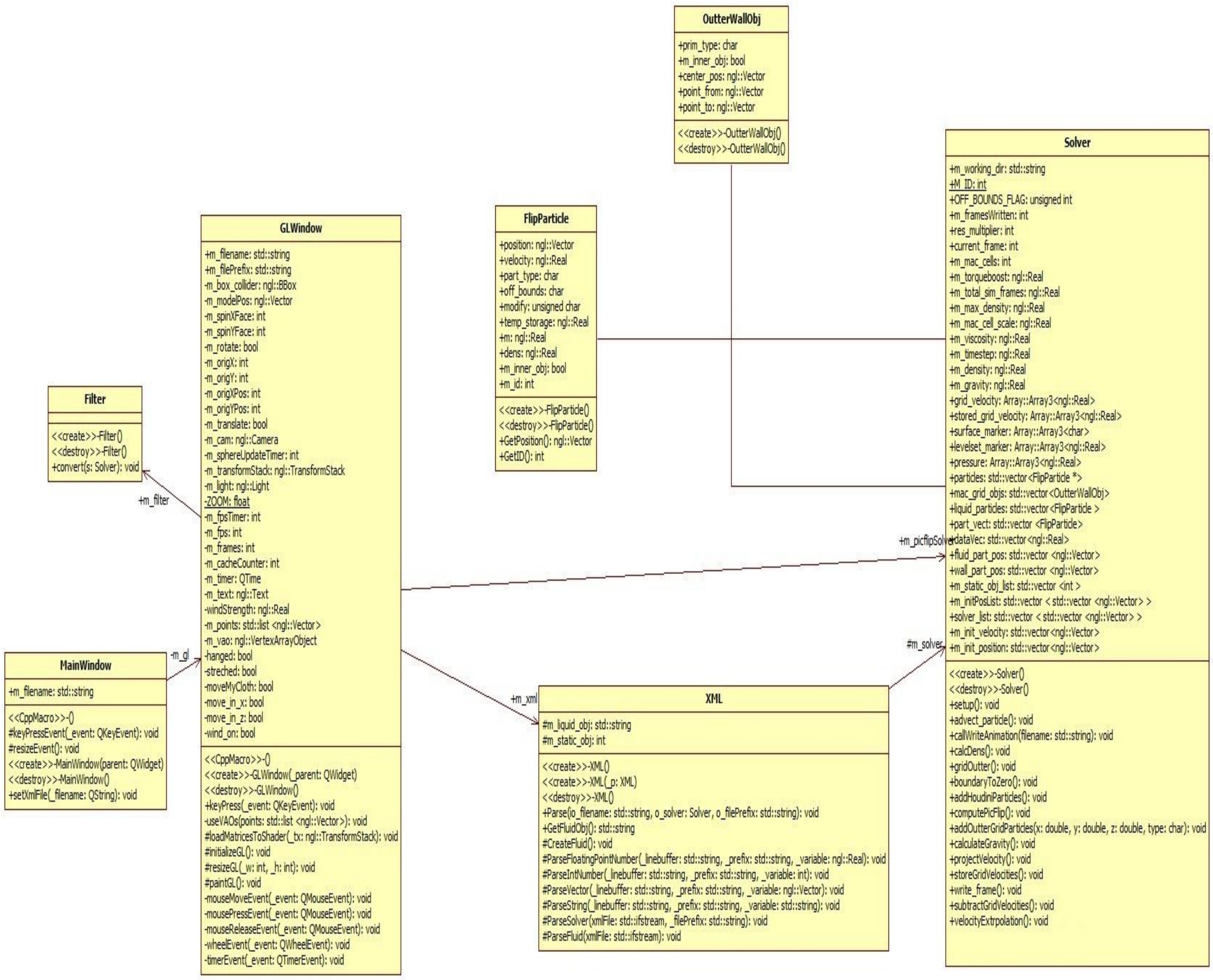

OutterWallObj
+prim_type: char
+m_inner_obj: bool
+center_pos: ngl::Vector
+point_from: ngl::Vector
+point_to: ngl::Vector
<<create>>-OutterWallObj()
<<destroy>>-OutterWallObj()
FlipParticle
+position: ngl::Vector
+velocity: ngl::Real
+part_type: char
+off_bounds: char
+modify: unsigned char
+temp_storage: ngl::Real
+m: ngl::Real
+dens: ngl::Real
+m_inner_obj: bool
+m_id: int
<<create>>-FlipParticle()
<<destroy>>-FlipParticle()
+GetPosition(): ngl::Vector
+GetID(): int
Solver
+m_working_dir: std::string
+M_ID: int
+OFF_BOUNDS_FLAG: unsigned int
+m_framesWritten: int
+res_multiplier: int
+current_frame: int
+m_mac_cells: int
+m_torqueboost: ngl::Real
+m_total_sim_frames: ngl::Real
+m_max_density: ngl::Real
+m_mac_cell_scale: ngl::Real
+m_viscosity: ngl::Real
+m_timestep: ngl::Real
+m_density: ngl::Real
+m_gravity: ngl::Real
+grid_velocity: Array::Array3<ngl::Real>
+stored_grid_velocity: Array::Array3<ngl::Real>
+surface_marker: Array::Array3<char>
+levelset_marker: Array::Array3<ngl::Real>
+pressure: Array::Array3<ngl::Real>
+particles: std::vector<FlipParticle *>
+mac_grid_objs: std::vector<OutterWallObj>
+liquid_particles: std::vector<FlipParticle >
+part_vect: std::vector <FlipParticle>
+dataVec: std::vector<ngl::Real>
+fluid_part_pos: std::vector <ngl::Vector>
+wall_part_pos: std::vector <ngl::Vector>
+m_static_obj_list: std::vector <int >
+m_initPosList: std::vector < std::vector <ngl::Vector> >
+solver_list: std::vector < std::vector <ngl::Vector> >
+m_init_velocity: std::vector<ngl::Vector>
+m_init_position: std::vector<ngl::Vector>
<<create>>-Solver()
<<destroy>>-Solver()
+setup(): void
+advect_particle(): void
+callWriteAnimation(filename: std::string): void
+calcDens(): void
+gridOutter(): void
+boundaryToZero(): void
+addHoudiniParticles(): void
+computePicFlip(): void
+addOutterGridParticles(x: double, y: double, z: double, type: char): void
+calculateGravity(): void
+projectVelocity(): void
+storeGridVelocities(): void
+write_frame(): void
+subtractGridVelocities(): void
+velocityExtrpolation(): void
GLWindow
+m_filename: std::string
+m_filePrefix: std::string
-m_box_collider: ngl::BBox
-m_modelPos: ngl::Vector
-m_spinXFace: int
-m_spinYFace: int
-m_rotate: bool
-m_origX: int
-m_origY: int
-m_origXPos: int
-m_origYPos: int
-m_translate: bool
-m_cam: ngl::Camera
-m_sphereUpdateTimer: int
-m_transformStack: ngl::TransformStack
-m_light: ngl::Light
-ZOOM: float
-m_fpsTimer: int
-m_fps: int
-m_frames: int
-m_cacheCounter: int
-m_timer: QTime
-m_text: ngl::Text
-windStrength: ngl::Real
-m_points: std::list <ngl::Vector>
-m_vao: ngl::VertexArrayObject
-hanged: bool
-streched: bool
-moveMyCloth: bool
-move_in_x: bool
-move_in_z: bool
-wind_on: bool
<<CppMacro>>-()
<<create>>-GLWindow(_parent: QWidget)
<<destroy>>-GLWindow()
+keyPress(_event: QKeyEvent): void
-useVAOs(points: std::list <ngl::Vector>): void
#loadMatricesToShader(_tx: ngl::TransformStack): void
#initializeGL(): void
#resizeGL(_w: int, _h: int): void
#paintGL(): void
-mouseMoveEvent(_event: QMouseEvent): void
-mousePressEvent(_event: QMouseEvent): void
-mouseReleaseEvent(_event: QMouseEvent): void
-wheelEvent(_event: QWheelEvent): void
-timerEvent(_event: QTimerEvent): void
Filter
<<create>>-Filter()
<<destroy>>-Filter()
+convert(s: Solver): void
+m_filter
MainWindow
+m_filename: std::string
<<CppMacro>>-()
#keyPressEvent(_event: QKeyEvent): void
#resizeEvent(): void
<<create>>-MainWindow(parent: QWidget)
<<destroy>>-MainWindow()
+setXmlFile(_filename: QString): void
-m_gl
+m_picflipSolver
#m_solver
+m_xml
XML
#m_liquid_obj: std::string
#m_static_obj: int
<<create>>-XML()
<<create>>-XML(_p: XML)
<<destroy>>-XML()
+Parse(io_filename: std::string, o_solver: Solver, o_filePrefix: std::string): void
+GetFluidObj(): std::string
#CreateFluid(): void
#ParseFloatingPointNumber(_linebuffer: std::string, _prefix: std::string, _variable: ngl::Real): void
#ParseIntNumber(_linebuffer: std::string, _prefix: std::string, _variable: int): void
#ParseVector(_linebuffer: std::string, _prefix: std::string, _variable: ngl::Vector): void
#ParseString(_linebuffer: std::string, _prefix: std::string, _variable: std::string): void
#ParseSolver(xmlFile: std::ifstream, _filePrefix: std::string): void
#ParseFluid(xmlFile: std::ifstream): void

Figure 12 : Class Diagram 2 Shows the relationship of the Solver with GLWindow class from which all the algorithmic steps of the simulation are called plus the XML class is used to import and read a file with all configuration settings and passes the values read to the solver. (part 2)

4.3 Brief Class Functionality Description

Accompanying the above diagrams, in this section a quick description of each class will be given:

XML
This is a class that contains the parser to an XML configuration file and sets up and initializes the custom fluid solver based on it.

Solver
This is a class that contains the main algorithm of the solver that guides all the needed functions for the simulation. It sets up the performs all the crucial steps of the custom solver needed such as setting up the staggered MAC grid, placing particles based on Houdini's scene particle objects, implementing the pic/flip main algorithm, handling the advection of the particles and calling the exporter of the particle point positions.

SmoothingKernels
This is a class that contains a series of smoothing kernels to be used in neighbouring particles' mutual contribution for operations like density evaluation, velocity resampling or transferring particle velocities to the MAC grid.

RK4Integration
This is a class that a implements a 4th order Runge-Kutta integration method. It is used as the ODE solver for the advection of particle positions in the last step of the solver's main algorithm.

OuterWallOBJ
This is a class that creates an MAC grid wall cubic object in which the simulation takes place. It defines all 6 sides of the MAC grid's cube and places it in the center of our scene's window to incorporate the simulation.

Modifier
This is a class that performs all sorts of all particle repositioning related operations . It calculates and reassigns an average of neighbouring particle velocities to a queried particle during velocity resampling process. It is also responsible for the position modification of particles to guarantee an evenly spaced distribution of the fluid volume while simulating.

MainWindow
This is the main re-sizable window which contains a GLWindow widget used to hold our basic openGL applications plus it's used to pass the xml settings configuration file to the Solver.

GridMapper
This is a class that performs all sorts transfers of velocities from particles to MAC grid and vice versa. It also calculates and stores linear grid velocities at X,Y,Z directions.

GLWindow
This is our main openGL window widget for NGL applications' drawing elements. Moreover, it's also the class in which the pic/flip algorithmic steps that are implemented in Solver are called one by one at each frame of the simulation.

FlipParticle
This is a class that represents an instance of a fluid particle of the whole fluid.
It contains the main attributes of a fluid particle instance such as its 3D position and velocity, its mass, its type (liquid or solid or empty), its density etc..

Filter
This is a class that performs filtering of the point positions of the particles that are imported from Houdini's scene to the custom solver's space.

Discretizer
This is a class that performs all sorts of discretization needed to identify each MAC grid cell. Furthermore, it performs the neighbour search of a queried particle but it also contains the functionality for setting indicators to each of the MAC grid's cells (solid, liquid, empty) plus a surface tracking method.

Declarations
This is a class that declares all global variables needed to identify the nature of the MAC grid's cells and particles (solid, liquid, empty).

Cache
This is a class that writes all particle positions to a .geo Houdini compliant file format file. It contains a method that prepares a list of liquid particles to be passed into writeHoudiniGeo method which is the actual exporter of particles' point positions.

ConjugateGradientSolver
This is a class that implements the Modified Incomplete Cholesky preconditioner and the Conjugate Gradient method for solving particles ' pressure on the MAC grid at each time step of the simulation.

4.4 Project Pipeline

As the initial idea of the project was to create a custom plugin for SideFX Houdini 3D Package, the pipeline was built around Houdini's environment by making use of Houdini's Object Model (HOM) plus the powerful capabilities of Houdini Digital Assets (HDAs). As more and more parts of Houdini are written in Python language, this offers great advantages such as :

- Scripting in digital assets
- Implementing nodes in Python
- Using Python in parameter expressions
- Creating user interfaces from Python

Considering the above, we created a two HDAs inside Houdini, one for each fluid to be loaded and one guiding and controlling the external custom C++ solver application.

4.4.1 Analytical Pipeline Structure Steps

1. Pre-creation of fluid objects (convert models to objects filled with particles and save them as .obj files)

2. Initialization of .obj Loading, creation and visualization of a fluid using HDA1 (picFlipLiquid) and Python scripting in its internal python module.

3. Adjustment of any of the fluid’s parameters

4. Addition of all created fluids to the solver plugin HDA2 (picFlipSolver)

5. Adjustment of any of the solver’s parameters

6. Writing all solver’s settings to an .xml settings file using Python scripting inside the picFlipSolver’s internal python module.

7. Trigger the external c++ solver and start the simulation

8. Parse the configuration .xml file and initialize the custom external solver based on it

9. Solve the simulation scene

10. Export data to a .geo houdini compliant file format at each frame of the simulation

11. Return to the picFlipSolver HDA inside Houdini and create a scene based on the .geo files written for all frames that have been simulated

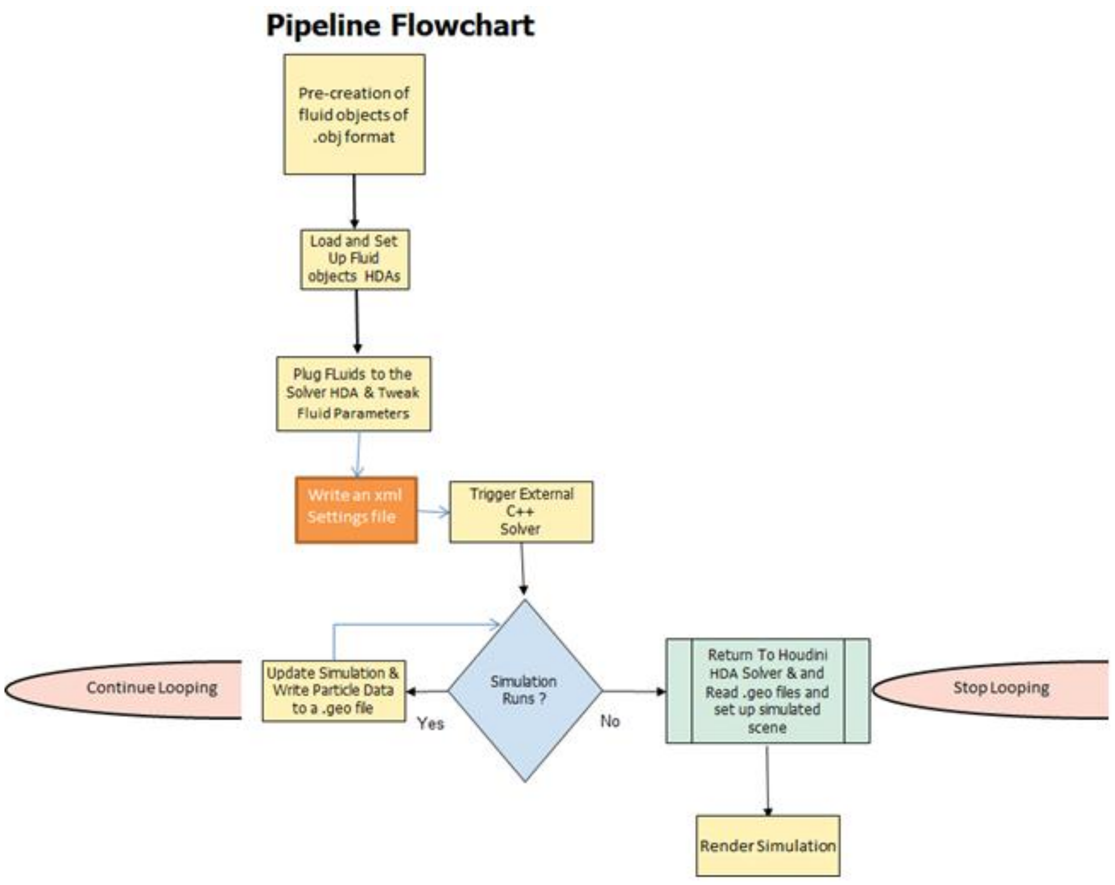

Pipeline Flowchart
Pre-creation of fluid objects of .obj format
Load and Set Up Fluid objects HDAs
Plug FLuids to the Solver HDA & Tweak Fluid Parameters
Write an xml Settings file
Trigger External C++ Solver
Simulation Runs ?
Yes
No
Update Simulation & Write Particle Data to a .geo file
Continue Looping
Return To Houdini HDA Solver & and Read .geo files and set up simulated scene
Stop Looping
Render Simulation

Figure 13 :This flowchart depicts all the important steps of the project's pipeline structure

4.5 Scene Object Representation

While initially designing the custom simulator an answer had to be given on how to define represent the fluid and static object entities in our custom simulation scene. Inspired by Priscott (2010) referring to Steele et. al (2004) approach, a particle representation of both was finally chosen. Having researched other approaches such as representing static objects with polygonal or parametric surfaces and computing fluid interactions with them, converting the static objects into objects filled with particles was offering convenience and an easy to code process (considering the time limitations of this project) as the main algorithm stays the same while adding an extra overhead to the simulation. This is something that could be improved in future work.

5 Implementation

In this section all the important parts of the actual implementation of the fluid solver are presented, analyzed and discussed in detail. The structure of this chapter is as follows :

1. Main Algorithmic Scheme (using Pseudo code)
2. Analysis of the most crucial functions encapsulated in each Class of the application
3. Detailed discussion of theoretical base of the methodologies applied.

5.1 Main Algorithmic Scheme

Intitialization Phase

```
• ReadXMLFile()
• foreach particle P houdiniParticlelist do
        // Convert To Solver Space
        • solverParticlelist = FilterParticlePosition(P)

•  SetUpSolver()
• ConstructMacGrid()
• foreach particle P solverParticlelist do
    • AddHoudiniParticle(P)
• foreach particle P particlelist do
    • MatchParticle To CorrespondingCell(P);
• computeFluidSurface()
• foreach particle P particlelist do
    • if( is placed in solid wall cell)
        ▪ InitiallStuckParticleRemoval()
```

Table 3 : Initialization Steps of the Algorithm

Looping Simulation Phase

```
• while(timer_ticks)
    o foreach particle P particlelist do
        ▪ MatchParticleToCorrespondingCell(P)
    o foreach particle P particlelist do
        ▪ calculateDensity(P)
    o foreach particle P particlelist do
        ▪ calculateGravityForce(P)

    o storeParticleVelocitiesToGrid()
 ///  temp store Current grid velocities
    o storeGridVelocities()
 /// solve pressure to grid and calculate new grid velocities
      o projectPressure()
      o newgridVel = calculateNewGridVelocities()
 /// set new boundary vellocities to 0
      o setBoundaryVelocitiesToZero()
 /// re evaluate grid velocities in regions that mightnot be defined
     ExtrapolateVelocity ()
    o writeSimulationFrame()
```

Table 4 : Operations while the simulation runs (1/2)

```
 /// flip
/// calculate new flip velocity
/// subtract current grid from saved grid velocities and store 'em
         o changeInVelocity = subtractGridVelocities()
///calculate new flip velocities based on change in grid's velocities
         o  foreach particle P particlelist do
               ▪  particleVel = storeGridVelocitiesToParticles
/// Interpolate the change in grid velocities to the particles existing    veloci-
ties

         o  foreach particle P particlelist do
               ▪  Flip_particleVel = particleVel + changeInVelocity

/// pic
/// calculate new pic velocity
      /// Interpolate the new grid velocity to the particles existing     veloci-
      ties
         o   foreach particle P particlelist do
               ▪   Pic_particleVel = particleVel + newgridVel;

///Interpolate pic flip velocities
         o  foreach particle P particlelist do
               ▪  particleVel = (1-visc_coefficient)* Pic_particleVel +
                  visc_coefficient *Flip_particleVel

///Advect Particles
         o  foreach particle P particlelist do
               ▪  integrate(P)

///Detect Collisions amongst Particles
         o  foreach particle P particlelist do
               ▪  collisionDetection() & resolve()

///Reposition Particles
         o  foreach particle P particlelist do
               ▪  if( P is liquid particle && is stucked in solid cell)
                     ●  repositionInsideTheFluid()
```

Table 4 : Iterative Operations while the simulation runs (2/2)

5.2 From Houdini To Custom Solver

At first, as in this chapter we are about to discuss the actual Solver's implementation and the issues dealing with it, we are going to start from this step of the pipeline where the Houdini custom plugin has given its turn to the external solver to solve the simulation. So, after the user has pressed the "Simulate" button of the plugin (see Chapter 6) the XML class of the solver has to parse the configuration data from the .xml settings file.

5.2.1 XML Configuration Functionality

XML parsing means nothing more than the XML class of the solver of the external C++ application reading line by line the configuration settings data that have been written previously and saved as a . xml in the solver's current directory. What this class offers is the initialization of the solver itself based on the values that have been read. A number of parameters that is highly important to be initialized before the solver starts such as the time step of the simulation, the number of the total frames, the prefix which will be used for naming the export of each .geo file at each frame so as not to overwrite the already simulated frames. Additionally, each simulated fluid's parameters such as position, viscosity, velocity are defined and initialized.
The class basically separates the .xml context based on the different tags in the file and performs the corresponding match to the proper variables of the solver.

5.2.2 Particle Fluids From Objects

In this chapter the process of creating and importing fluid objects from Houdini into the custom solver is presented. The fluid objects are pre-created inside Houdini using the "points from volume" SOP which basically fills a geometry node of any random shape with particles . After, the fluids have been created we save them as .obj files which are then loaded using NGL graphics library (Macey, 2011) obj class and more specifically the getNumVerts() method of AbstractMesh class which provides access to the total number of vertices in the object and that way objects are able to be imported into the scene as individual vertices. By matching these vertices to particles we construct the fluids inside the solver after having performed a position filtering for placing them correctly in our custom scene space.
One of the solver's primary objectives is to have the ability to load multiple fluids to interact within the scene and so a list of fluids is created each one with its list of particle positions and parameters.

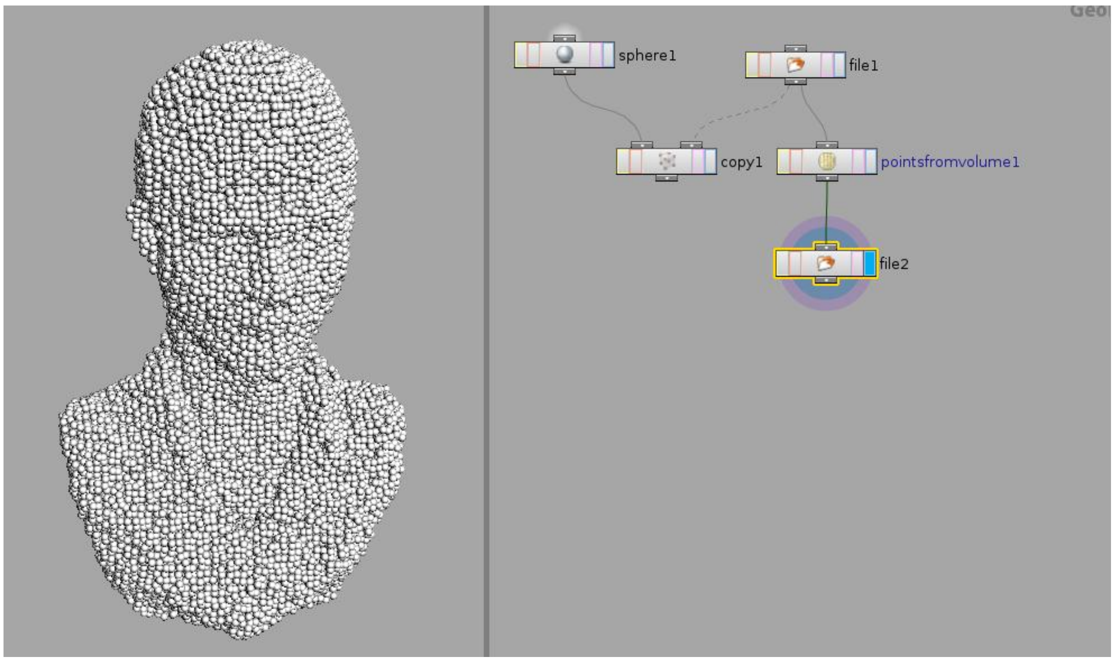


Figure 14 :This is the process of the creation of a "man figure" .obj to be loaded in a fluid HDA. We load a usual .obj model of a "man figure", we fill it with particles using the "pointsfromvolume" SOP and write this to a new .obj file.

ρ(x; t) density (kg/m3)
p(x; t) pressure (N/m2)
u(x; t) velocity (m/s)
f(x; t) body forces density(N/m3)

Table 5 : Fluid physical properties list in the simulation

5.2.3 Particle Position Filtering & Conversion

As this is a custom plugin meant to run exclusively from inside Houdini, the initial setup of the scene is prepared inside Houdini too. This means that, after having loaded a pre-created fluid object to the corresponding HDA and after having connected it to the HDA solver and pressed the "Simulate" button, the current particle positions of the fluid/s have to be converted from the Houdini's scene space to the custom fluid solver's space.
This functionality is handled by the Filter class of the solver and more specifically within the *convert()* method. A brief description of the method's functionality is presented below using pseudo code:

```
iterator = IterateHoudiniFluidsList()
For each fluid in iterator
  // Store the size of its current Houdini positions list
    fluidPositionsList = fluid.size()

  For each position p in fluidPositionsList
     //convert to MAC grid space
     newP = convertPosition(p)

     //store each fluid's converted position to a temp vector
     tempFluidPositionList.push_back(newP)

        //save a list with all fluids (each one with its own par-
        ticles and their corresponding positions)

// Store a list of its converted positions
        solver_list.push_back(tempFluidPositionList)

    empty_TempFluidPositionList()
```

Table 6 : Fluid list iteration and filtering conversion of each fluid's particle positions

Here we iterate over a list of Houdini fluids with their positions loaded through their .obj paths using the XML class and for each one of these fluids we loop through the list of its positions and convert each one to the MAC grid custom space by multiplying each position by a constant factor 0.025 (to ensure enough scale) and placing them at both sides of the 0.5 (custom solver origin position).

For example,

Houdini Scene particle position	Custom Solver particle position
(0,0,0)	(0.5+ **0***0.025), (0.5+ **0***0.025), (0.5+ **0***0.025)
(1,0,0)	(0.5+ **1***0.025), (0.5+ **0***0.025), (0.5+ **0***0.025)
(0,0,2)	(0.5+ **0***0.025), (0.5+ **0***0.025), (0.5+ **2***0.025)

Table 7 : Conversion function visualization from Houdini scene space to custom Solver scene space

Later, we store each converted position to a temporary vector which holds all the converted positions for the current fluid each time the iterator increases. When all the positions of the current fluid have been converted we add this vector of converted particle positions to a vector that is meant to hold a list of the fluids with their new converted positions locally inside the custom solver. Note that in each iteration, before we proceed in looping the list of particle positions of another fluid, the vector that contained the previous converted particle positions of the previous fluid must be emptied.

5.3 Simulation Set Up & Initialization

As soon as we have initialized the solver with the values specified from the HDAs ‘ parameters inside Houdini it’s time to set up the pic/flip solver. The most important steps are presented and analyzed in the flowing subsections.

5.3.1 Memory Allocation

The first thing we need to do is to allocate memory for all the variables related to the 3D MAC grid in which the simulation is taking place. Some examples of those variables are the “grid velocity”, “stored grid velocity”, “surface marker”, “levelset_marker”, “pressure”, “mac grid cells”. All the these variables have an immediate connection with the 3D MAC grid and so an appropriate 3D structure has to hold and maintain the data corresponding to each cell. For this thesis a high-performance multi-dimensional C++ array library by John C. Bowman (2010) was used to offer efficiency and consistency in terms of memory jumps while looping through each of these arrays during the simulation.
Secondly, we assign zero values to the all cell’s grid velocities and pressure and manually construct a 3D MAC grid with a specified number of 32 cells in each direction

(32x32x32) using c++ multidimensional arrays, as we noticed that this is a reasonable grid discretization value which offers a quite satisfying visual outcomes.

5.3.1.1 Multidimensional C++ Array Explanation

We can imagine a multidimensional c++ array as creating an array that points to other arrays, which point to other arrays, and so on. For example, to create a 2x3x4 array in C++, we should imagine the implementation as follows (The Code Log, 2010):

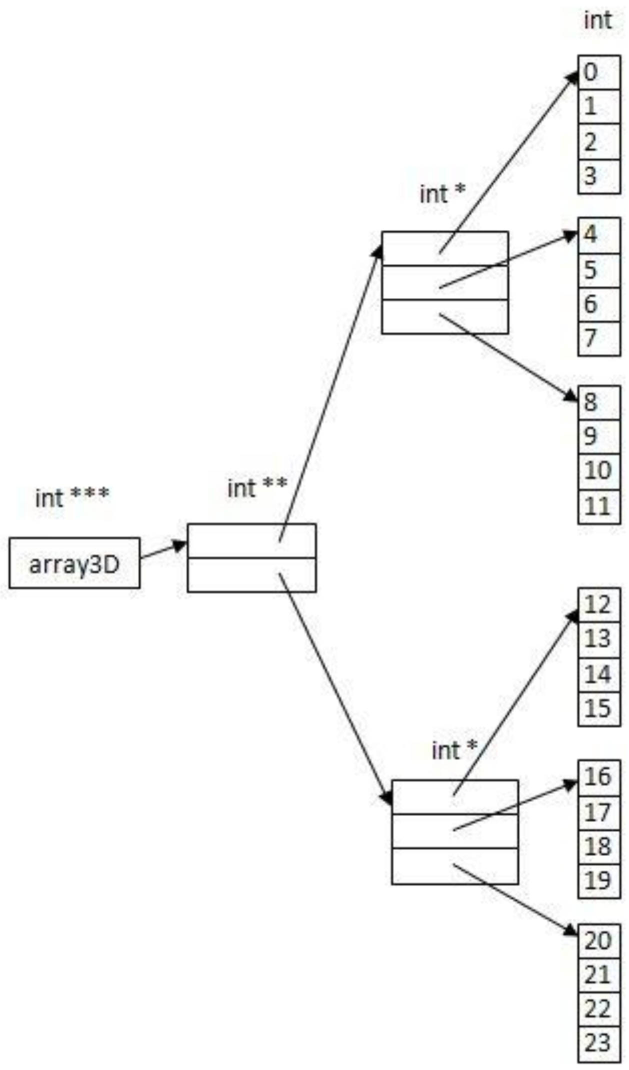


Figure 15 :This is a memory representation of the implementation of a multidimensional array
(by http://mycodelog.com/2010/05/21/array3d/)

5.3.2 MAC Grid Walls Creation

The next operation requires the construction of a 3D cube as sides of the MAC grid to represent the outer solid wall boundaries in which the simulation takes place. The 3D cube is constructed inside the solver by defining an OuterWallObj with sides from 0,0,0 to 1,1,1 as follows:

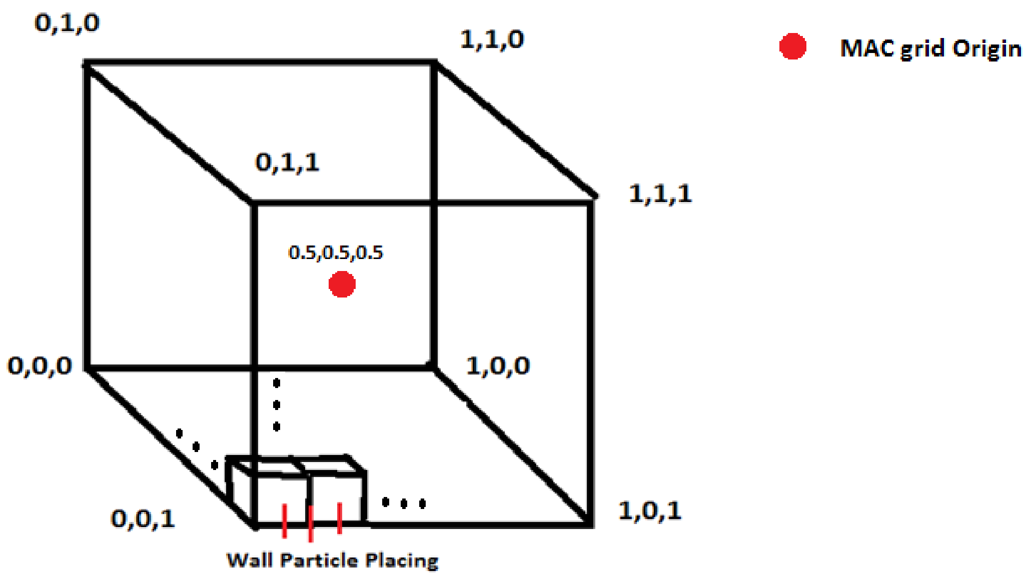


Figure 16 :This is cube representing the outer wall sides of the 3D MAC grid with indicators of the outer wall solid particle insertion during its construction.

We place the outer solid wall particles in each side of the 3d cube with a spacing of "mac_cell_scale /2" which represents the half width scale of each cell of the MAC grid and becomes the step for each next particle position to be placed.

mac_cell_scale = 1 / number of MAC grid cells

This way, we place a solid wall particle every half cell in each side of the grid in all 3 dimensions.

5.3.3 Fluid & Rigid Body Objects Creation and Placing

At this stage we are supposed to create and place correctly all Houdini's scene objects specified to be simulated and added to the custom picFlipSolver'sHDA inputs. To do this we are going to iterate over the pre-created vector that holds a list of the fluids with their new converted positions locally inside the custom solver.

(std::vector < std::vector <ngl::Vector> > solver_list).

For each of the these objects we check if it's specified as a fluid or as static collision object and mark it properly and create a FluidParticle instance for each of its particles. Additionally, we define initial position and velocity based on the corresponding pre-created vectors holding theses values specified through the picFlipSolver HDA inside Houdini. At last, we add the created particle to a generic vector holding all simulation particles.

5.3.4 Particle-Cell Identification

In this step of initialization, we basically match each placed particle's position in a cell of the 3D grid. This is a quite simple process and it's described algorithmically below:

- Loop through the list of all simulation particles
- Multiply each particle's position with the number of cells in each direction and acquire its cell index . For example in 1D when cells are 32 we have (ex. 0.5 * 32 = 16).The number 16 is the cell index which currently the particle with position 0.5 belongs to.
- Store the particle in the relevant corresponding cell based on its index.

5.3.5 Fluid Surface Marking

An essential step for this fluid simulation using the pic / flip method is to set markers in the cells of the grid so as to indicate whether it represents a liquid, a solid or an empty cell filled with atmospheric air. This information highly important and is used in many steps during the simulation algorithm such as setting boundary velocity to zero, the pressure projection step, velocity extrapolation and so on.
The actual process of marking cells can be analyzed in the following steps:

```
• foreach cell in mac_grid_cells do
   • fillWithAir(cell)
   • foreach particle in cell do
        ▪ if(particle == solid)
             • setGridMarker(solid)
          else
             if(levelset < 0 )
                 o setGridMarker(liquid)
               else
                 o setGridMarker(empty)
```

Table 8 : The MAC grid cells' marking process.
Each cell is checked and marked as a liquid, solid or empty based on a its level set value.

5.3.5.1 Level Set Function Analysis

Existing Methods Brief Description

A Level Set method generally is an implicit or an explicit technique which is used to describe the fluid surface and more specifically the interface motion between a fluid (ex. water) and another one (ex. air). There has been a variety of level set methods investigated over the past twenty years. Implicit level set methods, introduced by Osher and Sethian, basically rely on an implicit representation of the interface whose equation of motion is numerically approximated and built from hyperbolic conservation laws (Sethian ,2003). More specifically, at each cell of the staggered mac grid a signed distance function from the closest surface position is applied in the form of *φ(x) =y. When, φ(x) =0* means we are at the fluid's surface whereas if it's *φ(x) <0* means we 're inside the fluid otherwise if *φ(x) >0* we 're referring to an air region outside the fluid volume.

Furthermore, there is a variety of explicit approaches researched concerning fluid surface tracking with the most exceptional one to be the *"volume-of-fluid* method" (Hirt, 1981) in which the underlying domain is discretized and cell fractions are filled with values that represent the characteristic function in those cells; these values are zero or one except in those cells cut by the interface(Sethian, 2003). For example, Consider the function $\chi(x, y, t)$, where $\chi=1$ inside the interface **Γ** and $\chi=0$ anywhere else. So the the motion of χ could be written as,

$$\chi_t = -\mathbf{u} \cdot \nabla \chi \qquad (5.1)$$

In this view, all the points inside the set (where the function $\chi=1$) are transported under the velocity field.

Both implicit and explicit methods have several drawbacks. Implicit methods cannot handle ruptures in points of the fluid where the surface is thin and suffer memory inefficiency problems. Volume-of fluid explicit method also faces difficulties for example in handling a discontinuous interface.

We used

In this thesis we adopted an alternative approach by (Ando,2012) which calculates a level set function of the form:

$$\phi_{\mathrm{L}}(x) = \alpha_L N_0 \rho_0 - \sum_i \rho(p_i) \qquad (5.2)$$

i : current particle cell index

ρ : the initial max density value specified at the start of the simulation

α : a parameter distinguishing fluid volumes from fluid splashes (rate of tolerance of splashes)

N : the initial number of particles placed in each cell

The fluid volume is described by all cells of the grid in which $\varphi <=0$.
Parameter $\alpha=0.2$ and $N=8$, $2x2x2$ particles approximately for 3 dimensional fluids. With the use of this method, one advantage we do get is that in regions of the fluid volume that are sparse or even in places where gaps may exist are not being marked as fluid cells. This leads to the avoidance of the strange visual artifact of lonely particles floating onto the fluid surface in sparse regions and it also prevents the unnecessary increase of fluid volume (Ando, 2012).

5.3.6 Initial Stuck Particle Removal

This is the last step of the initialization stage were all fluid particles that have been placed in positions where their corresponding cell indexes were marked as solid wall cells at the previous (fluid surface marking) step are removed once and for all from the scene. This, prevents liquids to be stuck inside static obstacles and RBD objects or exist outside the borders of the outer MAC grid walls at the initial placing stage of all particles to the custom solver's scene space.

5.4 Iterative Simulation Algorithm

At this section, all the needed operations and the concepts developed around them which are essential to be applied at each time step of the simulation are going to be presented analytically.

5.4.1 Particle in Cell Discretization

This step is analyzed at section 5.3.4. It should be called iteratively during the simulation and be the first thing to do after a complete simulation cycle to match all the particles to their corresponding cells.

5.4.2 Density Calculation

In order to specify the local density on the simulation grid which is essential at each time step of the simulation (see level set computation 5.3.5.1), we have to be able to calculate and determine the molecules' (particles) concentration from the particle data. For that, we collect and sum the particle density values at each grid cell modulated by a kernel function *W* that is used to produce a weighted average contribution of nearby particles (Bridson, 2008).
More analytically, for each queried particle of the fluid we search and find neighbours around its cell index on the grid, we calculate the weighted contribution of each of the neighbour particles found based on its mass and distance from the queried particle moderated by a smoothing kernel (Kelager, 2006) and finally we calcu-

late the queried particle’s density by taking the ratio of the sum of all neighbours’ total weighted contribution over the max density value for the fluid (Turk, 2010) :

$$\rho(x) = \sum_i m_i \, W(p_i - x, h) \;/\; \rho_{max} \qquad (5.3)$$

where ,

- $\boldsymbol{\rho}$, the particle density
- $\boldsymbol{m_i}$, the mass of a neighbour particle
- $\boldsymbol{p_i}$, the position of a neighbour particle
- $\boldsymbol{x}$, the queried particle’s position
- $\boldsymbol{\rho_{max}}$,, the maximum density value of the fluid
- $\boldsymbol{h}$, the radius of contribution of a neighbour particle
- $\boldsymbol{W}$, a loose kernel function used

5.4.3 Solve External Forces

The only physical external force that takes place in the simulation and acts on the fluid is the gravity force $\boldsymbol{g}$. To compute the gravity force we use Newton’s 2nd Law of motion $\boldsymbol{F} = \boldsymbol{m}\,\boldsymbol{a}$. The general gravity equation for velocity with respect to time is derived as follows :

We know,

$$F = ma \qquad (5.4)$$

$$F = mg \qquad (5.5)$$

and from equations 5.4 and 5.5 we get:

$$ma = mg \Rightarrow m\frac{du}{dt} = ma \Rightarrow du = dt\,a \Rightarrow v - vi = dt\,a \qquad (5.6)$$

so,

$$\boldsymbol{v} = \boldsymbol{g}\,\boldsymbol{dt} + \boldsymbol{vi} \qquad (5.7)$$

Since the initial velocity **v_i =** 0 for an object that is simply falling, the equation reduces to:

$$v = g\,dt \qquad (5.8)$$

where,

v , the vertical velocity of the object in meters/second (m/s)

- **g** is the acceleration due to gravity (9.81 m/s^2)
- **dt** is the time in seconds that the object has fallen (Ron Kurtus, 2011)

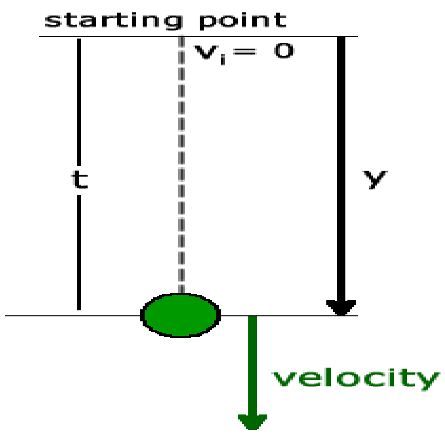


Figure 17 : Velocity of an falling object (with respect to time)

5.4.4 Particle Velocities to Grid

Considering that at each time step during the simulation at each staggered MAC grid cell we need to compute a weighted average of the nearby particle velocities so as to assign it the corresponding cell's face center, at this stage we need to define the method to transfer our particle velocities to the grid (shown in figure 10).
For each of the MAC grid cells we compute the weighted average of neighbour particles' contribution based on the their mass, their distance from the current evaluated cell using a kernel function of a specified radius, as we previously did for the density calculation step. We perform a standard trilinear interpolation so as to find the weight of each neighbour particle to a grid cell as shown below:

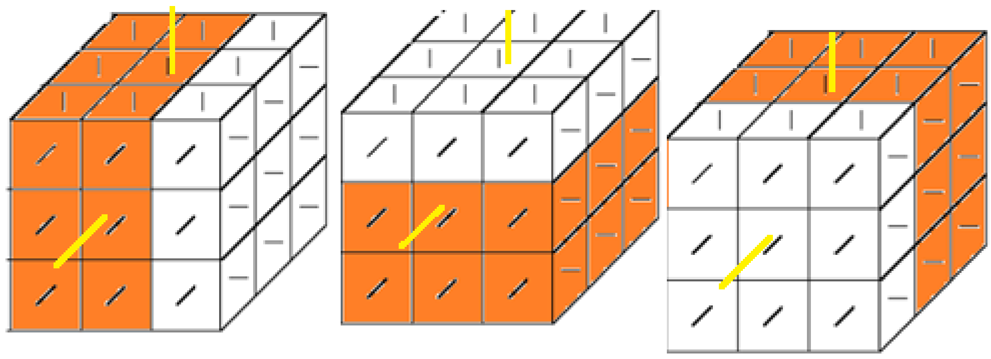


Figure 18 : Trilinear Interpolation of each neighbour particle of the center cell for the computation of its weighted contribution to this cell.
(Assume that each neighbour particle's weight is computed for cell 1,1,1 - middle cell in x, y, z dimensions in the center of the cube)

We use equation 5.9 to compute grid velocities transferred from particles for each X,Y,Z dimension:

$$u_{new}(\mathrm{p}) = \sum_i m_i \boldsymbol{u}_i W(p_i - x, h) \; / \sum_i m_i \; W(p_i - x, h) \qquad (5.9)$$

where ,

- $\boldsymbol{m_i}$, the mass of a neighbour particle
- $\boldsymbol{u_i}$, the new velocity of a neighbour particle
- $\boldsymbol{p_i}$, the position of a neighbour particle
- $\boldsymbol{x}$, the grid cell position
- $\boldsymbol{h}$, the radius of contribution of neighbour particle
- **W**, a stiff kernel function used

5.4.5 Boundary Velocities To Zero

At this stage, we need to enforce boundary velocities to zero. What this practically means is that in every part of the grid that has been marked as a solid wall cell we set grid velocity to zero. A solid wall boundary is where the fluid is in contact with a solid wall. If we are to determine that condition in terms of grid velocity we must ensure that the fluid will not be flowing into the solid or out of it, thus the normal component of velocity has to be zero:

$$u \cdot n = 0 \qquad (5.10)$$

where,

- **u,** the grid velocity
- **n,** the normal at this point

As we said since a solid wall object is not moving we should have zero velocity at this grid point as:

$$u\ solid = 0 \qquad (5.11)$$

At this point, the normal component of the fluid's velocity should match the normal component of the solid's velocity as:

$$u\ fluid = usolid -> u\ fluid\ = 0 \qquad (5.12)$$

5.4.6 Projection Step Algorithm & Theory

This is another crucial step of the simulation algorithm where we enforce the fluid to be incompressible. More analytically, at each time step we compute an intermediate velocity that tries to satisfy an incompressibility condition, so while it doesn't the current pressure is used as the means to project this intermediate velocity onto a divergence-free velocity field to get the next calculated of velocity and pressure.

5.4.6.1 Projection Method Main Algorithm

• Calculate the divergence of velocity b
• Build the Modified Incomplete Cholesky (MIC) preconditioner.
• Solve Ax = b linear system of equations with using the Preconditioned Conjugate Gradient iterative method (MICCG)
• Calculate new MAC grid velocities according to the pressure gradient update to current existing grid velocities.

The following section go into deep mathematical explanation of the concepts around the iterative "preconditioned conjugate gradient" method so as to find solve pressure that one could skip if not interested in mathematics background.

As we previously mentioned in chapter 3, a fluid needs to be incompressible because compressible fluids are in general expensive and more complicated to simulate. For an incompressible fluid we basically need that its volume does not change throughout the simulation.
A vector-field that satisfies the incompressibility condition is called "divergence-free" and to simulate incompressible fluids we need to assure that the velocity field stays divergence-free.

$$\nabla \cdot u = 0 \tag{5.13}$$

This is where the pressure comes in. We can imagine pressure as the medium that is needed to keep the velocity divergence-free (Bridson, 2008).
Let's derive exactly what the pressure has to be.
Taking the divergence of both sides of the momentum equation we have:

$$\nabla \cdot \frac{\partial \vec{u}}{\partial t} + \nabla \cdot (\vec{u} \cdot \nabla \vec{u}) + \nabla \cdot \frac{1}{\rho} \nabla p = \nabla \cdot (\vec{g} + \nu \nabla \cdot \nabla \vec{u}) \tag{5.14}$$

By changing the order of differentiation in the first term, we get the time derivative of divergence:

$$\frac{\theta}{\theta \tau} \nabla \mathrm{u} \tag{5.15}$$

And for incompressible fluids :

$$\frac{\theta}{\theta \tau} \nabla \mathrm{u} = 0 \tag{5.16}$$

So by rearranging (5.14) we have:

$$\nabla \cdot \frac{1}{\rho} \nabla p = \nabla \cdot (-\vec{u} \cdot \nabla \vec{u} + \vec{g} + \nu \nabla \cdot \nabla \vec{u}) \tag{5.17}$$

From this last Poisson type equation of pressure we can derive and measure how fast a fluid volume is changing.

5.4.6.2 Analysis of the Projection Step:

Compute the divergence of velocity
At first, we need to calculate "how much" the grid velocity is changing between two continuous grid cells

3D Divergence of Grid Velocity:

$$\nabla \cdot \vec{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} \tag{5.18}$$

So we need to basically evaluate only the divergence of velocity of grid cells marked as fluid only. And so the divergence for each fluid grid cell can be calculated as:

$$(\nabla \cdot \vec{u})_{i,j,k} \approx \frac{u_{i+1/2,j,k} - u_{i-1/2,j,k}}{\Delta x} + \frac{v_{i,j+1/2,k} - v_{i,j-1/2,k}}{\Delta x} + \frac{w_{i,j,k+1/2} - w_{i,j,k-1/2}}{\Delta x} \qquad (5.19)$$

The Discrete Pressure Gradient

Now, we need to have the mix pressure term somehow to derive the new grid velocities and that is possible using the pressure update formula below that uses central difference approximations:

$$\begin{aligned} u^{n+1}_{i+1/2,j,k} &= u_{i+1/2,j,k} - \Delta t \frac{1}{\rho} \frac{p_{i+1,j,k} - p_{i,j,k}}{\Delta x} \\ v^{n+1}_{i,j+1/2,k} &= v_{i,j+1/2,k} - \Delta t \frac{1}{\rho} \frac{p_{i,j+1,k} - p_{i,j,k}}{\Delta x} \\ w^{n+1}_{i,j,k+1/2} &= w_{i,j,k+1/2} - \Delta t \frac{1}{\rho} \frac{p_{i,j,k+1} - p_{i,j,k}}{\Delta x} \end{aligned} \qquad (5.20)$$

To explain this a bit further, consider the MAC grid now, where we need to subtract the **∂/∂x**-component of the pressure gradient from the **u** component of velocity. Note that there are two pressure values lined up perfectly on either side of the **u** component just waiting to be differenced.

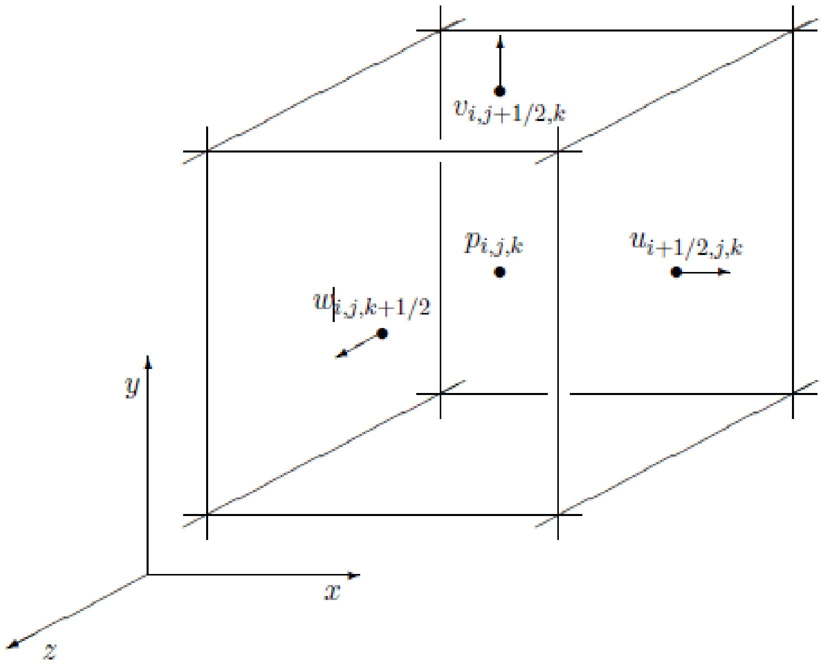


Figure 19 (reference figure 6) : 3D MAC grid

Now, all we need to do is to find the pressure that achieves incompressibility by substituting the pressure update formula (equation 5.20) into the 3D divergence formula (equation 5.19) (Bridson,2006).

Pressure Equations

Detailed Analysis of substitution of pressure update formula (5.20) to divergence-free formula (5.19) for a fluid grid cell (i,j,k) 3 dimensions:

$$\frac{u^{n+1}_{i+1/2,j,k} - u^{n+1}_{i-1/2,j,k}}{\Delta x} + \frac{v^{n+1}_{i,j+1/2,k} - v^{n+1}_{i,j-1/2,k}}{\Delta x} + \frac{w^{n+1}_{i,j,k+1/2} - w^{n+1}_{i,j,k-1/2}}{\Delta x} = 0 \qquad (5.21)$$

From which we get:

$$\frac{1}{\Delta x}\Bigg[\left(u_{i+1/2,j,k}-\Delta t\frac{1}{\rho}\frac{p_{i+1,j,k}-p_{i,j,k}}{\Delta x}\right)-\left(u_{i-1/2,j,k}-\Delta t\frac{1}{\rho}\frac{p_{i,j,k}-p_{i-1,j,k}}{\Delta x}\right)$$
$$+\left(v_{i,j+1/2,k}-\Delta t\frac{1}{\rho}\frac{p_{i,j+1,k}-p_{i,j,k}}{\Delta x}\right)-\left(v_{i,j-1/2,k}-\Delta t\frac{1}{\rho}\frac{p_{i,j,k}-p_{i,j-1,k}}{\Delta x}\right)$$
$$+\left(w_{i,j,k+1/2}-\Delta t\frac{1}{\rho}\frac{p_{i,j,k+1}-p_{i,j,k}}{\Delta x}\right)-\left(w_{i,j,k-1/2}-\Delta t\frac{1}{\rho}\frac{p_{i,j,k}-p_{i,j,k-1}}{\Delta x}\right)\Bigg]=0 \tag{5.22}$$

From which we finally derive:

$$\frac{\Delta t}{\rho}\left(\frac{6p_{i,j,k}-p_{i+1,j,k}-p_{i,j+1,k}-p_{i,j,k+1}-p_{i-1,j,k}-p_{i,j-1,k}-p_{i,j,k-1}}{\Delta x^2}\right)=-\left(\frac{u_{i+1/2,j,k}-u_{i-1/2,j,k}}{\Delta x}+\frac{v_{i,j+1/2,k}-v_{i,j-1/2,k}}{\Delta x}+\frac{w_{i,j,k+1/2}-w_{i,j,k-1/2}}{\Delta x}\right) \tag{5.23}$$

Finally in 5.23, we result into the numerical approximations of the Poisson problem described in a generic form below:

$$-\Delta t/\rho\nabla \cdot \nabla p = -\nabla \cdot u \tag{5.24}$$

5.4.6.3 Preconditioned Conjugate Gradient Solver

In the previous section we resulted in a large sparse system of linear equations. And this kind of systems can be solved using the Conjugate Gradient algorithm (CG) method and makes use of a preconditioner (PCG). In this section we will go into details of what this method is and what it offers.

5.4.6.3.1 Conjugate Gradient Method

The conjugate gradient method is method which numerically solves linear systems of equations (usually partial differential equations), whose matrix is symmetric and definite. This means that for a matrix A where,

- **A** is symmetric when $\mathbf{A}^T = \mathbf{A}$),
- **A** is positive definite when $\mathbf{x}^T\mathbf{A}\mathbf{x} > 0$ for all non-zero vectors **x** in $\mathbf{R}^n$), and real

  x is supposed to be the unique solution of the system.

Symmetric matrix

$$A = \begin{pmatrix} 1 & -3 & 7 \\ -3 & 5 & 4 \\ 7 & 4 & 3 \end{pmatrix}$$

Figure 20 : An example of a symmetric matrix.
The diagonal stays the same if we interchange rows with columns
(by http://metodososcaruis.blogspot.co.uk/2010/07/matrices.html)

The conjugate gradient method could be used either as a direct or as an iterative method. However when we are dealing large systems the direct approach is too expensive computationally and so we choose the iterative approach which performs better at these kind of systems (Wikipedia, 2012).
Algorithmically, in its simplest form the Conjugate Gradient could be expressed as:

```
guess at the solution x
      for i in iterations
     if(x==accurate_solution)
         stop
     else
         improve the existing solution x,
```

Table 9 : Generic explanation of the Conjugate Gradient method

At each iteration the method minimizes a particular measure of the error, and thus can be guaranteed to converge to the solution eventually (Muller, 2006)

General System Description

We can describe the whole system as:

$$A\,x\; =\; b \qquad (5.25)$$

where,

- **A**, is the coefficient matrix
- **x**, is a vector containing all the unknown pressures of cells that we need to find out
- **b**, is a vector of all divergences in each fluid cell

Each row of the A matrix corresponds to one equation for each fluid cell of the grid. The terms in each row of A represent the pressure coefficients in each equation and most of them are zero except for the those who correspond to this queried cell and its 6 neighbour cells (in 3D).

The **x** solution could also be describe as:

$$\mathbf{x}_* = \sum_{i=1}^{n} \alpha_i \mathbf{p}_i \qquad (5.26)$$

where **p**, is a sequence of mutually conjugate directions

(2 vectors u,v are conjugate with respect to A if $\mathbf{u^T}$***Av =0****. If* ***A*** *symmetric, positive definite then:*

$$\langle \mathbf{u}, \mathbf{v} \rangle_{\mathbf{A}} := \langle \mathbf{A}\mathbf{u}, \mathbf{v} \rangle = \langle \mathbf{u}, \mathbf{A}^{\mathrm{T}}\mathbf{v} \rangle = \langle \mathbf{u}, \mathbf{A}\mathbf{v} \rangle = \mathbf{u}^{\mathrm{T}}\mathbf{A}\mathbf{v}.$$

So, u, v are conjugate if they are orthogonal to this inner product and so ***U****conjugate****V*** *==* ***V****conjugate****U*** *)*

So, pressure coefficients $\boldsymbol{\alpha}_k$ could be resulted from:

$$\mathbf{b} = \mathbf{A}\mathbf{x}_* = \sum_{i=1}^{n} \alpha_i \mathbf{A}\mathbf{p}_i. \qquad (5.27)$$

In the iterative CG method, we only need to carefully specify $\mathbf{p}_k$ which might not give a satisfying approximation to the final **x** solution. However, this doesn't matter that much cause we get to improve this in the next iteration hence we solve the whole large linear system this way.

Useful Known Facts:

- **X** is also the unique minimizer of ***f*** :
$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} - \mathbf{x}^{\mathrm{T}}\mathbf{b}, \quad \mathbf{x} \in \mathbf{R}^n.$$
- if ***f*** **(x)** is smaller at each iteration means we are getting closer to the solution **x**
- $\mathbf{r}_k$ is the residual at the **k** th step (the error result of the iteration) and signifies "how much" divergence is left after we have updated pressure with our pressure estimate each time
- $\mathbf{r}_k$ is also the negative gradient of ***f*** at **x** = $\mathbf{x}_k$, which means that the method moves towards $\mathbf{r}_k$
- Initially we consider, $\mathbf{x}_0 = \mathbf{0}$
- $$\mathbf{p}_{k+1} = \mathbf{r}_k - \sum_{i \le k} \frac{\mathbf{p}_i^{\mathrm{T}}\mathbf{A}\mathbf{r}_k}{\mathbf{p}_i^{\mathrm{T}}\mathbf{A}\mathbf{p}_i}\mathbf{p}_i$$
- $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_{k+1}\mathbf{p}_{k+1}$ is the next optimal location
- $$\alpha_{k+1} = \frac{\mathbf{p}_{k+1}^{\mathrm{T}}\mathbf{b}}{\mathbf{p}_{k+1}^{\mathrm{T}}\mathbf{A}\mathbf{p}_{k+1}} = \frac{\mathbf{p}_{k+1}^{\mathrm{T}}(\mathbf{r}_k + \mathbf{A}\mathbf{x}_k)}{\mathbf{p}_{k+1}^{\mathrm{T}}\mathbf{A}\mathbf{p}_{k+1}} = \frac{\mathbf{p}_{k+1}^{\mathrm{T}}\mathbf{r}_k}{\mathbf{p}_{k+1}^{\mathrm{T}}\mathbf{A}\mathbf{p}_{k+1}},$$

Table 10 : Prerequisites of the Conjugate Gradient method

The Detailed Algorithm:

If a system $Ax = b$ where A is a real, symmetric and positive-definite matrix. The input vector X_0 can be an approximate to the initial solution or 0.

```
//initial error
    $\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$
//conjugate direction
    $\mathbf{p}_0 := \mathbf{r}_0$
//iteration number
    $k := 0$
repeat
    //pressure coefficient found
        $\alpha_k := \frac{\mathbf{r}_k^\mathrm{T}\mathbf{r}_k}{\mathbf{p}_k^\mathrm{T}\mathbf{A}\mathbf{p}_k}$
    //new solution of intermediate velocity
        $\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k\mathbf{p}_k$
    //new error calculated
        $\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k\mathbf{A}\mathbf{p}_k$
      if $r_{k+1}$ is sufficiently small then exit loop end if
    //new pressure coefficient
        $\beta_k := \frac{\mathbf{r}_{k+1}^\mathrm{T}\mathbf{r}_{k+1}}{\mathbf{r}_k^\mathrm{T}\mathbf{r}_k}$
    //new pressure calculation
        $\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k\mathbf{p}_k$
    //next iteration number
        $k := k + 1$
end repeat
```

Table 11 : The Conjugate Gradient Math Algorithm

The new solution result is x_{k+1} .

5.4.6.3.2 Using a Preconditioner

As we previously mentioned, the CG method can be computationally expensive in large systems and this because the number of iterations it takes to finally converge to the desired solution is proportional to the number of grid cells. This is where preconditioning comes in place!

The usual conjugate gradient takes more iterations the further away matrix A is from being the identity matrix, I. So, solving a system with the identity matrix is as :

$$I\,x = b \rightarrow x = b \qquad (5.28)$$

By using a preconditioner we basically imply that: that the solution of $\boldsymbol{Ax} = \boldsymbol{b}$ is the same as the solution of $\boldsymbol{MAx} = \boldsymbol{M}\,\boldsymbol{b}$ for some matrix **M**.
If M is approximately the inverse of A, so that MA get really close to being the identity matrix, this would mean that the could solve $\boldsymbol{MAx} = \boldsymbol{M}$ d really fast (Bridson, 2006).

Preconditioning Theory and Implementation

In general, we can solve linear systems of equations by doing row reductions in our system until the matrix is upper triangular or lower triangular:

$$\begin{bmatrix} 1 & 4 & 2 \\ 0 & 3 & 4 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 21 : Example Upper – Right Triangular Matrix

and then use backward substitution or forward correspondingly to get the solution.

LU Decomposition and Cholesky Method

Knowing that matrix equations with **triangular matrices** are easier to solve, we perform the LU decomposition which is nothing but expressing the A matrix as the product of its lower and its upper triangular matrix as follows:

$$A = LU \qquad (5.29)$$

- **L**, is the lower-left triangular matrix
- **U**, is the upper-right triangular matrix

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}.$$

Figure 22 : Example LU decomposition of a matrix ($A = LU$), where L has only zeros above its diagonal and U has only zeros below its diagonal

Now, since A is a positive and definite matrix we could write its "**Cholesky factorization"** as:

$$A = LL^T \qquad (5.30)$$

And so,

$$Ax = b \rightarrow L\,(L^T\, x) = b \qquad (5.31)$$

Now we can solve as:

$Lq = b$ with forward substitution (5.32)

$L^T x = q$ with backward substitution (5.33)

Incomplete Cholesky Method

Because, this solution is not particularly good memory wise for a 3 dimensional problem (A has a low number of zeros whereas L may have a lot) we move on to the **"Incomplete Cholesky factorization method".** This method tackles this specific problem by canceling all simple Cholesky's trials to add nonzeros to L at the positions where A currently has zeros. Eventually, we result with an L vector that is a sparse a A (so we guarantee memory) but we have caused errors on purpose so now:

$$A \neq LL^T \qquad (5.34)$$

This practically means that the lower-left triangular matrix of L is the same with A's lower-left triangular matrix in all nonzero positions of A and different in all zero positions of A . What we have achieved though is that L is very close to A so we can solve the sytem of equations almost as fast as if we had applied A^{-1} as a preconditioner.

Now, if we split A in its lower-left matrix if F its lower-left matrix and D its diagonal.

$$A = F + D + F^T \tag{5.35}$$

It can be shown that:

$$L = F E^{-1} + E \tag{5.36}$$

where,

E is a diagonal matrix.

So now, we only have to calculate the diagonal entries of E and we can just infer others from A (Muller, 2006).

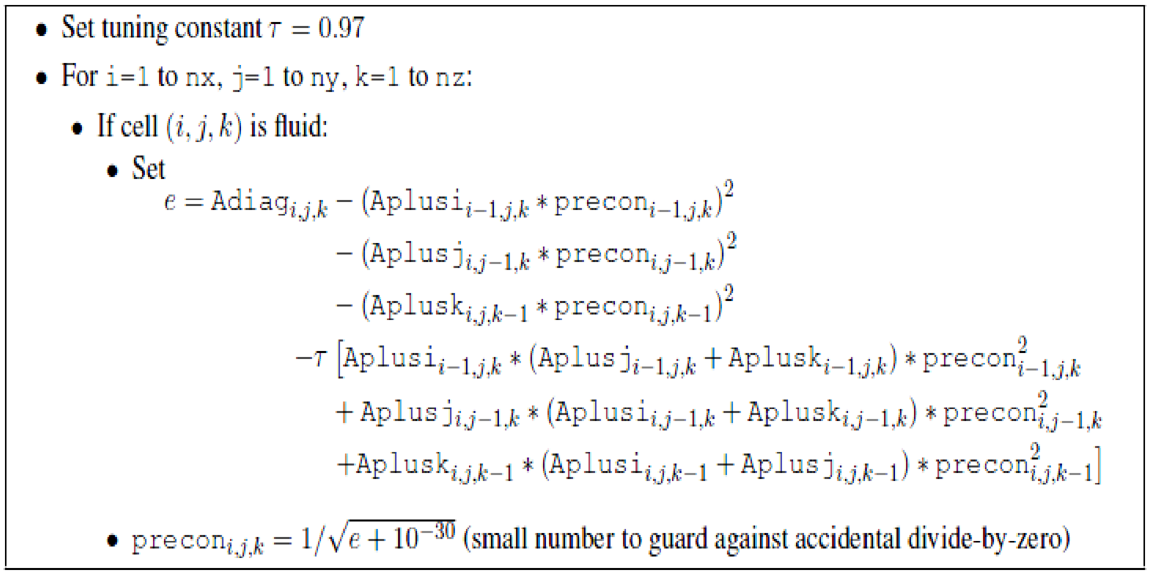
Modified Incomplete Cholesky Method

Finally, a slightly different variation of “**Incomplete Cholesky”** method is the **“Modified Incomplete Cholesky”** (MIC) which gives way better results in terms of convergence with almost no cost. In theory, MIC differs from IC only to the point that instead of cancelling nonzero entries to L in corresponding positons where A is zero, it adds it to the diagonal (Bridson, 2006).

And so, MIC is the same as IC but ensures that:

- The off diagonal nonzero entries of A are equal to the corresponding ones of LL^T
- The sum of each row of A is equal to the sum of each row of LL^T

And to gain even more better performance we take a weighted average of IC and MIC using a tuning constant *r* as shown below:



```
• Set tuning constant $\tau = 0.97$
• For i=1 to nx, j=1 to ny, k=1 to nz:
    • If cell $(i,j,k)$ is fluid:
        • Set
          $e = \mathtt{Adiag}_{i,j,k} - (\mathtt{Aplusi}_{i-1,j,k} * \mathtt{precon}_{i-1,j,k})^2$
              $- (\mathtt{Aplusj}_{i,j-1,k} * \mathtt{precon}_{i,j-1,k})^2$
              $- (\mathtt{Aplusk}_{i,j,k-1} * \mathtt{precon}_{i,j,k-1})^2$
            $-\tau\,[\mathtt{Aplusi}_{i-1,j,k} * (\mathtt{Aplusj}_{i-1,j,k} + \mathtt{Aplusk}_{i-1,j,k}) * \mathtt{precon}^2_{i-1,j,k}$
              $+ \mathtt{Aplusj}_{i,j-1,k} * (\mathtt{Aplusi}_{i,j-1,k} + \mathtt{Aplusk}_{i,j-1,k}) * \mathtt{precon}^2_{i,j-1,k}$
              $+ \mathtt{Aplusk}_{i,j,k-1} * (\mathtt{Aplusi}_{i,j,k-1} + \mathtt{Aplusj}_{i,j,k-1}) * \mathtt{precon}^2_{i,j,k-1}]$
        • $\mathtt{precon}_{i,j,k} = 1/\sqrt{e + 10^{-30}}$ (small number to guard against accidental divide-by-zero)
```

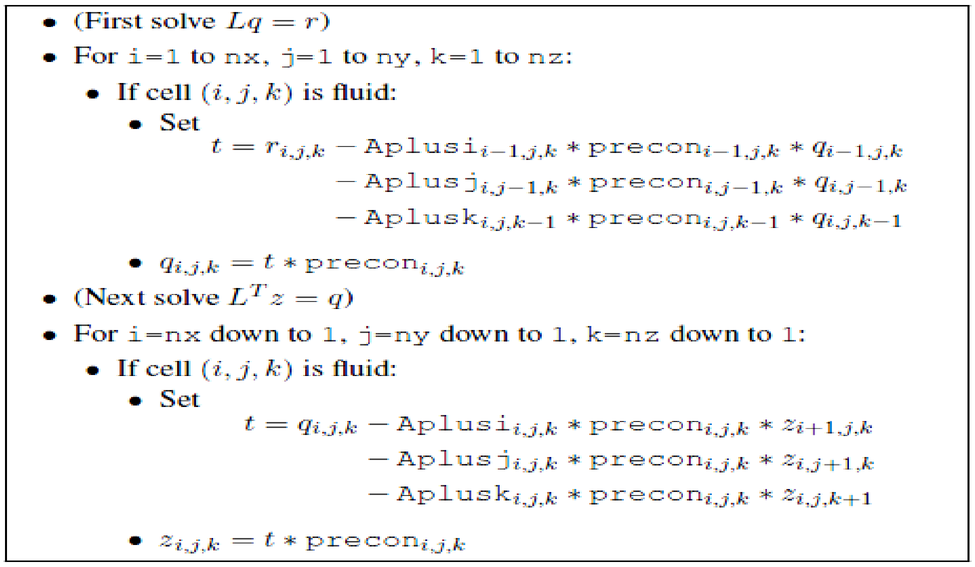
Table 12 : Calculation of the MIC preconditioner in 3D
(by http://www.cs.ubc.ca/~rbridson/fluidsimulation/fluids_notes.pdf)

where,

- **precon**, is the reciprocals of the E diagonal entries
- **Aplusi** & **Aplusj** & **Aplusk**, are the methods which we use to refer to the 6 neighbour cell faces of each grid cell in 3D.

Applying the Preconditioner

At this stage, we have already computed our preconditioner which is supposed to offer to us performance as far as the CG (Conjugate Gradient) is concerned. So, all we need to do is to apply this preconditioner to our 3D system which means to actually calculate the triangular solves (Muller, 2006).



```
• (First solve $Lq = r$)
• For i=1 to nx, j=1 to ny, k=1 to nz:
    • If cell $(i,j,k)$ is fluid:
        • Set
              $t = r_{i,j,k} - \texttt{Aplusi}_{i-1,j,k} * \texttt{precon}_{i-1,j,k} * q_{i-1,j,k}$
                  $- \texttt{Aplusj}_{i,j-1,k} * \texttt{precon}_{i,j-1,k} * q_{i,j-1,k}$
                  $- \texttt{Aplusk}_{i,j,k-1} * \texttt{precon}_{i,j,k-1} * q_{i,j,k-1}$
        • $q_{i,j,k} = t * \texttt{precon}_{i,j,k}$
• (Next solve $L^T z = q$)
• For i=nx down to 1, j=ny down to 1, k=nz down to 1:
    • If cell $(i,j,k)$ is fluid:
        • Set
              $t = q_{i,j,k} - \texttt{Aplusi}_{i,j,k} * \texttt{precon}_{i,j,k} * z_{i+1,j,k}$
                  $- \texttt{Aplusj}_{i,j,k} * \texttt{precon}_{i,j,k} * z_{i,j+1,k}$
                  $- \texttt{Aplusk}_{i,j,k} * \texttt{precon}_{i,j,k} * z_{i,j,k+1}$
        • $z_{i,j,k} = t * \texttt{precon}_{i,j,k}$
```

Table 13 : Applying the MIC preconditioner in 3D
(by http://www.cs.ubc.ca/~rbridson/fluidsimulation/fluids_notes.pdf)

5.4.7 Velocity Extrapolation

At this section, we describe the extrapolation of velocity out into the air. This step is essential for us so as to be able to sample the fluid's velocity along the trajectory of its particles When the fluid is moving towards new regions of the 3d MAC grid (towards the air for example) it actually moves outside the existing fluid volume where the velocity is not known at these points yet and so we need to define it.

5.4.7 .1 Velocity Extrapolation Algorithm

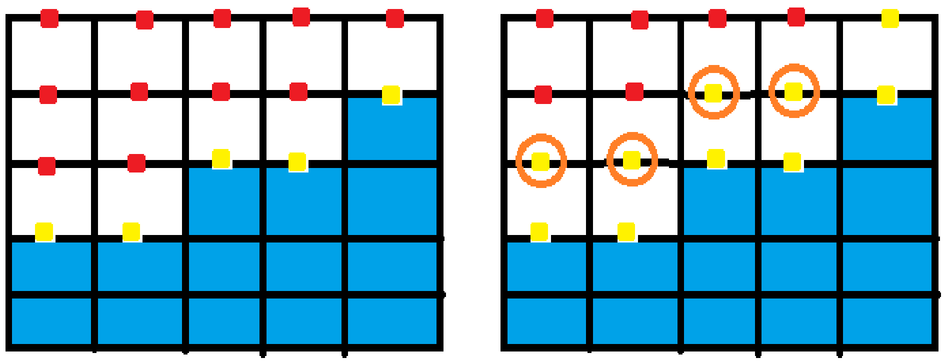


Figure 23 : Velocity Extrapolation in regions of the MAC grid where velocities are unknown yet.

We basically loop through the MAC grid cells and put markers in all fluid & near fluid as well as hard solid wall and empty cells (where current and previous cell solid or empty).
Then, we loop again through the MAC grid cells and for each cell marked as hard solid or empty cell and current cell is nowhere around fluid cells, loop through the list of each of this cell's neighbour cells according to its 6 faces. Finally, collect all of the grid velocities of all neighbour cells marked as fluid or near a fluid and assign a weighted average to the current cell.

```
• foreach i,j,k in grid_cells do
    o If( grid_cells(i,j,k)==fluid_or_near fluid)
        ▪ fluidMark=( grid_cells(i,j,k)
    o else If( grid_cells(i,j,k)== near_solid_or_air)
        ▪ solidOrEmptyMark=( grid_cells(i,j,k)
• foreach i,j,k in grid_cells do
    o If(grid_cells(i,j,k)== solidOrEmptyMark && grid_cells(i,j,k!=fluidMark))
        ▪ foreach face in neighourGridCells(i,j,k) do
            • If (neighourGridCells (face) == fluidMark)
              {
            • newGridVel= neighourGridCells(face).GetVelocity()
            • total++;
            • sum+=newGridVel;
              }
        ▪ weightedAverageGridVel = sum / total;
```

Table 14 : The algorithm for velocity extrapolation

By doing so, we manage to obtain the necessary velocity boundary conditions at and near the free surface of the fluid .

5.4.8 Pic Flip Velocities Calculation

At this stage we are about to calculate the flip velocities, the pic velocities and perform a linear interpolation between them so as to acquire a new pic/flip velocity. Because of the fact that pic velocity is suited to more viscous flows whereas flip velocity to inviscid flows, this interpolation between them allows us to specify the degree of an artificial viscosity of the fluid.

FLIP Algorithm

```
• foreach cell in MAC do
    grid_vel_change =subtract(new_grid_vel, prev_grid_vel);
• foreach p in particles do
    cur_particle_vel = saveParticleVelocity(p);
• foreach p in particles do
        cur_particle_vel += grid_vel_change(p) ;
```

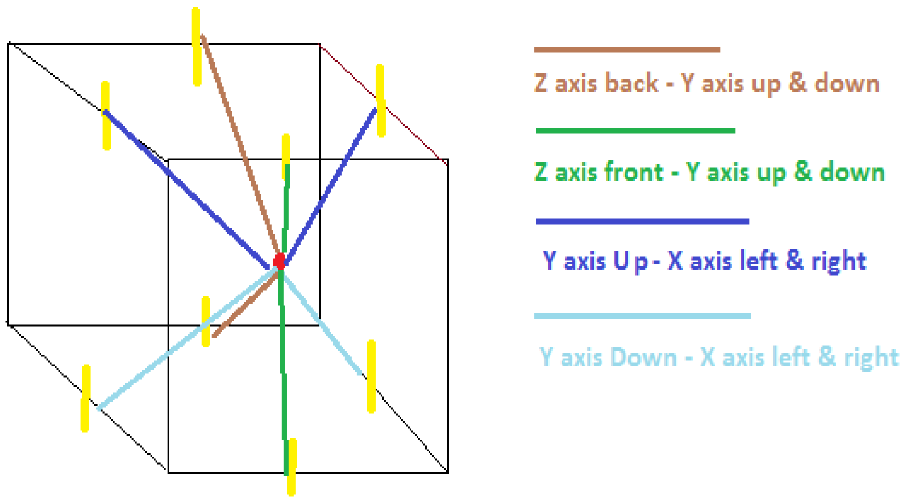
X =	new_grid_vel – prev_grid_vel
Y =	cur_particle_vel
Z =	X
Flip =	Z + Y

Table 15 : (reference table 1) Flip velocity calculation.

Here we notice that the changes of the grid velocity from a previous time step are mapped to the particles and used later so as to add onto the existing particle velocities. The whole point is that these grid velocities are then used to guide the motion

of the particles. The main disadvantage the flip velocity is that the fluid suffers from its noisy behaviour.
Now, as far as mapping the change in grid velocities from the grid to the particles is concerned, the method is described below:

METHOD EXPLANATION
For each particle we compute linear velocities based on the particle's position and cell index. More specifically, we sum up the 8 cell's velocities at the cells grid points with regards to the particle's position at this cell. This practically means that for a particle in a cell, we sum the percentages of contribution multiplied with the grid velocities and map them to the particle velocity.



Z axis back - Y axis up & down
Z axis front - Y axis up & down
Y axis Up - X axis left & right
Y axis Down - X axis left & right

Figure 24 : Transferring the grid velocities back to a particle in a cell performing tri-linear interpolation and contribution weighting based on the distance of the particle from each grid cell point carrying a sampled velocity.

PIC Algorithm

```
• foreach p in particles do
    cur_particle_vel = saveParticleVelocity(p);
• foreach p in particles do
        cur_particle_vel += new_grid_vel (p) ;
```

X =	new_grid_vel

Y =	cur_particle_vel
PIC =	Y + X

Table 16 : (reference table 2) Pic velocity calculation.

At this point, we are just perform trilinear interpolate as we did in flip algorithm above but instead of transferring back the change from new and previous grid velocities we just pass the new grid velocities found by the projection step back to particles.

Pic Flip Weighted Average

This is the last step of the pic/flip velocities calculation, where we actually perform a linear interpolation between the two different velocities calculated (flip and pic) so as to acquire the weighted average of both. By mixing those two velocities we can adjust the amount of viscosity of the whole fluid by an artificial *viscosity_coefficient* term as Pic velocities produce more viscous and Flip more inviscid flows.

$$final\_particle\_vel = interpolate\ (pic, flip, viscocity\_coefficient)$$

5.4.9 Particle Advection

At this section, we present and analyze the advection part of the algorithm in which particle positions are simply updated through the grid velocity field calculated at the previous steps. More analytically, we advect all particles of the simulation using the Runge-Kutta 4th order ODE (ordinary differential equation) solver which is highly accurate compared to other simpler and more inaccurate methods. Additionally, we perform all sorts of collision detection checks and responses such as for the all particles and the MAC grid or with solid obstacles plus some particle position correction for particles stuck at solid obstacles.

5.4.9.1 Runge-Kutta ODE Solver

From Newton's 2nd Law of motion we know that :

$$F = ma \rightarrow a = F/m \qquad (5.37)$$

We also know acceleration is the rate of change in velocity over time so:

$$\frac{d\boldsymbol{u}}{dt} = \boldsymbol{F}/\boldsymbol{m} \qquad (5.38)$$

The same way velocity can be calculated by the rate of change in position over time:

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{v} \qquad (5.39)$$

We can easily presume from the above that knowing the current position and velocity of an object and possibly all the forces that are applied to it we can integrate to find its position and velocity at some point in the future.

Runge-Kutta Method Mathematical Analysis

Take as an example the function $y = f(t, y)$ and $y(t_0) = y_0$, y is a function of both y and time and at the start point t*o* *y=yo* .

The RK4 method is extended as follows:

$$\begin{aligned} y_{n+1} &= y_n + \tfrac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \\ t_{n+1} &= t_n + h \end{aligned} \qquad (5.40)$$

where y_{n+1} is the RK4 approximation of $y(t_{n+1})$, and

$$\begin{aligned} k_1 &= hf(t_n, y_n), \\ k_2 &= hf(t_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1), \\ k_3 &= hf(t_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_2), \\ k_4 &= hf(t_n + h, y_n + k_3). \end{aligned}$$

So, each next value (y_{n+1}) is associated with the present value (y_n) plus the weighted average of four increments, where each increment is the product of the size of the interval, h, and an estimated slope specified by function f on the right-hand side of the differential equation.

- k_1 is the increment based on the slope at the beginning of the interval, using y_n, (Euler's method) ;
- k_2 is the increment based on the slope at the midpoint of the interval, using $y_n + \frac{1}{2}k_1$;

- k_3 is again the increment based on the slope at the midpoint, but now using $y_n + \frac{1}{2}k_2$;
- k_4 is the increment based on the slope at the end of the interval, using $y_n + k_3$.

In averaging the four increments, greater weight is given to the increments at the midpoint (Wolfram Mathworld, 2012).

The important thing to notice is that the error per step is of the order of h^5, while the total accumulated error has order h^4.

5.4.9.2 Collision Detection & Response

Detection
At this section we perform a simple collision detection framework for both fluid-fluid and fluid-solid collisions, since both are represented as particles at the current solver. The method we used is a simple sphere-to-sphere collision method in which a collision is detected if the distance between the queried particle and a neighbour particle is less than a specified *tolerance (collision radius).* This *tolerance* value is in immediate connection with the density of the fluid per grid cell. In fact,

Collision_radius = cell_density = density / cells

So,

If (distance < Collision_radius) → a *collision has been detected*

Response
At each collision detection there must be a collision response too. At this, step we initially find the normal direction vector of the plane that the two particles collided define. Then, project the velocity onto the normal's direction, push out the fluid particle from the solid particle to the normal's direction.

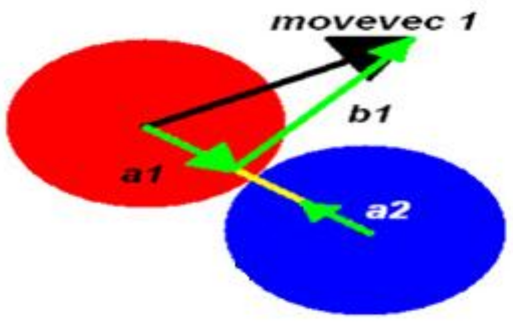


Figure 25 : 2D representation of the normal direction of the collision response (green vertical arrow)
(by http://www.gamasutra.com/view/feature/131424/pool_hall_lessons_fast_accurate_.php?print=1)

5.4.9.3 Stuck Particle Repositioning

At this stage, we perform a simple search for all fluid particles that happen to exist in regions where cells have been marked as solid wall cells at the stage of *fluid_surface_marking.* For all these particles found, they are labeled so to be repositioned somewhere else inside the fluid volume. This way, we can achieve both "fluid volume" maintenance and avoid the stuck particle artifact that happens as a result of the Lagrangian advection nature of the flip method which uses the grid velocity field as a guide.

Repositioning

For all particles labeled as stuck in the above step, we add them in a list and search for cells inside the fluid volume which are totally surrounded by fluid cells so as to reposition these particles there. After having moved them to their new positions, we update the grid cells with those particles by matching each one of the repositioned particles to their corresponding cell (see Chapter 5.3.4), and assign to them new velocities by resampling the neighbour particle velocities of their new positions (Ando, 2012). Both repositioning and resampling are presented below:

Reposition Algorithm

```
•   foreach p in particles do
       if(fluiparticle(p))
           if(p.cell==marked_as_wall)
                 label= reposition(p);

•   foreach p in particles do
       reposition_list.push_back(p);

•   foreach i,j,k in MAC_grid_cells do
           if(searchFluidSurroundedCells(i,j,k))
             //cell surrounded by fluid cells
             liquid_surrounded_list.push_back(i,j,k);

•   foreach n in reposition_list do
       particles[n].repositionTo(liquid_surrounded_list[n])
```

Table 17 : Reposition stuck particles to new positions somewhere inside the fluid volume

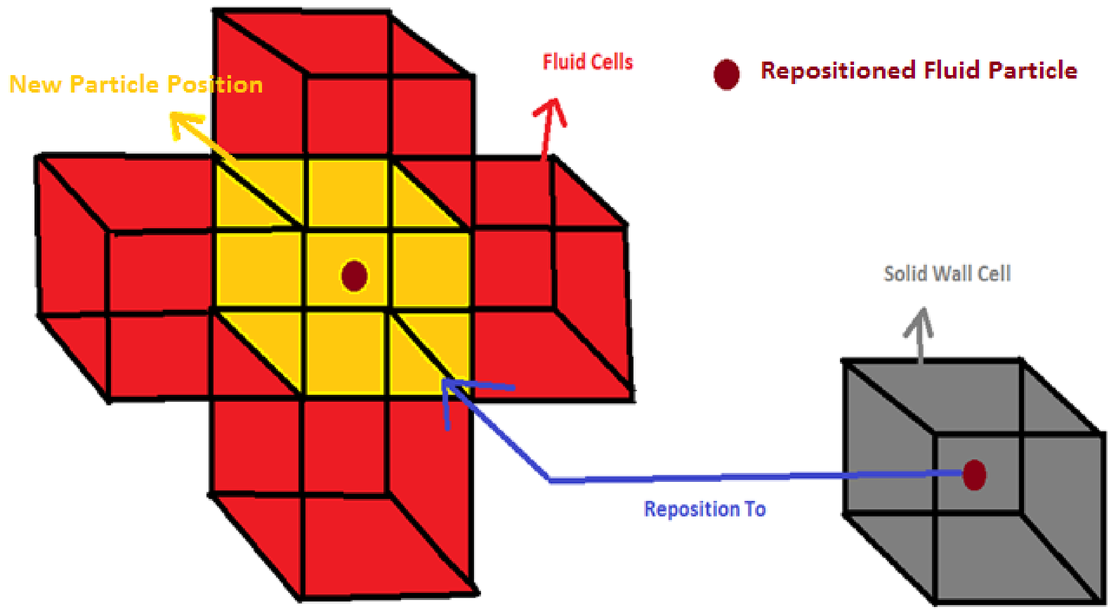


Figure 26 : Repositioning of a particle whose cell index is marked as solid wall to a pure fluid region (cell surrounded by fluid cells) cell.

Resample Algorithm

```
•   foreach p in reposition_list do
      neibourlist = findNeighbourParticles(p)
     foreach n in neibourlist do
          if(n==fluidParticle)
          {
           distance = findDistanceBetween(p,n)
           weighted_contribution = n.mass * W(distance,radius)
           p.vel += weighted_contribution * n.vel
           average_weighted_contribution += weighted_contribution
          }

    p.vel = p.vel / average_weighted_contribution
```

W : a stiff kernel used to measure each neighbour's contirbution

Table 18 : Assign new velocities to just repositioned particles based on resampling their new neighbours' velocities.

The weighted contribution of each neighbour particle to the repositioned particle could be expressed mathematically as:

$$u_{new}(\mathrm{p}) = \sum_i m_i \, \boldsymbol{u}_i \, W(p - n_i, h) \;\; / \sum_i m_i \;\, W(p - n_i, h) \tag{5.41}$$

where ,

- $\boldsymbol{m_i}$, the mass of a neighbour particle
- $\boldsymbol{u_i}$, the new velocity of a neighbour particle
- $\boldsymbol{n_i}$, the position of a neighbour particle
- $\boldsymbol{p}$, the position of the newly placed particle
- **h**, the radius of contribution of neighbour particle
- **W**, a stiff kernel function used

5.4.10 Particle Distribution Correction

It is generally known that in the pic flip method a surface reconstruction correction step is needed to overcome the problem of bumpy surfaces caused by the uneven distribution of the particles throughout the fluid flow. By applying a set of kernel weighted forces based on the “mass-spring” model amongst particles we manage to gradually remove the pic/flip noise and eventually achieve particles to be evenly distributed in space (Ando, 2011).

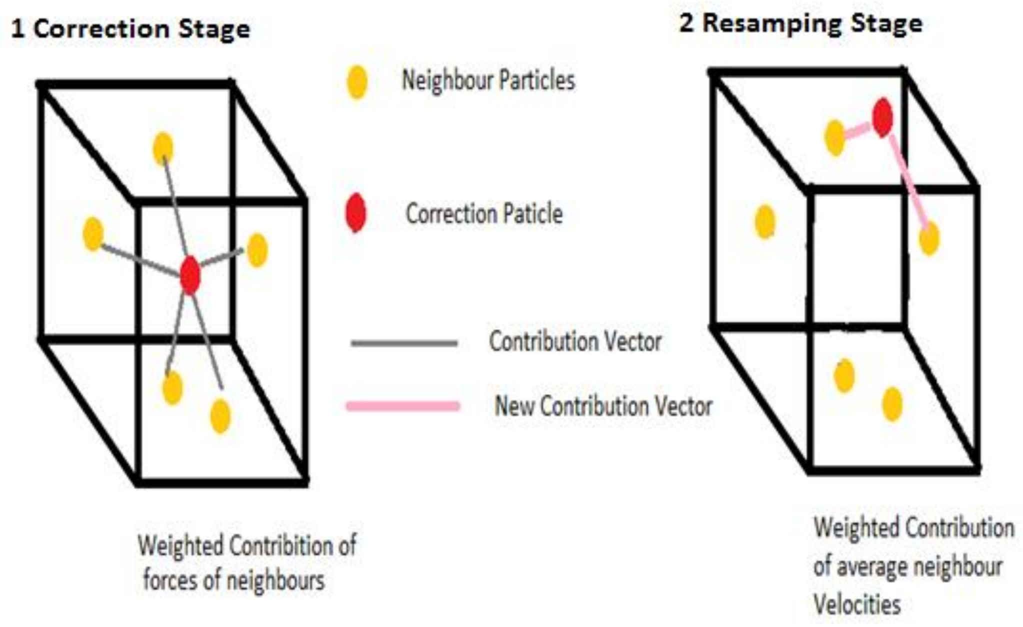
5.4.10.1 Correction Algorithm

```
   foreach p in particles do
     if(p==fluidParticle)
     {
       neibourlist = findNeighbourParticles(p)
         foreach n in neibourlist do
           sq_dist = findDistanceBetween(p,n)^2
              weighted_contr = spring_coefficient * n.mass * W(sq_dist, radius)

               if(sq_dist > Tolerance)
                  force(x,y,z) += weighted_contr * ((p.posit - n.posit) / sqrt.dist) * radius
                else
                  {
                    if(n==fluidParticle)
                       force(x,y,z) += weak_force*radius/time step
                    else
                       force(x,y,z) += strong_force*radius/time step
                  }
         //Xnew = Force * dt
         p.pos += force(x,y,z)*time step
      }

foreach p in particles do
// compute each new particle's velocity by resampling the neighbour velocities of the
// its new modified position
         newVel = velocityResampling(p.pos)

     UpdateParticlePositions(p,newVel)
     UpdateParticleVelocities(p.pos)
```

Table 19 : The algorithm description of the particles’ distribution correction.

In the above algorithm, for each fluid particle's neighbours we calculate a weighted spring force based on their weighted contribution (depending on their in between distance), we compute the total 3D force to be applied onto the current queried particle, we update its new position as a result of this force and finally we resample new velocities from new neighbours (based on new position found at the previous step) and apply newly computed velocity to the particle.

Figure 27 : On the left, a particle's position is being modified based on the total force



weighted contribution of neighbour particles.
On the right, the same particle at its new position is updating its velocity based on neighbour particles' velocities weighted average contribution.

5.4.11 Export and Write Particle Positions

This is the last step of our iteration algorithm. What we actually do at this stage is constructing one list with all fluid, static or RBD particles of each fluid to be exported at each frame. This list contains all of the current fluid particle positions based on which we create one **.geo** file (Houdini Ascii file format) per frame. Finally, after the simulation has finished we can read and load back the simulation data into Houdini for later rendering.

Simulation Data as .geo Houdini file format

The ASCII .geo & .bgeo file format is the standard format for storing Houdini geometry. The .geo format stores all the information contained in the Houdini geometry detail (Houdini12 Import-Export, 2012). Since .geo it's native to Houdini's environment but also a human readable and writable file format, it was chosen as the means to communicate and load the simulation data easily back to Houdini for later rendering. So, at each frame of the simulation we create a .geo file containing the number of points and primitives exported as well as their positions at each frame.

5.4.12 Artificial Vortex Behaviour

Figure 28 : Vortex fluid behaviour
(by http://www.123rf.com/photo_3435864_water-vortex-note-some-motion-blur-for-added-effect.html)

At this stage an extra new feature that has been added to the solver is presented and analyzed. This new addition has to do with the creation of artificial vortex behaviour of the fluid during its flow. Generally, as vortex we describe a spinning and often spiral, turbulent fluid flow as the motion of the fluid is swirling rapidly around a center. We also know that in a "free" vortex the velocity and the rate of rotation increases at the center of the fluid and decreases progressively outwards. In this project we have been inspired by this specific behaviour and tried to match and incorporate this concept when transferring particle velocities to the grid as a weighted contribution of neighbour particles of the cell depending on their distance.

Algorithm

```
foreach cell in MAC do
  neighbourlist = findNeighbours(cell)
     foreach n in neighbourlist do
        distance = cell.pos - n.pos

// a standard trilinear interpolation to compute the contibution weight of each neighbour to this cell
        weight = n.mass * W1 (distance, radius)

// a trilinear interpolation to compute an increased contribution weight of each neighbour to this cell
// using a squared distance and causing vortex-like behavior
       vortex_weight = n.mass * W2 (distance^2, radius)

     sumx += (weight +( vortex_weight *1.3)*m_vortexboost)*n->vel
     sumw += weight

 grid_vel = sumx/sumw
```

```
W1 : is a standard stiff kernel
W2 : is cubic kernel (adjusted for artificial vortex-like fluid behaviour
w :  a standard weighted contribution
sumx : gathers a mixture of both weights computed (standard-vortex)
sumx : gathers standard weights compute for mapping particles' velocities to grid
```

Table 20 : The algorithm description of the standard velocity mapping to a grid cell and an adjusted squared distance weighted contribution for the creation of the effect of an artificial vortex-spinning behaviour of the fluid.

In the above code snippet, we initially we perform the standard trilinear interpolation needed as part of the pic/flip algorithm to transfer the particle velocities and map them to our staggered MAC grid. Additionally, exploiting the varying velocity contribution of a free vortex type, where velocity scale of a fluid is proportional to the distance from its center, we have applied this behaviour locally for each of our grid cells and we achieved the spinning effect of an artificial vortex-like behaviour for our fluid. The amount of contribution of the vortex to the overall fluid behaviour is manually adjusted within a specific parameter called "artificial vortex boost", acting as a coefficient inside our custom picflipSolver HDA inside Houdini. The final grid velocity of each cell is computed by the ratio of the total vortex-adjusted weighted contribution over the total standard weighted contribution.

5.4.13 Smooth Kernel Functions

This section presents a number of smoothing kernels that are used throughout the implementation of the solver for different purposes but generally for calculating weighted contributions. More specifically, the main principle that governs these kernel functions is that they determine the amount of influence a particle may have on the particle that is being tested against. Moreover, these kernel functions use a radius distance h, which practically means that if the distance between two tested particles is less than this radius then a non-zero value is returned as a result of applying this kernel function, otherwise 0 is returned (Monaghan, 1992).

Loose Quadratic Kernel

This kernel function is used in the solver so as to approximate the amount of contribution of neighbouring particles when calculating density but also when weighting the contribution of forces of neighbour particles during the particle distribution correction step. This parametric function is of a form:

$$W(r,h)_{looseQuad} = \begin{cases} 1 - r / \; h^2 \;, 0 \leq r \leq h \\ \textit{otherwise} \\ 0 \end{cases} \qquad (5.42)$$

where,

- **r**, the distance between queried points
- **h**, the radius of influence

And its corresponding 3d graph is :

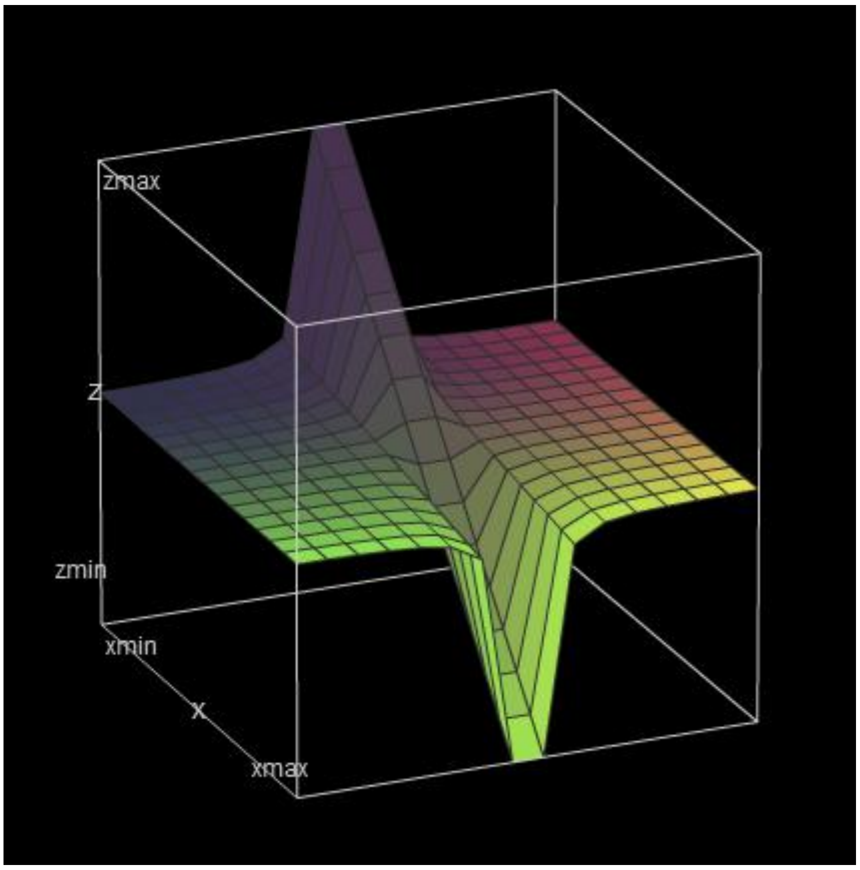


Figure 29 : Quadratic Loose kernel function 3D graph
(by http://www.math.uri.edu/~bkaskosz/flashmo/graph3d/)

Stiff Quadratic Kernel

This is another kernel function which is used in the solver so as to approximate the amount of contribution of neighbouring particles when resampling new velocities for particles just repositioned (see section 5.4.9.3), when the uneven particle distribution is corrected (see section 5.4.10.1) and also when transferring particle velocities to their corresponding grid cells . This parametric function is of a form:

$$W(r,h)_{stiffQuad} = \begin{cases} h^2/r - 1 \ , \ 0 < r \le h \\ \textit{otherwise} \\ 0 \end{cases} \qquad (5.43)$$

where,

- **r**, the distance between queried points
- **h**, the radius of influence

And its corresponding 3d graph is :

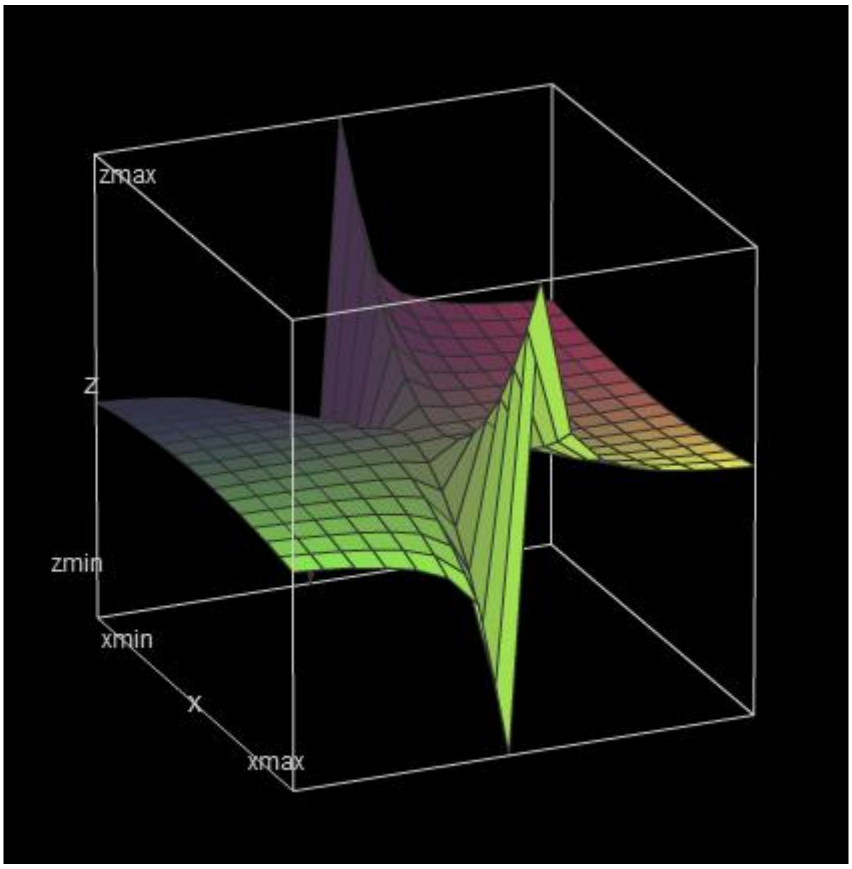


Figure 30 : Quadratic Stiff kernel function 3D graph
(by http://www.math.uri.edu/~bkaskosz/flashmo/graph3d/)

Cubic Kernel
This is the last kernel function used in the solver. It is mostly related to the artificial vortex effect added as a new feature, and it's used to approximate the amount of contribution of neighbouring particles of a squared distance when resampling new velocities for particles just repositioned (see section 5.4.9.3) and when the uneven particle distribution is corrected (see section 5.4.10.1). This parametric function is of a form:

$$W(r,h)_{cubic} = \begin{cases} 1 - r / 2 * h, & r \leq h \\ \textit{otherwise} \\ 0 \end{cases} \qquad (5.44)$$

where,

- **r**, the distance between queried points
- **h**, the radius of influence

And its corresponding 3d graph is :

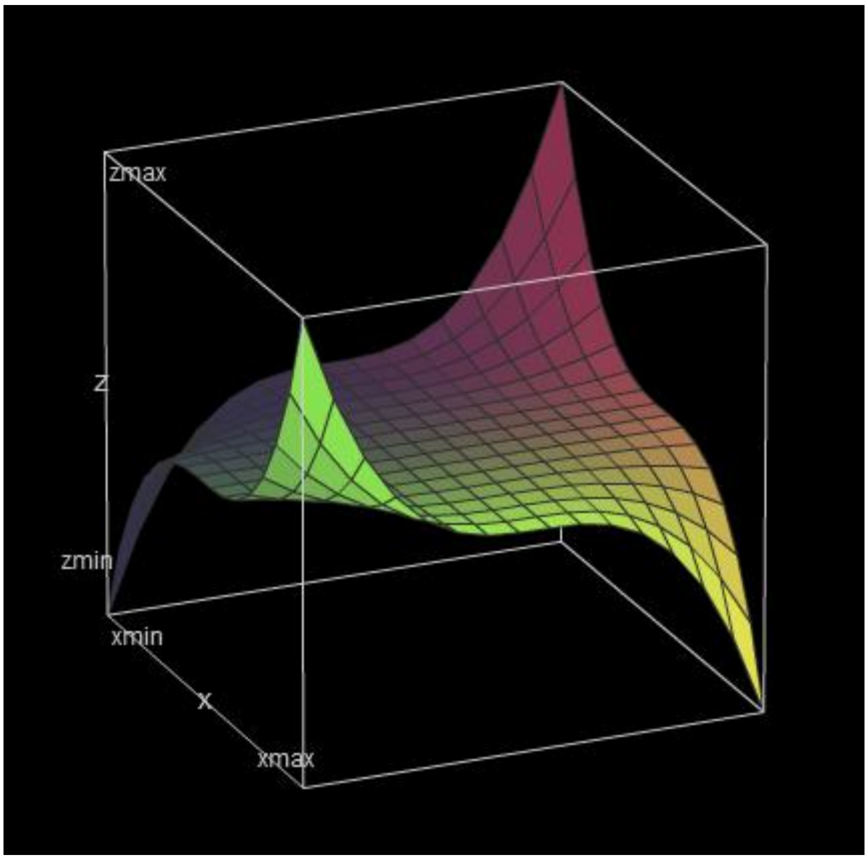


Figure 31 : Cubic kernel function 3D graph
(by http://www.math.uri.edu/~bkaskosz/flashmo/graph3d/)

6 Pipeline Issues

At this chapter we are going to explain in detail the pipeline used during this project which describes the custom solver's interdependencies, steps and issues that must be addressed and followed for a successful usage of this custom plugin.
At first, and as we mentioned earlier this is a custom solver that is meant to be run from inside Houdini 3D package and thus we will start with analyzing the Houdini side of the pipeline. Secondly, will present the solver's side pipeline issues and at the end we are going to return back to Houdini as it is responsible for reloading back the simulation data and render the final piece of simulation.

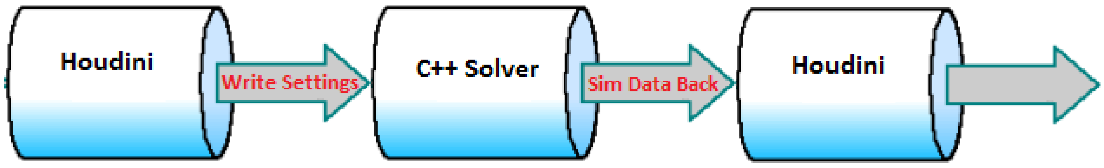
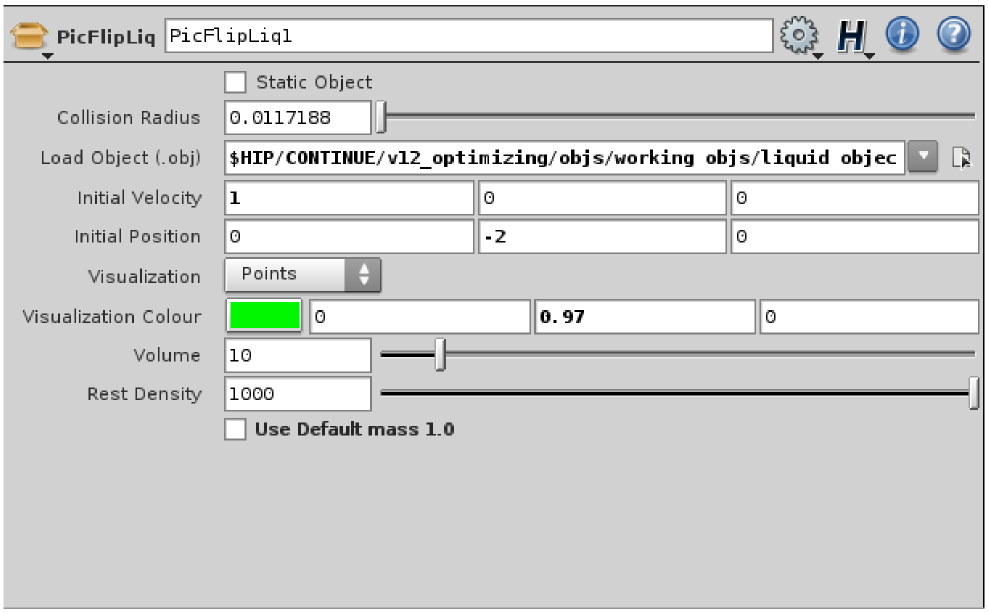


Figure 32 : Generic Pipeline of the project

6.1 Custom Houdini Digital Asset creation

6.1.1 PicFlip Fluid Houdini Digital Asset

The whole plugin concept is that a subject user will be able to open the Houdini 3D package and start setting up the simulation from inside there. For that reason, we created our first custom HDA , which represents an instance of a fluid object inside Houdini. This HDA loads any pre created .obj models to be simulated as fluids or RBD objects and contains a number of parameters to be tweaked regarding its behaviour during the simulation. The whole idea behind this asset for the custom solver to be capable of simulating multiple fluids inside a single scene, and that is the reason why there should also be a separate fluid instance nodes from which each one of them could be controlled as a HDA.


PicFlipLiq PicFlipLiq1

Static Object

Collision Radius	0.0117188			
Load Object (.obj)	$HIP/CONTINUE/v12_optimizing/objs/working objs/liquid objec			
Initial Velocity	1	0	0	
Initial Position	0	-2	0	
Visualization	Points			
Visualization Colour	🟩	0	0.97	0
Volume	10			
Rest Density	1000			

Use Default mass 1.0

Figure 33 : Custom HDA representing a fluid object to be added to the solver and be simulated

More specifically, this HDA contains a “Load Object” path which points to a pre-created .obj filled with particles to be simulated. As soon as the user specifies the path, we create a geometry SOP under “/obj” path inside Houdini which contains a file node (fluid object to be loaded), a “sphere” and a “copy” SOP which are used so as to allow the user to visually place and set up the fluid into the scene before simulating it. Additionally, there are more parameters which are explained in detail in a user guide accompanying this report.

6.1.2 PicFlip Solver Houdini Digital Asset

At this stage we are presenting any fluid assets previously created being added together to the main custom HDA solver. Later on, from inside this solver asset, we are able to control several parameters as far as the simulation is concerned; for example the viscosity of the fluids being simulated, the time step of the simulation, the number of the total simulation frames, the path to the settings file and the solver binary, the visualization type and colour of the fluids an so on.

Fluid Addition to the Solver

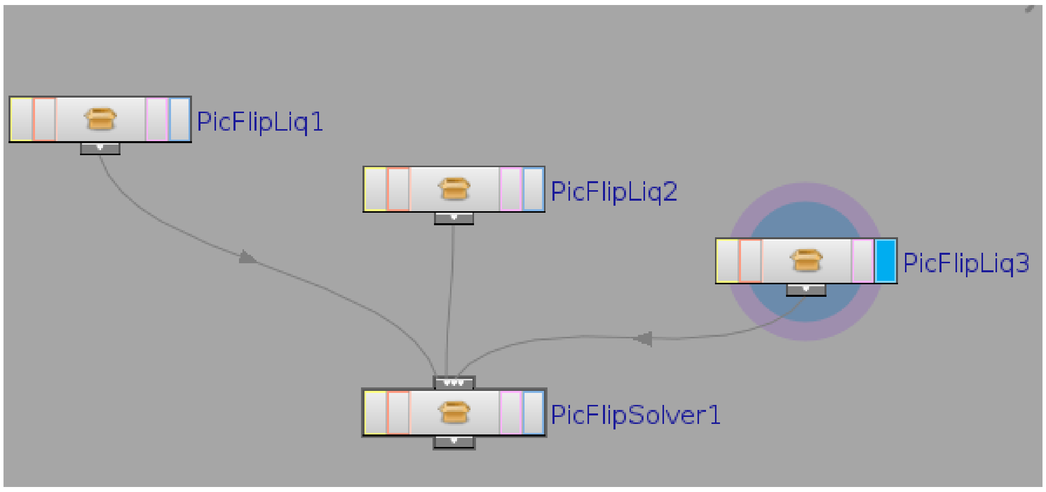
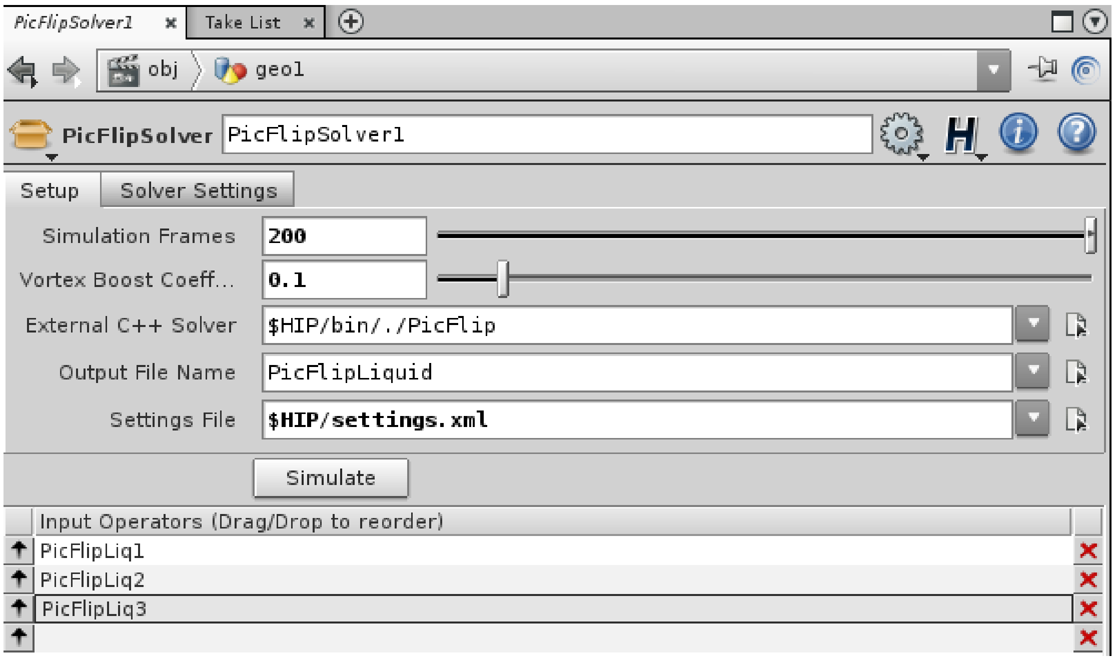


Figure 34 :Multiple Fluid Addition to the solver HDA

Solver HDA



PicFlipSolver: PicFlipSolver1

Simulation Frames: 200

Vortex Boost Coeff...: 0.1

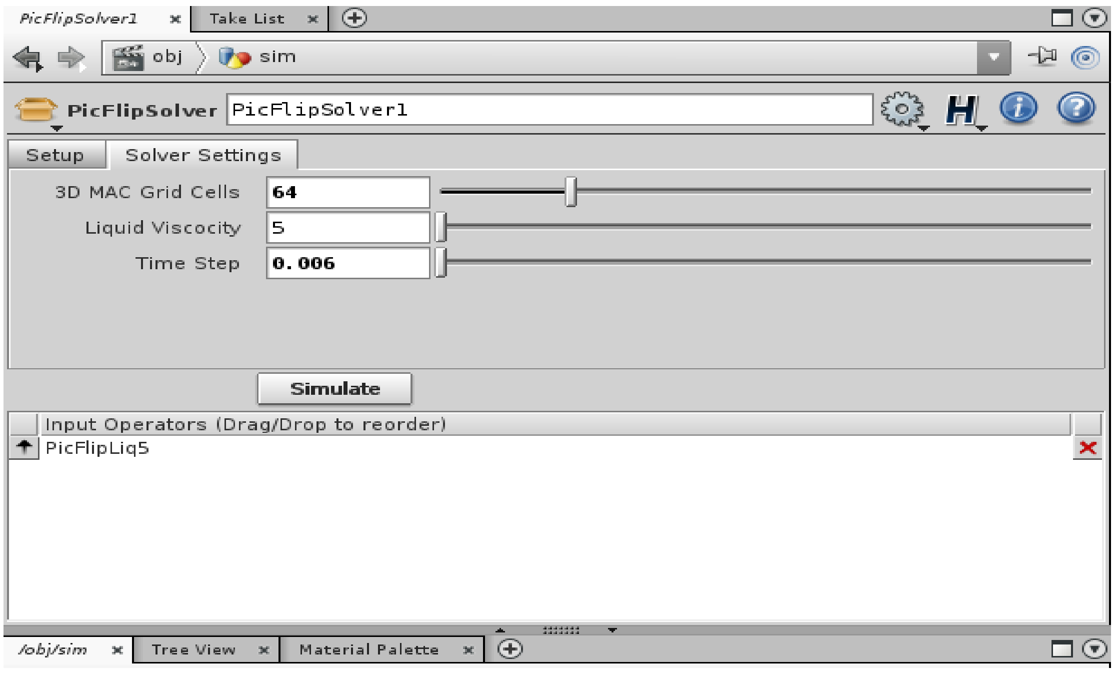
External C++ Solver: $HIP/bin/./PicFlip

Output File Name: PicFlipLiquid

Settings File: $HIP/settings.xml

Input Operators:
PicFlipLiq1
PicFlipLiq2
PicFlipLiq3

Figure 35 : Custom HDA representing the main solver which collects all fluids to be simulated (part 1)



3D MAC Grid Cells: 64
Liquid Viscocity: 5
Time Step: 0.006

Figure 35 : Custom HDA representing the main solver which collects all fluids to be simulated (part 2)

At this stage, after all fluids have been set up separately through their HDAs and the simulation parameters have also been finalized, all of these settings' information needs to be transferred to the external C++ solver which runs through the main HDA solver (picFlipSolver).

6.2 Python & Houdini

Houdini is a powerful 3D package which amongst others offers an application programming interface (API) that lets us collect information from and control Houdini using the Python scripting language. It is called HOM (Houdini Object Model) and allows us handling Houdini modules, nodes and functions and in this project we used it for writing custom scripts inside the Python module of our HDAs to:

- Create and visualize the fluid geometry (.obj) loaded within a fluid HDA.
- Control the fluid geometry's position after it has been loaded.
- Writing configuration settings from Houdini to the external c++ solver
- Read back simulation data and visualize the simulated scene

In the third and most important part of the whole usage of Python scripting in this project, we are about to retrieve all data needed for the external c++ solver's initialization step. We achieve that, using a Python script inside the solver HDA, which through the HOM allows us to have access to each of the HDAs' parameter names and values. We open an .xml file specified and write all need data there so as for the external solver to be able to parse them at the next step. Additionally, we trigger the external solver from inside the solver HDA (os.system(executable

+ " " + xmlfilepath)) passing the xml file created as an argument. Finally, after the external sover has finished its work, we create a geometry node under /obj inside houdini (named OutSim) which includes all fluids simulated with an appropriate "particlefluidsurface" node set up and attached to them for further visualization of the simulation. We should note here that fluids as well as static or RBD objects are loaded as .geo files created at each frame during simulation from the external solver. Finally, the user can render out the simulation according to its scene preferences (shading etc.).

6.3 XML Step Analysis

This section deals with the concepts of XML configuration file creation and parsing. Since we are creating a plugin which on its own is of a variable nature, we need the external solver application to be eligible enough so as to handle different user preferences and requirements specified as parameters inside Houdini's HDAs. One of the easier ways to achieve this is through the use of an XML configuration file to store these preferences.

At first, when the user presses the "Simulate" button of the custom HDA solver, an xml configuration file with all parameter settings is created into the current Houdini scene's directory. Since the external solver is placed there as well, it initiates the execution taking the xml file as an argument.

The second step, is for the external application itself to read and set up the solver based on this xml file. As xml is a file format which contains plain readable text embedded between tags, we simply parse configuration xml file's tabs (ex. <solver>, < liquid> etc.) and initialize the solver's variables based on their content. (See Appendix)

7 Simulation Results and Discussion

At this chapter we are going to present a number of simulations corresponding to a number of different scenarios so as to demonstrate the solver's full functionality. Additionally, we are about to discuss several issues that had to be taken into consideration during this project as well as the overall efficiency achieved.

7.1 Simulation Scenarios

Scenario 1 Single Primitive

In this first scenario, we are simply simulating a primitive sphere consisting of 15894 particles with almost no viscosity (viscosity 5%). There is nothing too interesting as far as this simple sphere simulation is concerned , apart from the fact it falls into the laws of physics and represents a quite accurate real world splashing liquid sphere.

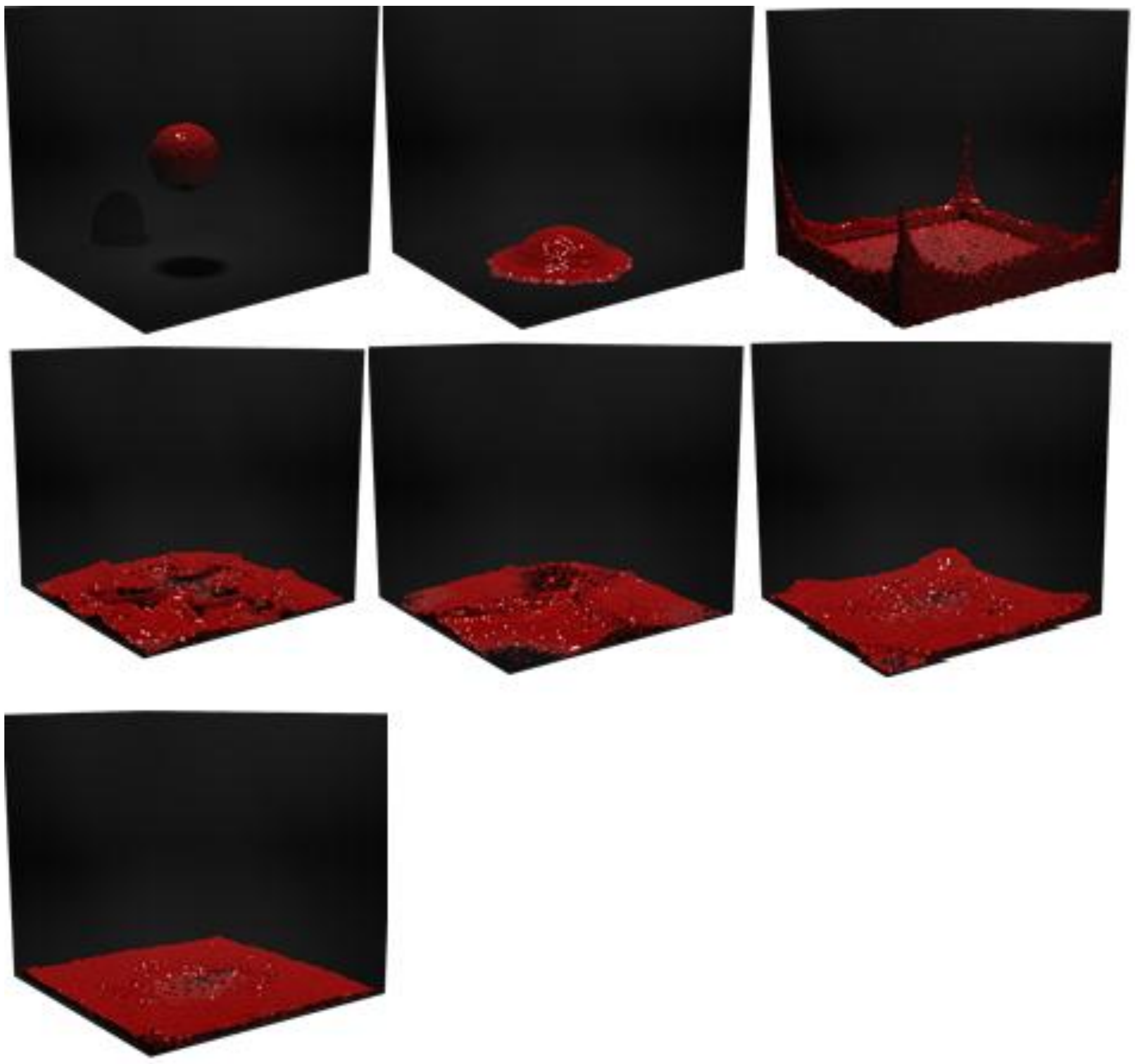


Figure 36 : Simple primitive liquid sphere 15894 particles with low viscosity (5 %)

Scenario 2 Single Liquid .obj Loading

In this second scenario, we are loading and simulating a pre-created .obj file consisting of 8048 particles with almost no viscosity (viscosity 5%). It is obvious that one of the project objectives (pre-created .obj loading using NGL graphics library (Macey, 2011) obj class) has been achieved with quite satisfying and promising visual results.

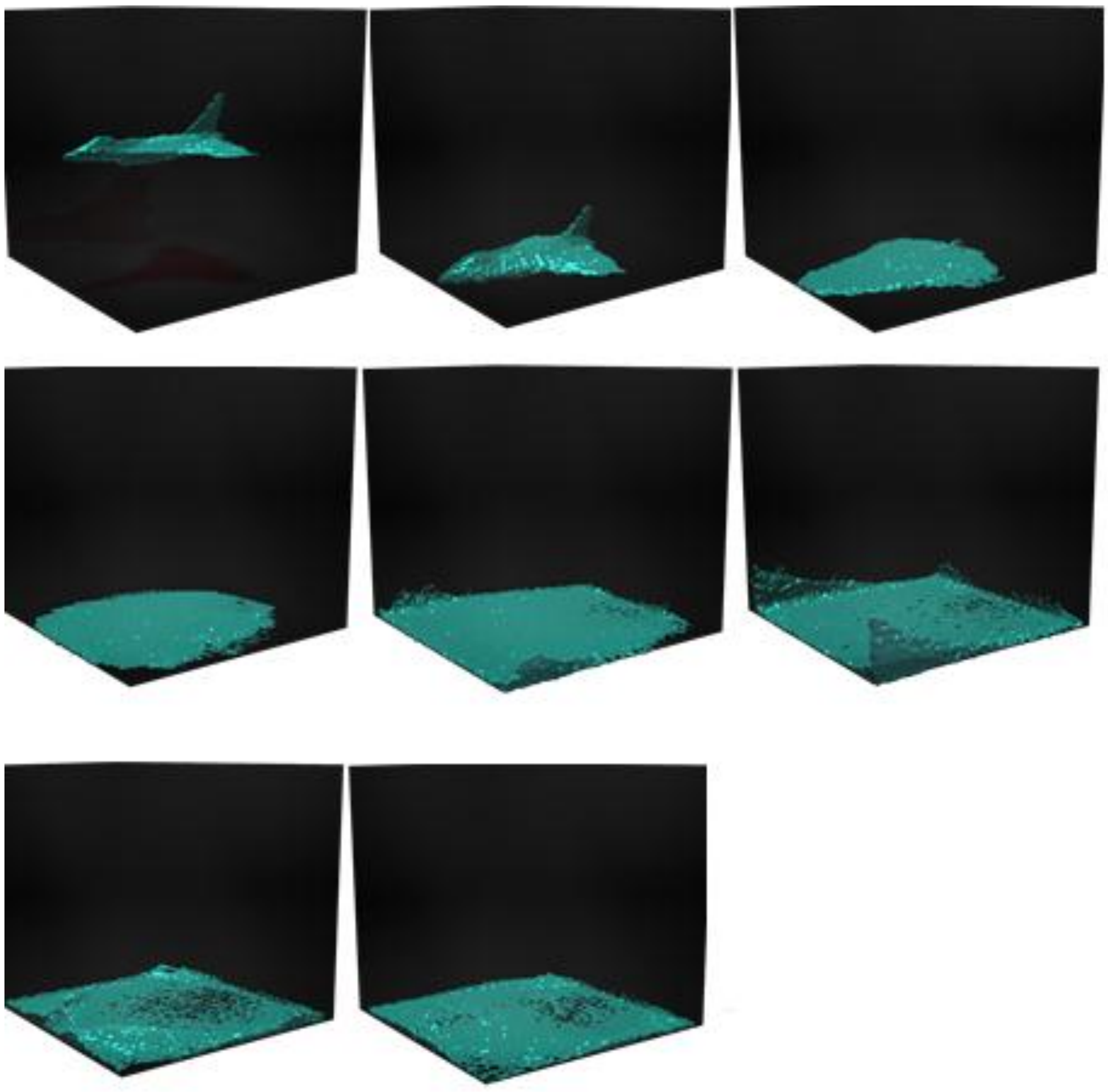


Figure 37 : .obj “Plane” model loaded and simulated as liquid with 8048 particles with low viscosity (5 %)

Scenario 3 Multiple Interacting Liquids

In this scenario, we present the solver's ability to simulate multiple fluids into a single scene. The most noticeable effect here is the big hole created at the center of the liquid box where the sphere falls, depicting a real world behaviour regarding the pressure which is analogous to the mass concentration (density). So, as the particles around a specific liquid area are increasing the pressure is increasing too causing the following effect. Viscosity is also set at low value (5%).

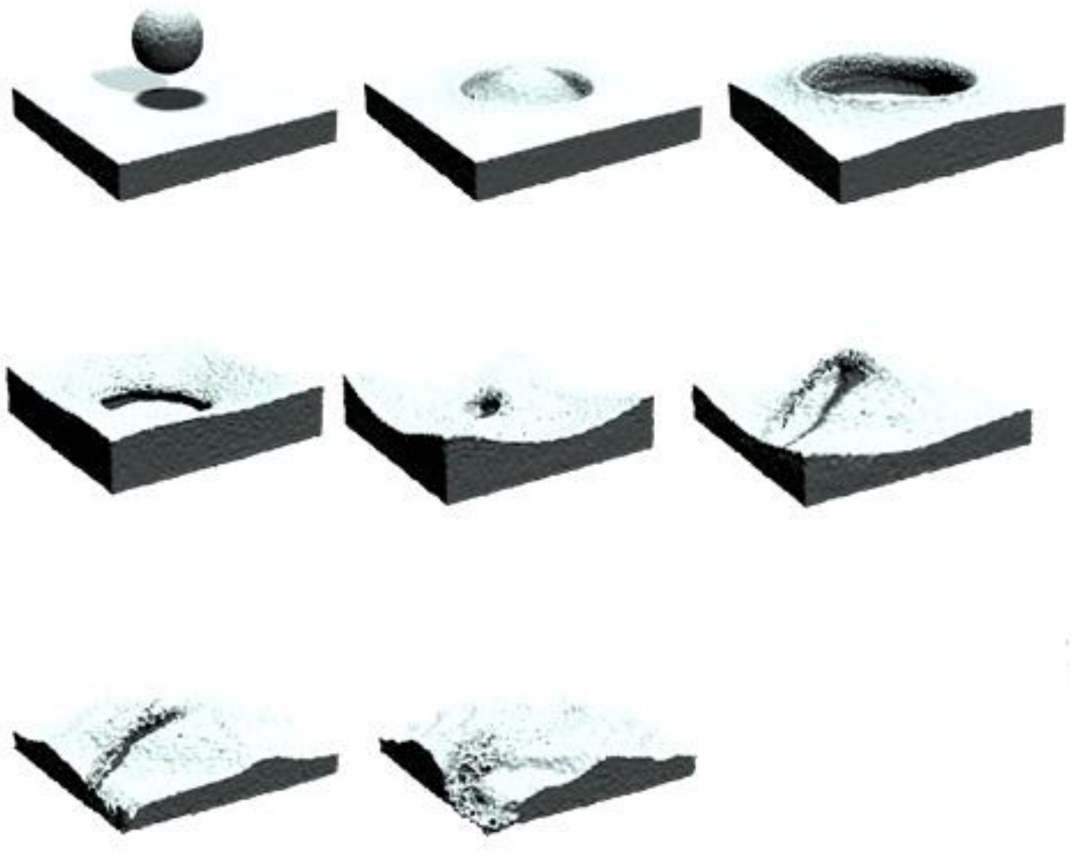


Figure 38 : Two liquids interacting with low viscosity (5%) : Primitive Sphere 15894 particles, Primitive Box 69216 particles

Scenario 4 Liquid Interacting with Static .obj

This is the 4rth scenario, in which we show one liquid and one static object both created inside Houdini and loaded into the external custom solver, interacting with each other. More specifically, there is a “camera” .obj liquid consisting of 130972 particles is colliding and being wrapped around with the solid “man figure” static object consisting of 55047 particles.

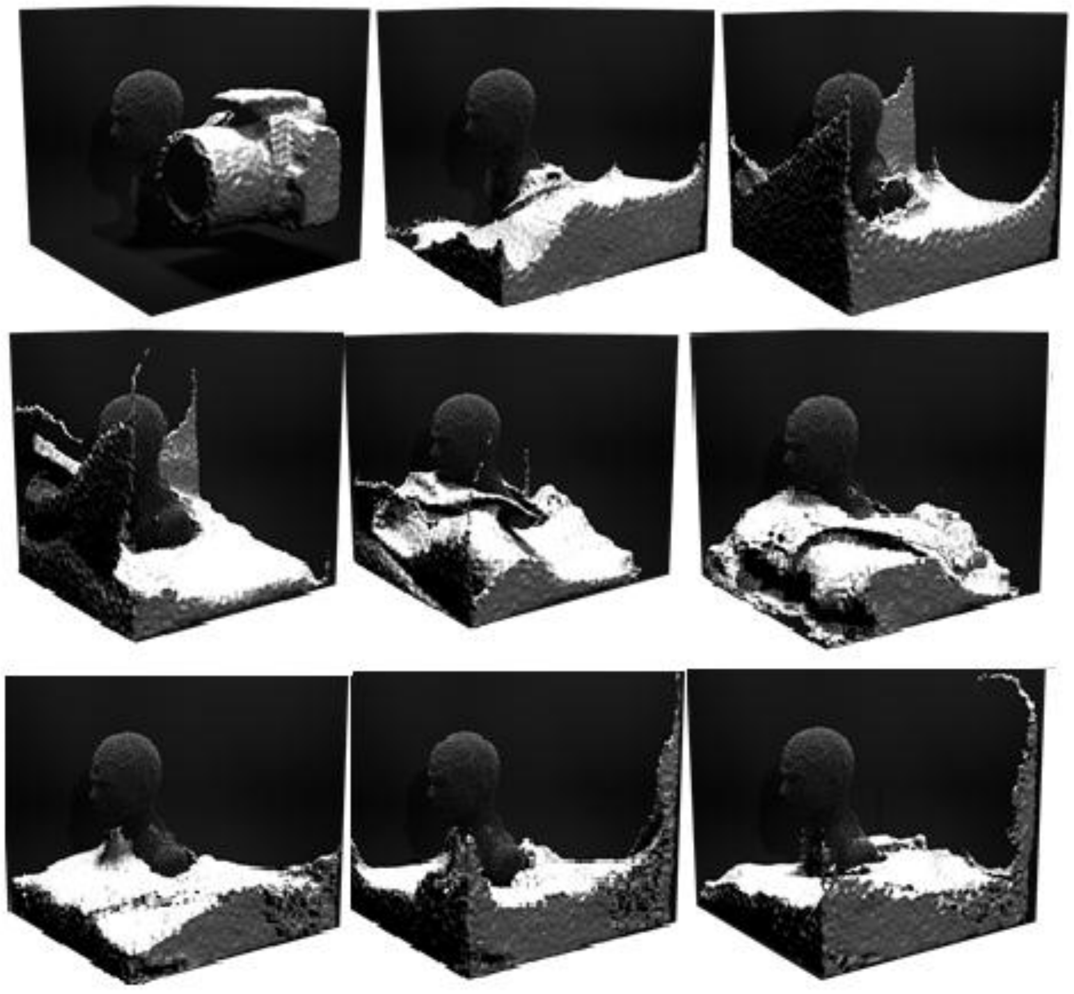


Figure 39 : “Camera” liquid object colliding with a static object in a single scene. Liquid’s viscosity is also set to (5%) low value : Camera 130972 particles, Man Figure 55047 particles

Scenario 5 Liquid Interacting with RBD .obj

At this scenario, a liquid object interacts with a rigid body object (RBD). There's an initial z axis velocity of -5 given to the RBD "bullet" .obj which is heading towards the "Man figure" liquid and penetrates it leaving a hole in the head area. Another, initial project objective has been met at this point by achieving simulating a real world situation where moving liquids may interact with moving RBD objects.

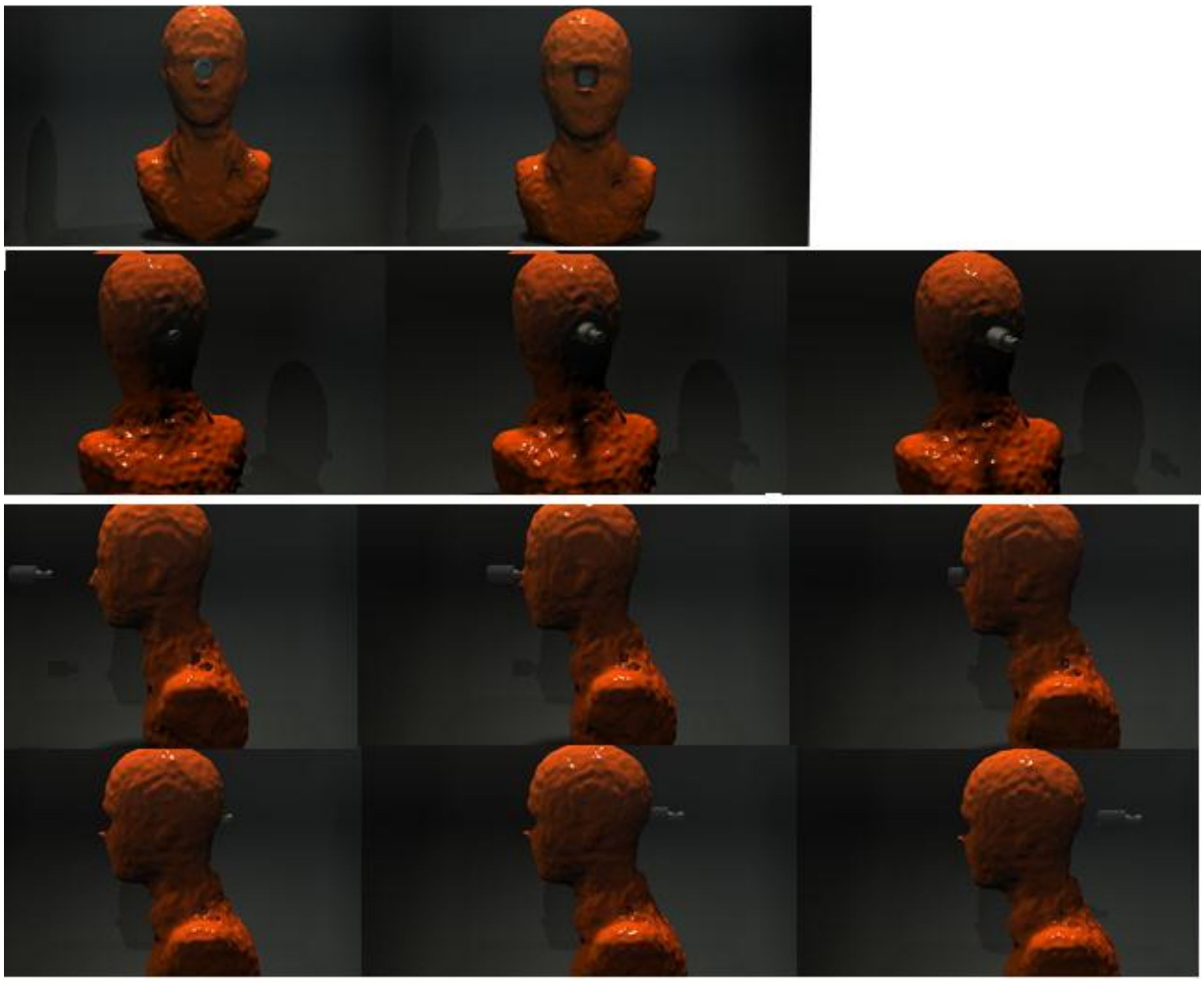


Figure 40 : "Man Figure" liquid object colliding with an RBD "bullet" object in a scene. Liquid's viscosity is also set to (5%) low value : Man Figure 55047 particles, Bullet 124 particles

Scenario 6 Viscous Liquid Interacting with Static .obj

In this scenario, we are examining the visual results of a viscous liquid object (“food ball”) interacting and colliding with a static primitive sphere object. The “foob ball” viscosity has been set to 1 so this practically means [visc*pic + (1-visc) *flip] (with visc=1) and so we are based on the viscous pic at 100% and 0% at flip velocities. The “food ball” consists of 6547 particles and the sphere consists of 15894 particles. As it is obvious from the rendered shots below, the liquid food ball shows a clear resistance to change its shape and volume topology which is a feature that high viscous liquids have in real world too.

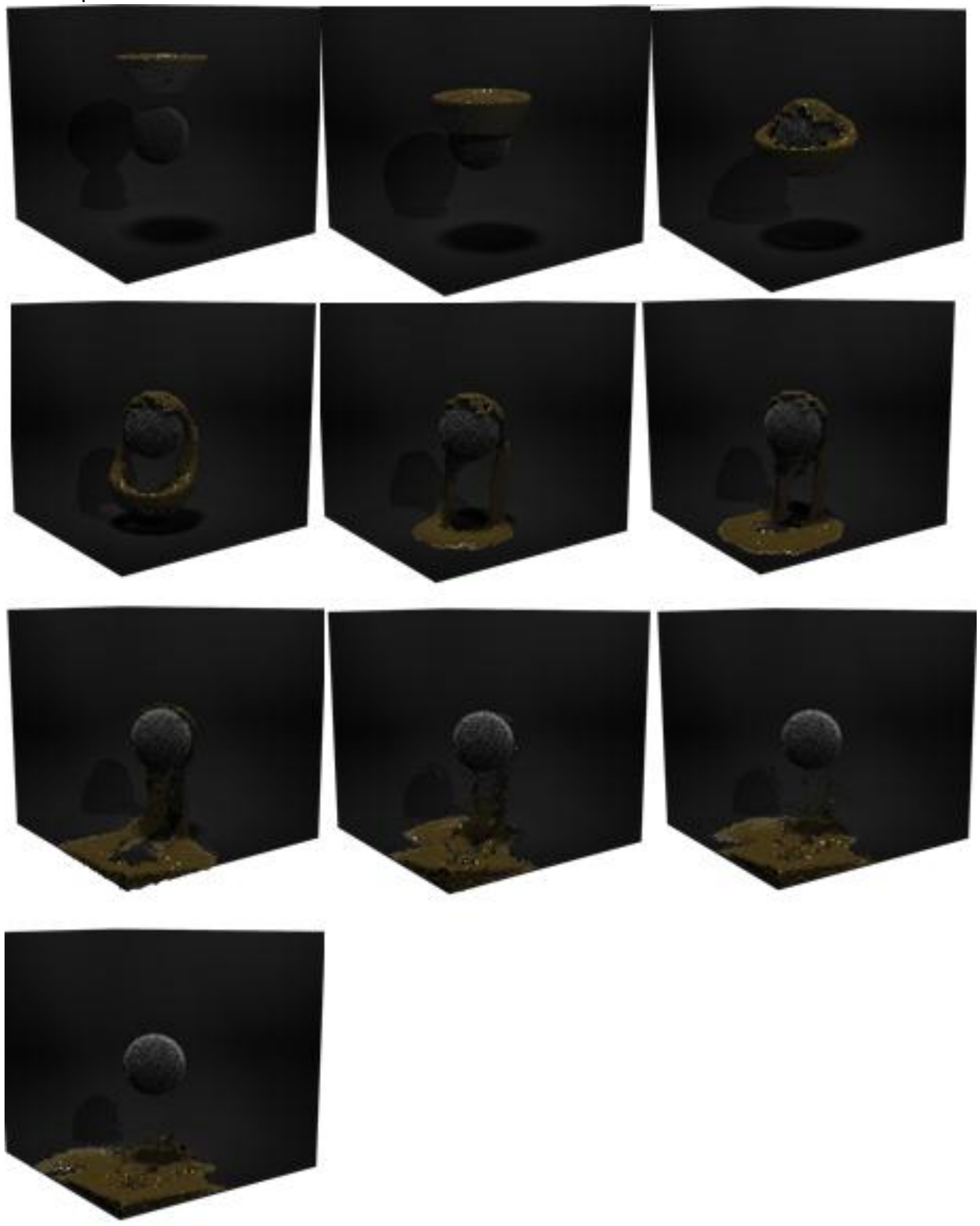


Figure 41 : Viscous "food ball" liquid object colliding with an static sphere object in a scene. Liquid's viscosity is also set to (100%) max value : Food ball 6547 particles, sphere 15894 particles

Scenario 7 Artificial Liquid Vortex Behaviour

In this last scenario we are about to present a new feature that has been added to the solver which causes a spinning turbulent fluid flow motion in the total fluid behaviour. As in real world vortex situations, the velocity and the rate of rotation increases at the center of the fluid and decreases progressively outwards; in this thesis the vortex behaviour is applied to each of the MAC grid cells during the simulation and so we can see many more spinning areas on the whole fluid rather than one big rotational hole in the middle of it. In this simulation, we have applied a "vortex boost" of value (1, maximum value for this solver) in a non-viscous primitive sphere, exactly the same with the one presented in the 1st scenario. It is more than obvious that the overall fluid flow has changes, its motion is way more turbulent which causes a variety of splashes and waves in different shapes and directions.

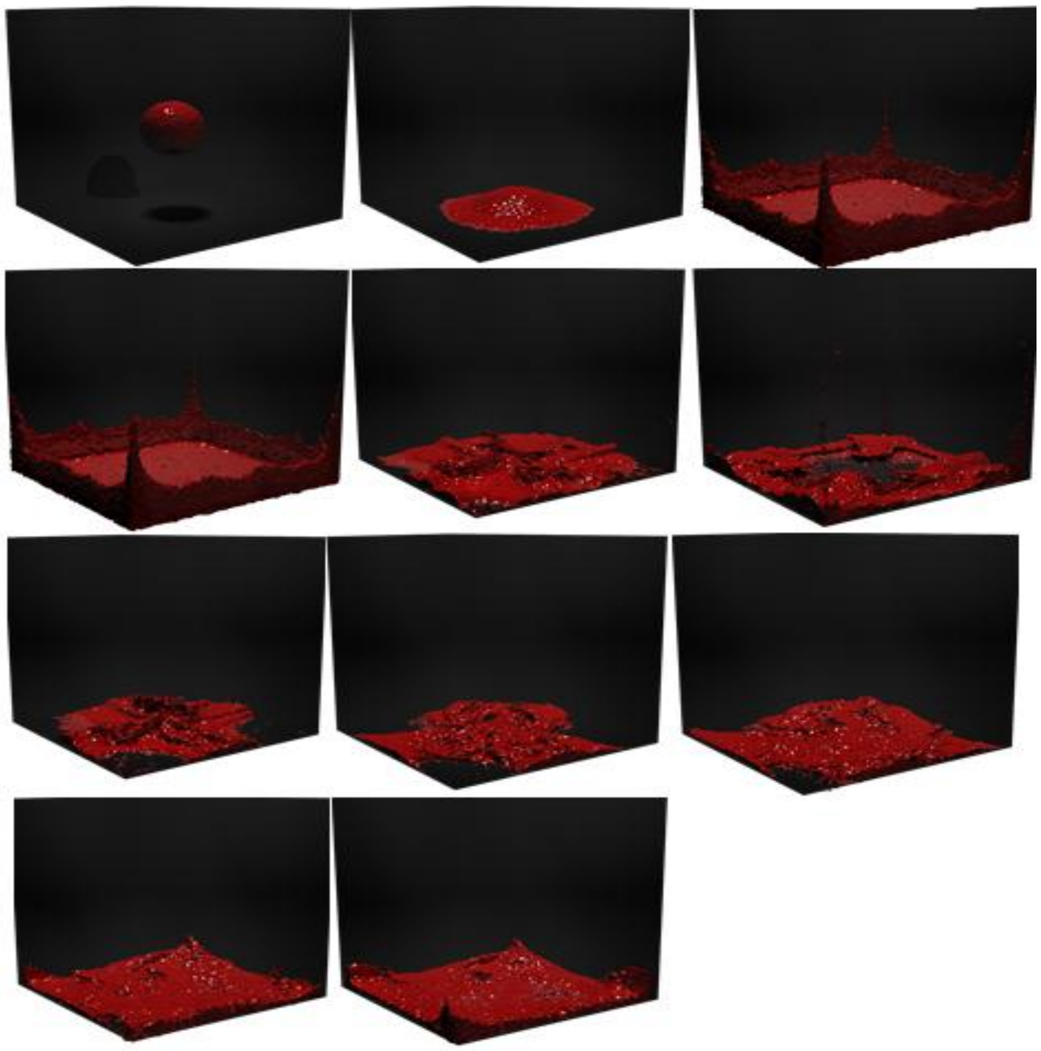
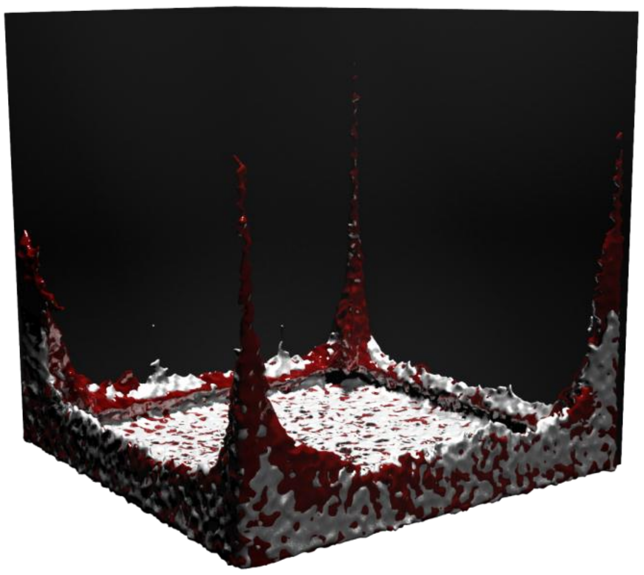


Figure 42 : Primitive liquid sphere with artificial Vortex Boost coefficient set to 100%. Sphere consists of 15894 particles. Noticeable is the spinning motion behaviour causing turbulent flow at the center of the fluid and big splashes at the edges.



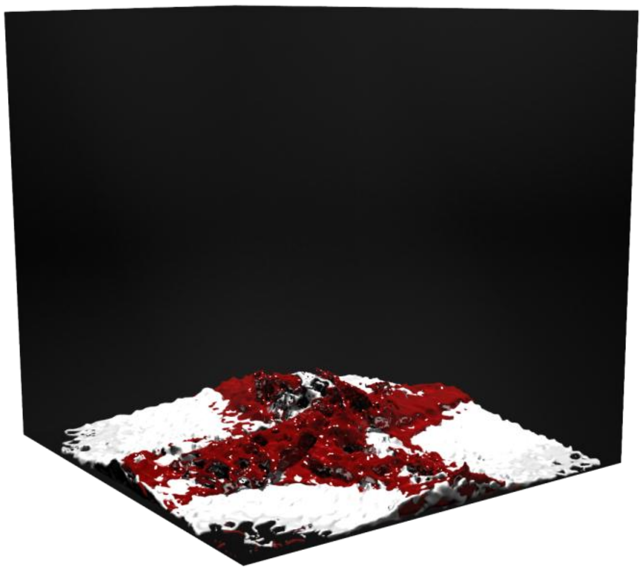


Figure 43 : Comparison 0% vortex boost (white liquid) and 100% vortex boost (red liquid). Note here, the two liquids are not interacting with each other but instead they have been pre simulated each one on each own and later placed in this single scene to be rendered together for demonstration purposes. The figure is just juxtaposing the different topology in the fluid flow as a result of the new artificial vortex boost coefficient feature added to the solver.

7.2 Known Issues

7.2.1 Pic Flip Issues

The most important issue with a Pic solver is that it suffers from numerical dissipation. This practically results in smoother simulations with lower accuracy as far as the fluid surface is concerned. On the other hand, a Flip solver shows a quite noisy and unstable behaviour when applied on its own. To alleviate this issue we are mixing the Pic velocities of the Pic solver with the Flip ones at a reasonable percentage as Bridson suggests in “Animating Sand as Fluid” Siggraph paper in 2005, and so we result in a quite accurate fluid flow which includes many thin fluid details such as small splashing areas without necessarily having to introduce a significant amount of numerical dissipation.

7.2.2 Time step Issues

The time step of the simulation is another very important aspect we have to consider when setting up the a scene. Most of the times especially when simulating liquids the time step can be increased even to a value of 0.01 since we only have fluids mixing with each other. However, one situation where we should be very careful with is when we are simulating fluids with static or rigid body objects where the time step should be way smaller to guarantee accurate fluid-static object interaction and proper early collision detection and response. Especially, when we are setting up initial velocities to one or both of the simulated objects where they are moving much more fast; in that case we found that a stable time step is a value of 0.006 and so the integration method can assert that all particles’ new positions are correct and we don’t experience the strange visual artifact of liquids intersecting with solid objects.

7.2.3 Viscosity Issue

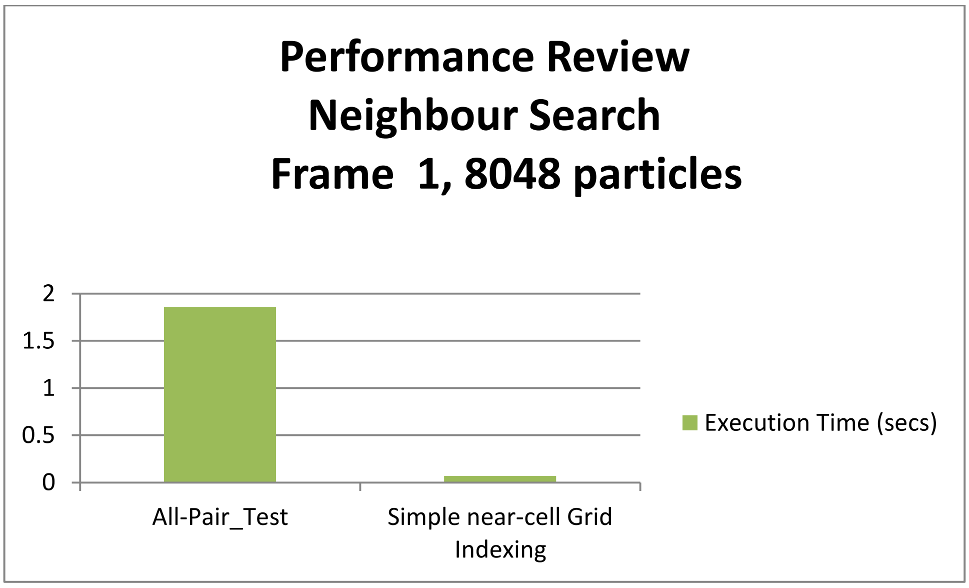
Fluid viscosity is another issue which we had to address during this project. Since we don’t explicitly solve the viscosity force of the Navier-Strokes Equation (3.3) we are taking advantage the fact that the Pic velocities found result in viscous flows while the Flip velocities found result and they are suited in more inviscid flows. By linearly interpolating both and adding a adjustment variable such as a “viscosity coefficient” to the solver we are able to control the viscosity levels from inside the solver HDA inside Houdini.

7.3 Performance and Efficiency

At this section, we are about to discuss the overall performance and efficiency of the solver. The initial target of such as solver is to have a linear complexity of O(n) as the number of particles in the simulation increases.

7.3.1 Neighbour Search

There is a variety of neighbour search methods researched in various papers offering big time boosts in fluid and other types of simulations so neighbour search is usually an important subject as far as the overall efficiency of solver is concerned. In this paper, we have initially thought to use 3D Spatial Hashing method so as to increase neighbour search. However for this project, since we were creating a grid based solver whose the structure we already had from a previous stage we chose a different approach which falls into the borders of a simple cell indexing technique in which we exploited the already existing grid structure and the main neighbour search was limited to the nearby cells of a queried particle's cell index,. The results as far as the accuracy of this method were quite convincing from the very start so we stuck to this solution which gives a close to linear complexity results in contrast with the naïve "all-pair-test" approach of iterating through all the particles of the simulation and measure distances from each queried particle each time resulting in O(n2) complexity.



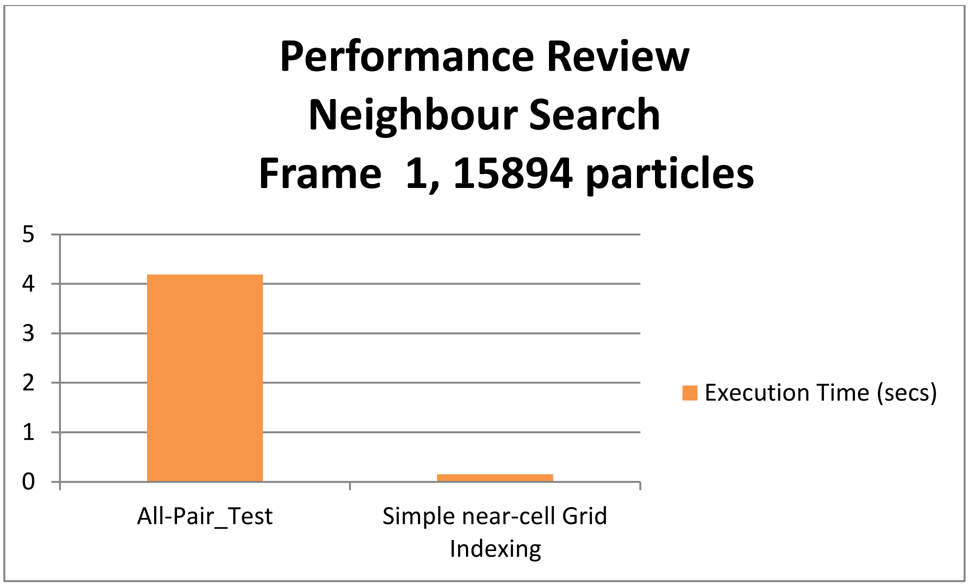


Figure 44 : The above column charts clearly show the difference in execution time and complexity of the 2 methods. "All-Pair-Test" gives a O(n2) complexity as the number of particles increases whereas by making use of this simple cell indexing method leaves the complexity at linear level.

7.3.2 Using OpenMP Directives

More and more these days, the complexity of fluid simulations is increasing and so the needs for alternative ways to keep up with this complexity definitely demands for more CPU cores' exploitation. As it was one of the initial objectives of this project, we had to increase the overall performance of the solver by including OpenMP. OpenMP is an API that supports multi-platform shared memory multiprocessing programming in C, C++ and other programming languages. It was chosen because it's a set of compiler directives and library functions that positively influence and boost the run-time execution behaviour of a program and in parallel gives a simple interface for parallelizing various kinds of applications (Yliluoma,2007).

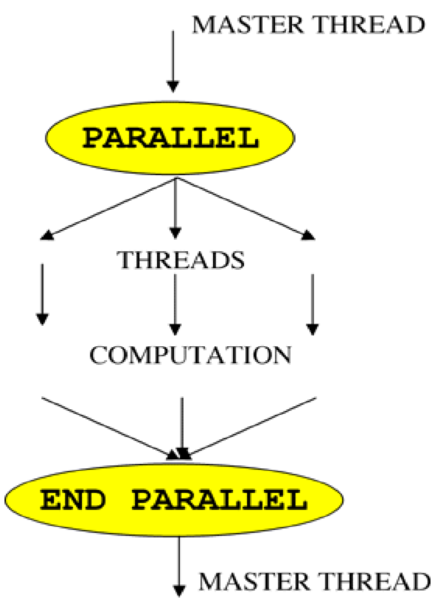


Figure 45 : The main fork-joint model of OpenMP (by http://static.msi.umn.edu/tutorial/scicomp/general/openMP/content_openMP.html)

The great features about OpenMP, which were the main reasons for choosing it as a method for increasing our solver's efficiency are:

- Easy of Usage
- Ensures automatic manipulation of pool threads which allows for mixing of sequential and parallel code without any user interference.
- Ensures automatic synchronization as an embedded functionality which pre-vents threads which share a "for" block of memory from continuing beyond the limits of this loop until all of them have finished execution inside it.

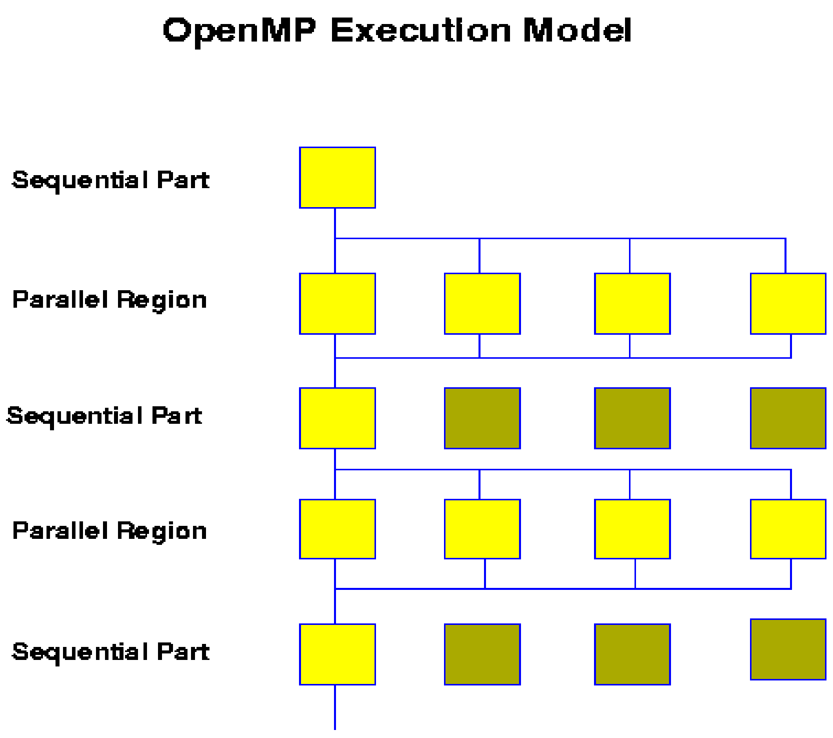


Figure 46 : A serial segment of code is being parallelized using all CPU cores available in the system and returns to the main execution Master Thread multiple times during the iteration of a program's algorithm
(by http://www.lrz.de/services/software/parallel/openmp/)

Below we present the results in execution times from two tests undertaken for a "Man Figure" liquid consisting of 55047 particles showing the performance gained by using OpenMP.

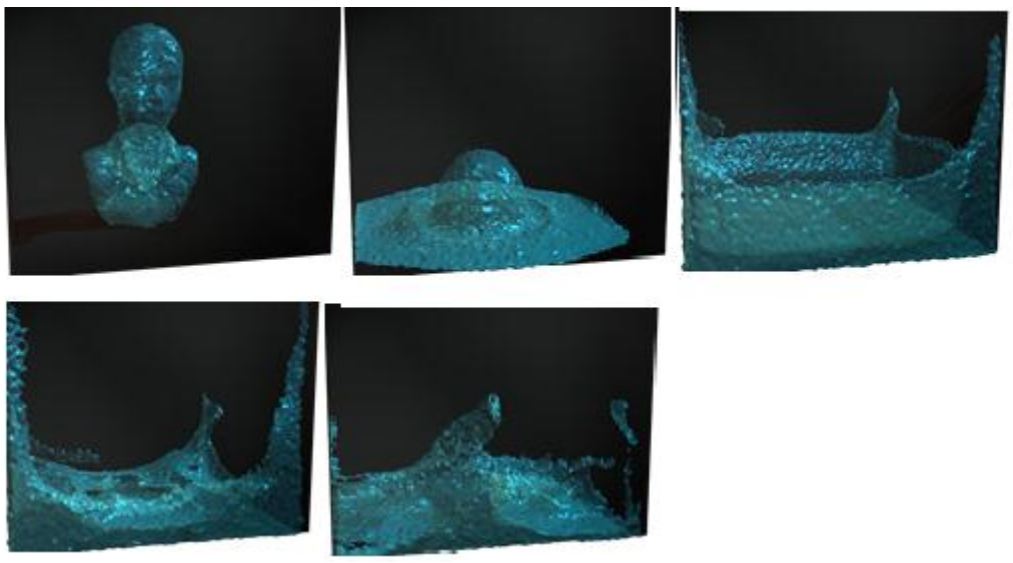


Figure 47 : "Man Figure" .obj liquid simulation test case for OpenMP consisting of 55047 particles with low viscosity (5%)

In the following column charts we present and juxtapose the boost in performance acquired from the use of OpenMP. Two specific cases have been examined:

1)The “Density Calculation”, which is a quite expensive method of our algorithm on its own .
2)The “Advection” method, which also takes a significant amount of time at each time step of the simulation.

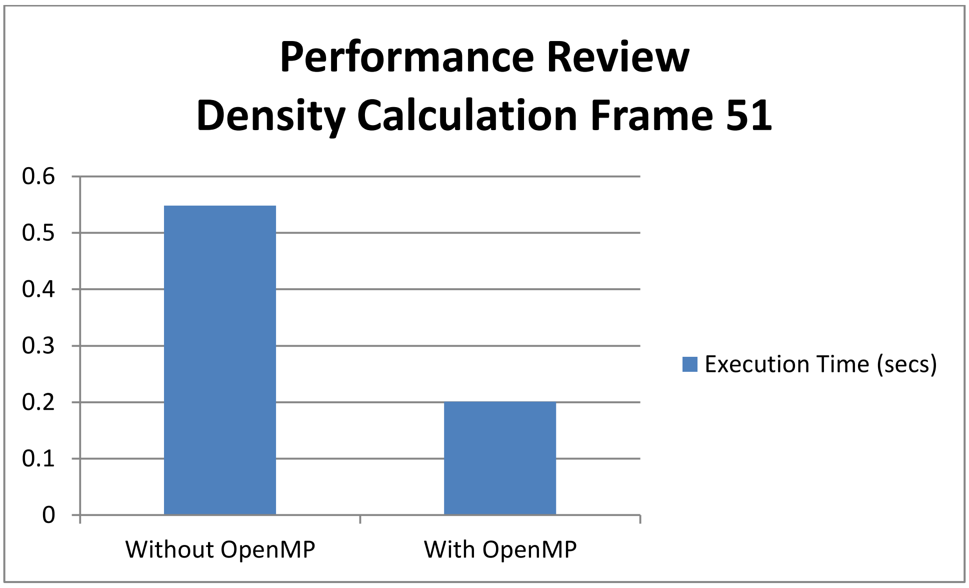


Figure 48 : Density Calculation execution time comparison of “Man Figure” .obj liquid simulation test case for OpenMP consisting of 55047 particles with low viscosity (5%)

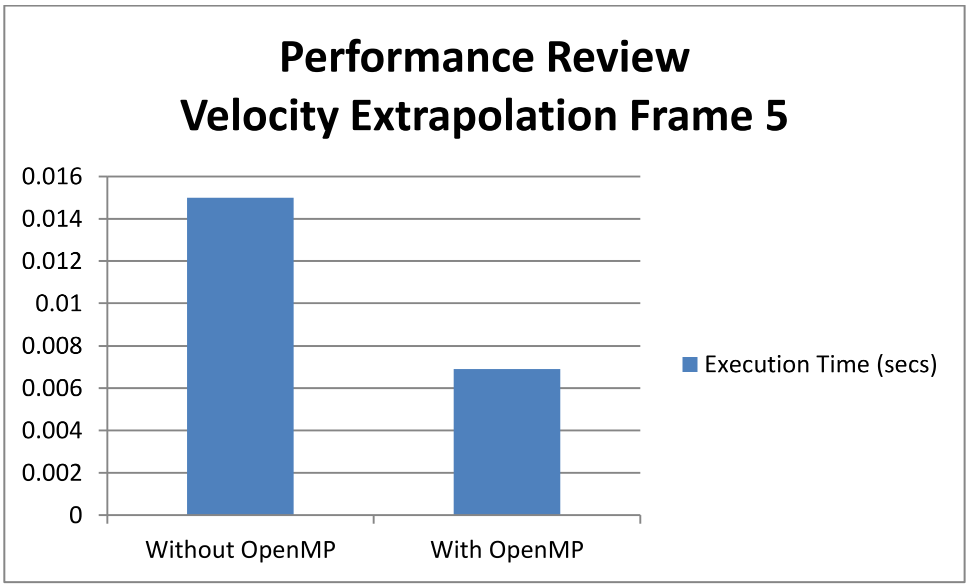


Figure 49 : Velocity Extrapolation execution time comparison of “Man Figure” .obj liquid simulation test case for OpenMP consisting of 55047 particles with low viscosity (5%)

The above examples are only two examples of the main loop that have been parallelized with OpenMP directives and it is obvious that we actually boost performance more than 50% overall. Finally, we should mention that for 100 frames of simulation of the "Man Figure" .obj, it took 6.83 minutes without the use of OpenMP directives and on the contrary it took 3.33 minutes for the same amount of frames using OpenMP. Thus, we successfully introduced a significant boost up in the execution time of our solver achieving one more very important initial objective of this project.

8 Conclusion and Future Work

In this final chapter we are about to discuss and draw a conclusion for the whole project. In general, in this project a Pic / Flip solver custom plugin for simulating fluids inside Houdini 3D package was implemented using OpenMP directives. It is considered to be successful as it managed to meet all of its initial aims and objectives relating to the understanding and implementation of a fluid solver based on the Pic/Flip method, the incorporation to Houdini using custom HDAs to load, set up and control the pre simulation stage, but also the introduction and application of OpenMP compiler directives which helped in boosting up the overall fluid solver's efficiency.

8.1 Critical Analysis of Initial Objectives

- At first, we achieved to simulate a number of fluids into a single scene interacting with each other offering quite pleasing visual results.
- At the pre simulation stage, we also managed to control and vary several fluid parameters such as the initial position, the initial velocity, the visualization type and color of the fluids to be simulated
- The interaction amongst fluids and static or RBD objects was also another one of the project's initial targets so as to simulate real world situations where fluids collide with static objects.
- Additionally, we were able to handle large amounts of simulation data in terms of the number of particles in a scene with the use of the simple near-cell indexing technique for neighbour search which prevented quadratic complexities O(n2). On the top of this, we increased the overall solver's performance with the successful use of OpenMP compiler directives.
- We managed to set up and control the solver based on the parameters specified in the custom Houdini plugin. For this purpose an interface parser was successfully created to read and import an xml configuration file pre-created inside Houdini.
- Furthermore, a proper point position exporter was embedded in the solver so as to write particle positions to a Houdini compliant file format (.geo) for later loading and rendering of the final simulation.
- Apart from fluid HDA, we also achieved in setting up different simulation parameters with the use of a custom Houdini Digital Asset for the solver as well,

and Python scripting was used to retrieve and write all appropriate needed settings' from fluid and solver HDAs' parameters to a configuration file.

- Last but not least, we managed to successfully load any simulation (.geo) data created at each simulation frame back into Houdini and added a particle fluid surface node so as to properly visualize the loaded simulation as soon as it's finished using Python scripting as well.

8.2 Possible Extensions and Improvement

As this was a quite limited time project, there are a number of possible improvements that could be applied to the existing solution and make it even more flexible and efficient.

Efficiency

Initially, and as far as performance is concerned a possible extension would be to parallelize the solver on the GPU and since graphics cards these days offer an increasing number processing cores the current solver could run significantly faster than simply on a multi-core system. The generic idea behind performing fluid simulation of particles in a grid on the GPU is to use a fragment programs that perform the computations at each of the grid's cell for solving separately one by one all the steps of the algorithm such as advection, force application, viscosity diffusion etc..

Collision Detection & Response

Moreover, since for the current representation of any static or RBD objects we are using particles, this could be extended to representing static objects with polygonal or parametric surfaces and computing collision detection and response of liquid interactions with them. This would prevent any slowdown in the overall performance of the solver which is caused by the iteration over any extra particles added to represent the static or RBD objects but it would introduce a significant overhead depending on the algorithm chosen to handle the collision detection-response. For example, in the case of a collision detection algorithm which performs checks at a polygonal mesh RBD we would have to check against every polygon possibly with some sort of point-Polygon intersection algorithm which is quite computationally expensive and it can be even more inefficient as the number of polygons of the RBD increases. Of course, there are many other ways to perform collision detection and response each one with its specific advantages and disadvantages such as AABB (Axis Aligned Bounding Box) method and others but this falls out of the scope and time limitations of this project.

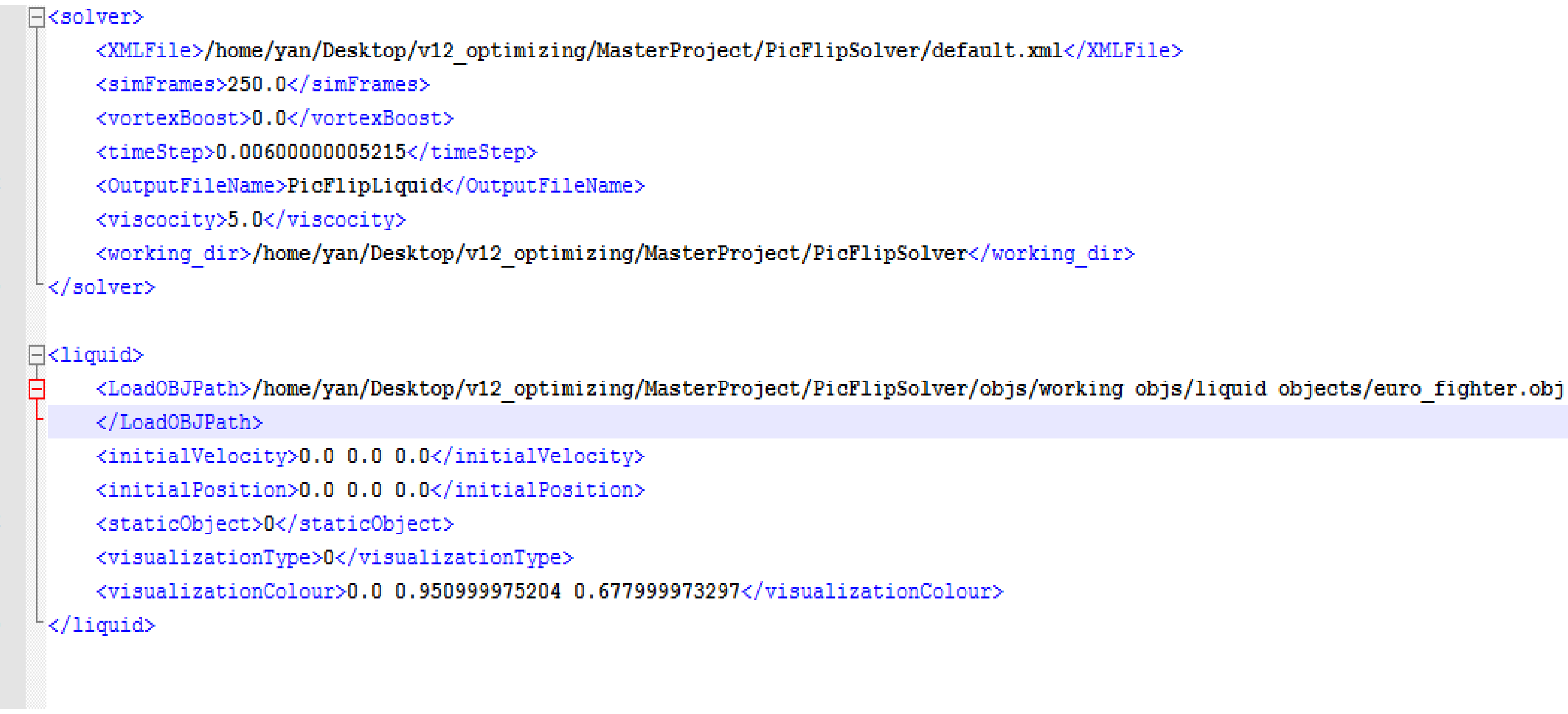
Full RBD System Extension

Finally, the solver could be extended to include and solve complete fluid RBD interactions. This basically means that a full rigid body system would have to be integrated so as to allow any fluid forces to be applied back to the RBD objects as well and affect their overall behaviour and motion. It could achieved by developing a custom rigid body system or use a ready to use library and incorporate it to the already existing solver.

Appendix

Configuration Settings XML file example

Figure 50 : One example of the configuration settings file in .xml format written to the current directory from inside Houdini. The c++ external application is responsible for parsing the lines of this file and set up the solver accordingly.



```
<solver>
    <XMLFile>/home/yan/Desktop/v12_optimizing/MasterProject/PicFlipSolver/default.xml</XMLFile>
    <simFrames>250.0</simFrames>
    <vortexBoost>0.0</vortexBoost>
    <timeStep>0.00600000005215</timeStep>
    <OutputFileName>PicFlipLiquid</OutputFileName>
    <viscocity>5.0</viscocity>
    <working_dir>/home/yan/Desktop/v12_optimizing/MasterProject/PicFlipSolver</working_dir>
</solver>

<liquid>
    <LoadOBJPath>/home/yan/Desktop/v12_optimizing/MasterProject/PicFlipSolver/objs/working objs/liquid objects/euro_fighter.obj
    </LoadOBJPath>
    <initialVelocity>0.0 0.0 0.0</initialVelocity>
    <initialPosition>0.0 0.0 0.0</initialPosition>
    <staticObject>0</staticObject>
    <visualizationType>0</visualizationType>
    <visualizationColour>0.0 0.950999975204 0.677999973297</visualizationColour>
</liquid>
```

Bibliography

1. Fluid Simulation for Computer Graphics, R. Bridson, A K Peters, 2008.

2. Zhu , Y. Bridson , R., 2005. Animating Sand as a Fluid . Association for Computing Machinery ACM SIGGRAPH, (2005).

3. Brackbill, J.U., Kothe , D.B., Ruppel , H.M., 1988. FLIP: A LOW-DISSIPATION, PARTI-CLE-IN-CELL METHOD FOR FLUID FLOW . Computer Physics Communications, 48 (1988), 25-38.

4. Cummins , S.J., Brackbill , J.U , 2002. An Implicit Particle-in-Cell Method for Granular Materials . Journal of Computational Physics , 180, 506–548 .

5. J.U. Brackbill, H.M. Ruppel, FLIP: A method for adaptively zoned particle-in-cell calculations in two dimensions, J. Comput. Phys. 65 (1986) 314343.

6. BARDENHAGEN , S. G., B RACKBILL , J. U., AND S ULSKY, D.
2000. The material-point method for granular materials. Comput. Methods Appl. Mech. Engrg. 187, 529–541.

7. Ando, R., Turey, N., Tsuruno , R., 2012. Preserving Fluid Sheets with Adaptively Sampled Anisotropic Particles . IEEE TRANSACTIONS ON VISUALIZATION AND COM-PUTER GRAPHICS ,VOL. X, NO. X, (2012).

8. HARLOW, F. H. 1963. The particle-in-cell method for numerical solution of prob-lems in fluid dynamics. In Experimental arithmetic, high-speed computations and mathematics, (1963).

9. Boyd, L., and Bridson, R. 2011. MultiFLIP for Energetic Two-Phase Fluid Simula-tion. ACM Trans. Graph. VV, N, Article XXX (Month YYYY), PP pages.

10. Zhu , Y, 2005. Animating Sand as a Fluid . Thesis, (MSc Masters Project).
The University of British Columbia

11. Robert Bridson, (2012). Reference of the C++ Language Library , Available from: http://www.cs.ubc.ca/~rbridson/

12. The C++ Resources Network, (2012). Reference of the C++ Language Library , Available from: http://www.cplusplus.com/reference/
Macey, J. (2012). Ngl graphics library, Available from: http://nccastaff.bournemouth.ac.uk/jmacey/GraphicsLib/index.html.

13. SideFX, (2012). Masterclass | Fluid Solvers from Scratch, Available from: http://www.sidefx.com/index.php?option=com_content&task=view&id=2200&Itemid=166

14. An Introduction to Parallel Programming, Peter Pacheco, Elsevier Inc, 2011.

15. M. Evans and F. Harlow, The particle-in-cell method for hydrodynamics calculations. LA-2139, 1957.

16. Steele, K., Cline, D., Egbert, P. K. and Dinerstein, J. (2004). Modeling and rendering viscous liquids: Research articles, Comput. Animat. Virtual Worlds 15(3-4): 183{192.

17. S. Osher and J. A. Sethian, "Fronts propagating with curvature dependent speed: algorithms based on hamilton-jacobi formulations," *J. Comput. Phys.*, vol. 79, no. 1, pp. 12–49, 1988.

18. C. W. Hirt and B. D. Nichols, "Volume of fluid vof method for the dynamics of free boundaries," *Journal of Computational Physics*, vol. 39, pp. 201–225, January 1981.

19. Wolfram MathWordl, (2012). Runge-Kutta Method, Available from: http://mathworld.wolfram.com/Runge-KuttaMethod.html

20. SideFX, (2012). Import and export geo/bgeo format, Available from: http://www.sidefx.com/docs/houdini12.0/io/formats/geo

21. Numerical Heat Transfer and Fluid Flow, Suhas Patankar, Taylor & Francis, 1980.