LiU-ITN-TEK-A--11/055--SE

PIC/FLIP Fluid Simulation Using A Block-Optimized Grid Data Structure

Fredrik Salomonsson

2011-09-12

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Science and Technology
Linköping University
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings universitet
601 74 Norrköping

LiU-ITN-TEK-A--11/055--SE

PIC/FLIP Fluid Simulation Using A Block-Optimized Grid Data Structure

Examensarbete utfört i medieteknik
vid Tekniska högskolan vid
Linköpings universitet

Fredrik Salomonsson

Examinator Jonas Unger

Norrköping 2011-09-12

Abstract

This thesis work will examin and present how to implement a *Particle-In-Cell* and a *Fluid-Implicit-Particle* (PIC / FLIP) fluid solver that takes advantage of the inherent parallelism of Digital Domain's sparse block-optimized data structure, DB-Grid. The methods offer a hybrid approach between particle and grid based simulation.

This thesis will also discuss and go through different approaches for storing and accessing the data associated with each particle. For dynamically create and remove attributes from the particles, Disney's open source API, Partio is used. Which is also used for saving the particles to disk.

Finally how to expose C++ classes into Python by wrapping everything into a Python module using the Boost.Python API and discuss the benefits of having a script language.

Acknowledgement

I want to thank my supervisor Doug Roble, Creative Director of Software at Digital Domain, for giving me valuable inputs, guidance and discussions. I also want to thank people from the software team at Digital Domain: John Johansson for giving me input on the fluid implementation and helping me out with gathering results. Maria Barot for helping me solve all the mile long compile errors that I came across during the Python wrapping. Blake Sloan for helping me with the pesky linking errors. Kree Cole-McLaughlin for valuable inputs during my code review. I also want to thank my examiner Jonas Unger at Linköping University.

Contents

List of Figures

List of Tables

1 Introduction

Filming a big tsunami can be quite costly and also dangerous to do with practical special effects e.g recreate a tsunami in a water tank. A cheaper and better controlled way is to simulate these effects in a computer. Were the director have full control of the motion and decide from how big the waves shall be to how many.

1.1 Fluids in many shapes

The motion of liquids and gases can be described with the famous *Navier-Stokes* equations (see section 2.5) but solving these has been proven to be difficult. Two different approaches exist on how to solve these equations. One is to solve them using particles to track the motion (*Lagrangian*) and the other is to use fixed points in space to compute the motion (Eulerian). On the Lagrangian side, one method that has shown promise is the *Smoothed particle hydrodynamics* (SPH) see Desbrun et.al. [9] and Müller et.al. [21]. Mostly because it can run in real time using the GPU.

Many of the Eulerian methods are based on Stam's papar Stable Fluids [28], where he presented an unconditionally stable model for self advection, *semi-Lagrangian Advection*, that still could produce complex fluid-like motion. Most of the Eulerian methods are to computational heavy or memory demanding to run in real time. But one method that can run in real time, with the use of a *restricted tall cell grid*, is the method presented by Chentanez and Müller in [24]. The reason for preferring Eulerian, which we will explain further in section 2.1, is the use of a grid that make the numerical approximation easier. Since we evaluate fixed points and not arbitrary, jittered, points in space.

There is also methods which use both the Lagrangian and the Eulerian approach to overcome the different issues these two methods have. The earliest is called *Particle-in-cell* (PIC) and first mentioned by Harlow et.al. in [13]. This hybrid method was then improved by Brackbill and Ruppel in [4] and is called *Fluid-Implicit-Particle* (FLIP).

1.2 Digital Domain

Digital Domain is a special effects company, founded in 1993 and is located in Venice, California. Digital Domain has won three Academy awards for *Best Visual Effects* and three *Scientific and Technical Achievement* awards. Been involved in over 80 films including *Titanic*, *What Dreams May Come True* and *The Curious Case of Benjamin Buttons*. They have an in house Eulerian based fluid simulation engine called *fsim* which earned them one of the *Scientific and Technical Achievement* awards.

1.3 Motivation

Over the last few years a lot of tools have been developed for level sets and fluid simulations at Digital Domain: a new sparse block-optimized grid data structure *Dynamic-Blocked-Grid*, DB-Grid by Museth [22] and [23]. Johansson [15] implemented a *Particle Level Set* for DB-Grid thus introduce a lot of tools for handling particles and storing particles in DB-Grid.

The tools have then been ported to SideFX's Houdini as *surface operators* (SOPs). These SOPs can be connected into a network where each SOP manipulates the data flowing through it. One issue with this is that it is hard to put together a fluid solver inside the program with the use of these

SOPs. Since Houdini makes it impossible to take smaller time steps than one frame throughout the whole network. Each node can take smaller time steps, but only locally for that specific node.

For example moving a car between two points, x_0 and x_1 for a time period of t is split in to two parts; one function that moves the car forward and the other rotates the car. To avoid crashing the car we divide t into smaller time steps Δt and move and rotate the car in each iteration to keep the car on track. If these two functions were implemented in Houdini as SOPs and t represent one frame. Executing this would result in that first SOP would move the car straight forward during the time period t internally taking sub time steps each iteration. When it is finished the second SOP would rotate the car under a time period of t. The result would be that the car would drive straight forward (most likely crash) and then, when it stopped, start to spin.

Therefore the company wants a script language to be able to glue these parts together and form a fluid solver with the use of the new grid structure. Also to implement a hybrid grid and particle fluid solver since a lot of the tools already existed for handling particles [15].

1.4 Aim

The first step is to implement a script language support which enables us to quickly test the different parts, we will wrap/expose all classes that uses the data structure DB-Grid with the use of the API Boost.Python [18]. Thereafter implement a PIC/FLIP based fluid solver in C++ and evaluate how it performs. The aim of this implementation is to get a working PIC/FLIP fluid solver and to be able to set it up using the script language Python. The fluid solver should be able to use the block-optimized grid data structure efficiently and be able to run on multiple threads. The C++ implementation and the Python module should have similar interfaces to make it easier to translate scripts back to C++ or vice versa.

1.5 Structure of the report

This report is structured as follows:

- Section 2 will give the theory we need to be able to implement the PIC/FLIP, a short describtion of the data structure DB-Grid and the script language Python.
- Section 3 will go through in detail our implementation.
- Section 4 will present the result of our tests we have done with our implementation, with both tables and figures.
- Section 5 will discuss the result we got in the previous section.
- Section 6 will concludes this thesis and also list what can be improved with our implementation.

2 Background

This section gives an overview of the background needed to understand the rest of this thesis, introduce key terms and conceptions that will be used through out this report.

2.1 Lagrangian and Eulerian

When working with continuum spatial functions in a computer that can only represent discrete values we must be able to track these functions over time. The two main approaches of tracking these are the Lagrangian viewpoint and the Eulerian viewpoint. The Lagrangian viewpoint is the most

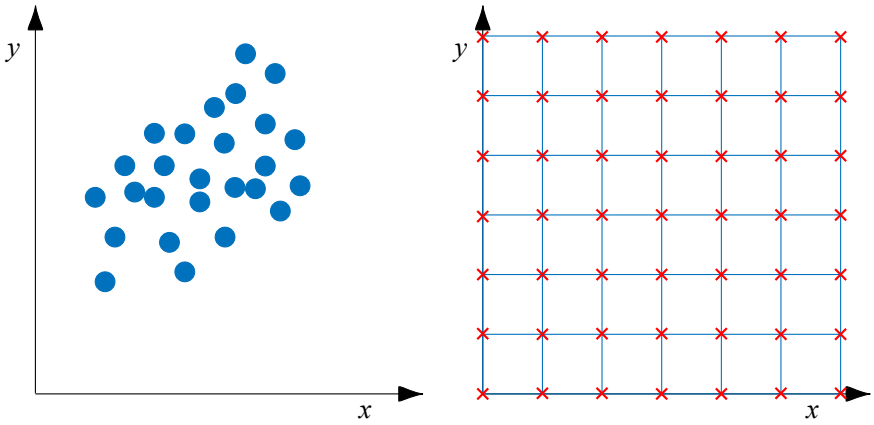


Figure 1: The Lagrangian approach to the left and the Eulerian to the right

widely used approach that tracks points in space, where each point store a quantity q (it can be velocity, density, temperature etc). The benefit of this approach is that the whole domain does not need to be stored, thus reducing storage cost and the points can be tracked fairly accurate. Downsides of this approach are that it is hard to keep track of the relation between the points and also hard to analytically work with the spatial derivatives as well as estimate them. The Particle system and the polygon mesh are the two most common methods that use the Lagrangian viewpoint.

The Eulerian viewpoint takes a different approach, instead of having free points floating around and tracking the spatial function. It looks at fixed points in space, which is sampled with a fixed length Δx , Δy and Δz between the points. And measure how the quantity q changes over time. The advantages and disadvantages with this approach are the opposite of the Lagrangian. Since the points are fixed in space the spatial derivatives are easier to handle analytically as well as estimate them. But it suffers from numerical dissipation and since the whole domain is sampled, it has a large memory footprint. The Eulerian approach is most commonly used for fluid simulation just because it handles the spatial derivatives so nicely.

2.2 Staggered and Collocated Grid

There are two different kinds of grid structures that are used in this thesis when discretizing the space. The first, and rather common, is the *collocated* grid i.e. all of the different variables are stored at the center of each cell in the grid. Also worth mentioning is that a cell center in DB-Grid is equal

to a sample point in the grid. The cell is the volume/area between the sample point and its positive neighbors. This differs from what is shown for example in [5] where the cell center is in the middle of the cell. It is mostly used for level sets, scalar and particle fields. It can be used for the velocity field as well but due to the *central differences* are inaccurate, it is not recommended. Take the 1 dimensional example Bridson gives in [5] e.g. a function $f(i) = -1^i$ is sampled at each grid point q_i and the derivative of this quantity is extimated using the first central difference scheme:

$$\left(\frac{\delta q}{\delta x}\right)_i \approx \frac{q_{i+1} - q_{i-1}}{2\Delta x} \tag{1}$$

Even if this formula is unbiased and accurate to $O(\Delta x^2)$ it has one flaw and that is it ignore the sample point q_i, which is being evaluated. In this case it will give that $\frac{\delta q}{\delta x} = 0$, which is not correct. Other schemes that can be used is back and forward difference :

$$\left(\frac{\delta q}{\delta x}\right)_i \approx \frac{q_{i+1} - q_i}{\Delta x} \tag{2}$$

but they are biased to either left or right and are only accurate to $O(\Delta x)$.

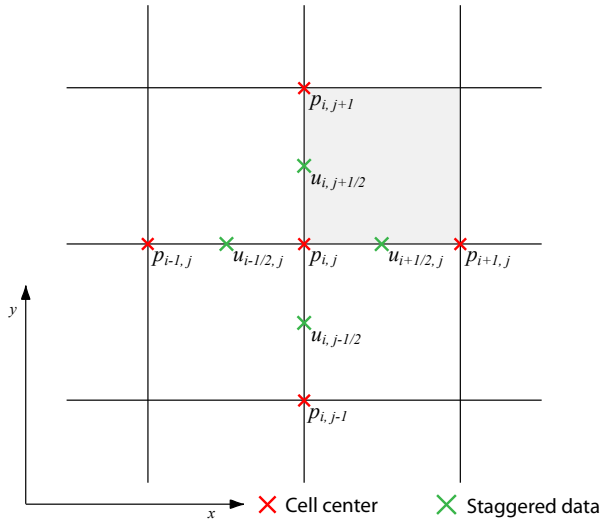


Figure 2: The two dimensional staggered grid (MAC). Gray area indicates the cell area that belongs to $p_{i,j}$

To fix this issue we use a *staggered* grid, also know as a MAC (*marker-and-cell*) grid, it was introduced by Harlow et. al. in [13]. Where the different variables are stored at different locations on the grid. Figure 2 shows a 2 dimensional staggered grid for the velocity and pressure, where each component of the velocity is sampled at its Cartesian counterpart and the pressure $p_{i,j}$ is sampled at the cell center.

To go back to the example mention earlier, the derivative of the quantity q can be solved by using a staggered grid and sample q's half-way points instead, $q_{i+1/2}$. The derivative can then be estimated by using this formula:

$$\left(\frac{\delta q}{\delta x}\right)_i \approx \frac{q_{i+1/2} - q_{i-1/2}}{\Delta x} \tag{3}$$

Equation 3 is unbiased and has the same accuracy as equation 1 but lacks the issue where the point it evaluates is skipped and does not creates what is called a non trivial *null-space* [5].

The staggered grid can be a little confusing to handle in some cases. One is the half indices $i + 1/2$ that are used since they are convenient theoretically and conventionally. For the implementation the half indices are implemented as integers. We use three grids that are offset in the normal direction of the component i.e for the u component of the velocity, the grid is offset $(-0.5, 0, 0)$ and has the dimension $(w+1) \times h \times d$. The v component's grid is offset by $(0, -0.5, 0)$ and has the dimensions $w \times (h + 1) \times d$. The last component w's grid is offset by $(0, 0, -0.5)$ and has the dimension $w \times h \times (d + 1)$.

If we want to evaluate the whole velocity at a specific position we always need to do some sort of interpolation to get that value, even when accessing a grid point. If we want an arbitrary position on the grid we can perform a bilinear or trilinear interpolation, for each component, to get the vector we are looking for. As for the grid points, it is a bit easier, it boils down to some simple averaging. In three dimensions these averages are:

$$\vec{u}_{i,j,k} = \left(\frac{u_{i-1/2,i,k} + u_{i+1/2,i,k}}{2}, \frac{v_{i-1/2,i,k} + v_{i+1/2,i,k}}{2}, \frac{w_{i-1/2,i,k} + w_{i+1/2,i,k}}{2} \right) \tag{4a}$$

$$\vec{u}_{i+1/2,j,k} = \left(u_{i+1/2,j,k}, \frac{\begin{array}{c} v_{i,j-1/2,k} + v_{i,j+1/2,k} \\ +v_{i+1,j-1/2,k} + v_{i+1,j+1/2,k} \end{array}}{4}, \frac{\begin{array}{c} w_{i,j,k-1/2} + w_{i,j,k+1/2} \\ +w_{i+1,j,k-1/2} + w_{i+1,j,k+1/2} \end{array}}{4} \right) \tag{4b}$$

$$\vec{u}_{i,j+1/2,k} = \left(\frac{\begin{array}{c} u_{i-1/2,j,k} + u_{i+1/2,j,k} \\ +u_{i-1/2,j+1,k} + u_{i+1/2,j+1,k} \end{array}}{4}, v_{i,j+1/2,k}, \frac{\begin{array}{c} w_{i,j,k-1/2} + u_{ix,j,k-1/2} \\ +u_{i,j+1,k-1/2} + u_{i,j+1,k+1/2} \end{array}}{4} \right) \tag{4c}$$

$$\vec{u}_{i,j,k+1/2} = \left(\begin{array}{c} u_{i-1/2,j,k} + u_{i+1/2,j,k} \\ +u_{i-1/2,j,k+1} + u_{i+1/2,j,k+1} \end{array}, \frac{\begin{array}{c} v_{i,j-1/2,k} + v_{i,j+1/2,k} \\ +v_{i,j-1/2,k+1} + v_{i,j+1/2,k+1} \end{array}}{4}, w_{i,j,k+1/2} \right) \tag{4d}$$

To see the 2-dimensional formulas please see [5] page 24.

2.3 Level Set

There are two ways of describing an object, explicitly describe it: e.g. by the use of a mesh or implicitly with the use of a function. The implicit function can be sampled on to a grid to form a level set. For convince the implicit surface (iso-contour) $\phi_{i,j,k}$ is defined at $\phi_{i,j,k} = 0$ since this makes it easy too define the inside and the outside of the object. This is useful for fluid simulation where the level set define the fluid surface, the cells on the grid can easily classified as empty or fluid by just checking the sign.

$$\begin{array}{lcl} \phi_{i,j,k} = 0 & : & \text{on the surface,} \\ \phi_{i,j,k} > 0 & : & \text{outside,} \\ \phi_{i,j,k} < 0 & : & \text{inside.} \end{array} \tag{5}$$

Most commonly used function for the level set is the *signed distance function*. It gives the distance to the closest point on the surface whereas the sign define if the current point is outside or inside. For a given surface S the *distance function* is defined as:

$$distance_S(\vec{x}) = min_{\vec{p} \in S} \|\vec{x} - \vec{p}\| \tag{6}$$

Combining equation 5 and 6 gives the *signed distance function*:

$$\phi(\vec{x}) = \begin{cases} distance_S(\vec{x}) & : \quad \vec{x} \text{ is outside,} \\ -distance_S(\vec{x}) & : \quad \vec{x} \text{ is inside.} \end{cases} \tag{7}$$

One of the great feature with the sign distance function is that if a point on the inside of the surface move along the normal $\hat{n}$, the closest point wont change and the distance to it depends only how much the point move in that direction. This gives that the directional derivative is 1:

$$\nabla\phi \cdot \hat{n} = 1 \tag{8}$$

It also the quickest way to the surface hence if the point move in any other direction, ϕ wont change any faster, meaning that $\nabla\phi$ is the same as $\hat{n}$:

$$\nabla\phi = \hat{n} \tag{9}$$

and putting the two equations together gives the following equation:

$$||\nabla\phi|| = 1 \tag{10}$$

which is called the Eikonal equation. Since the level set only has values at the center of each cell a trilinear or bilinear (depending on if 3 or 2 dimensions are used) interpolation is needed to get values between these points. See Johansson [15] for a good explanation on how trilinear interpolation is done. To get further information about the level set we recommend to read the book by Osher and Fedkiw [26], since this is only a brief overview of the level set.

2.3.1 Narrow band method

Storing a fully sampled level set using a grid takes a lot of memory, therefore research has been done on how to reduce this part. One way is to only store a part of the level set, namely a thin band around the surface. The band is defined between $[-\gamma, \gamma]$ where γ is the distance to the surface. This can be expressed as:

$$|\phi_{i,j,k}| < \gamma \tag{11}$$

This method is called the narrow band method and gives a sparse representation of the level set that perfectly suits the volume data structure used by Digital Domain.

2.4 DB-Grid

One of the issues that plagues the Eulerian approach is the high memory footprint when sampling the whole domain. To reduce this issue many efficient volume data structure has been developed, one of them is Digital Domain's DB-Grid. Which Johansson in [15] showed that it is slightly faster than Sony Imageworks open source volume data structure Field3D [35]. The DB-Grid was developed by Museth when he was at Digital Domain and a

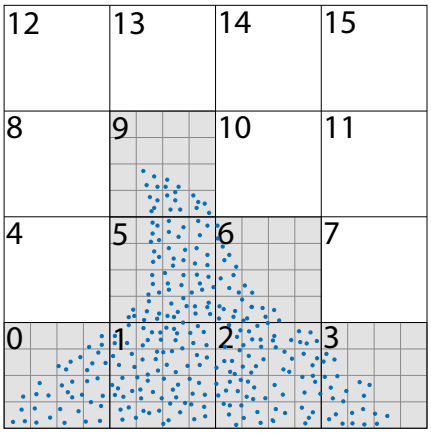


Figure 3: A two dimension example of a particle field using the DB-Grid data structure. Gray indicates that the block is dirty. A similar example can be seen for the level set in [15]

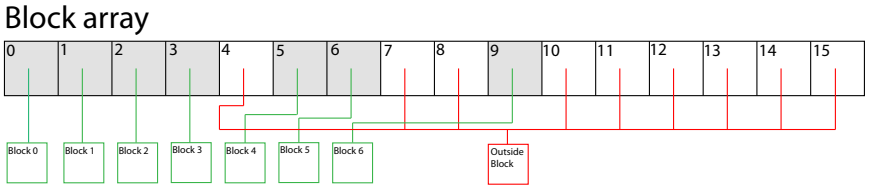


Figure 4: Shows the underlying structure of DB-Grid. Green means that it points to an allocated block, red indicates that it points to the outside block i.e empty.

short description can be seen in [22] and [23]. It is design to handle sparse volume data, where the grid is divided into blocks of elements.

The equally sized blocks has the size in power of two, usually eight or sixteen. The Blocks are classified into three different types, *Dirty block*, *Outside block* and *Inside block* where the last is used if the data is a level set. When a volume is loaded into the DB-Grid all the blocks that contains any data are allocated and the empty ones points to the same *Outside block*. Hence reducing the memory footprint significantly. A more detail description about the structure of DB-Grid can be found by Johansson in [15].

Other advantages with DB-Grid, besides the sparse data structure and its fast access time to the data, is its flexibility: it can handle a lot of different types of volume data. Not only level sets, as it was design to handle from the beginning, but also scalar fields, velocity field and particle fields as well. In this thesis we use this blocked optimized data structure to implement the PIC and FLIP method.

2.5 Eulerian Fluid

Fluid simulation in visual effects took a huge leap forward when Stam presented his stable fluid in [28]. Before that it was belived that physcial fluid models was to expensive to simulate, mostly due to that models was based on unstable schemes. But with the stable fluid it was now possible to create

intressting swirling fluids within resonable time.

The heart of every fluid simulation lies the famous *incompressible Navier-Stokes equations*:

$$\frac{\delta \vec{u}}{\delta t} = -(\vec{u} \cdot \nabla)\,\vec{u} - \frac{1}{\rho}\nabla p + \nu \nabla \cdot \nabla \vec{u} + \vec{f}, \tag{12a}$$

$$\nabla \cdot \vec{u} = 0 \tag{12b}$$

where $\vec{u}$ is the velocity, ν denotes the *kinematic viscosity* of the fluid, ρ is the *density*, p is the pressure and f is an *external force*. The symbol ∇ is the vector of spatial partial derivatives, $\nabla = \left(\frac{\delta}{\delta x}, \frac{\delta}{\delta y}, \frac{\delta}{\delta z}\right)$ in three-dimensions. The Cartesian coordinate system in this thesis is defined as the $y-$axis is pointing vertically upwards and $x-$ and $z-$axis are horizontal.

The first differential equation (12a), in the Navier-Stokes equations, is called the *momentum equation* and describes how the fluid accelerates due to the forces acting on it. It is actual three differential equation wrapped up in one vector equation, corresponding Newton's second law of motion $\vec{F} = m\vec{a}$ and for further explanation on this please see [5].

The second differential equation (12b), is called the *incompressibility condition* and ensure that the fluid's volume does not change i.e stays incompressible. Even if real life fluids actually do change their volume, otherwise it would be impossible to hear underwater, but the change in volume is hardly noticeable. To accommodate for this in fluid simulations, what is called *compressible fluids*, is both complicated, expensive and does not affect the visually that much; apart from extreme cases like sonic booms and shock waves. Therefore it is easier to see all fluids, gases included, as incompressible.

To get a better understanding of the equations we will briefly describe the different parts. The term containing the kinematic viscosity, $\nu \nabla \cdot \nabla \vec{u}$, is called diffusion term [28] or viscosity term. It describes how viscous the fluid is and can be seen as an internal friction e.g. honey has high viscosity where as water has low viscosity. The $\nabla \cdot \nabla$ can sometimes be shorted to ∇^2.

The external force f contains all of the forces acting on the fluid which is not generated by the fluid itself. One of them is gravity g and is usually defined as: $(0, -9.81, 0)\, m/s^2$ but is depending on the location. Other forces can also be added to f in order to control the fluid and make it behave in a certain way.

$(\vec{u} \cdot \nabla)\,\vec{u}$ is called the *advection* term, and this nonlinear term is responsible for moving the fluid with itself. It is important for fluid-like behavior such as swirls and vortices.

$\frac{1}{\rho}\nabla p$ along with equation 12b is responsible for maintaining the *incompressibility* of the solution and this is solved in a step called Projection, which will be explained further down.

2.5.1 Splitting

The Navier-Stokes equations are defined in continuous form hence we need a way to discretize them in order to numerically simulate them using the computer. One method which is popular and works well with computer graphics is the *splitting* method.

This method works, as the name applies, by splitting a complicated equation into is component parts, and solve each component separately in turn. The order in which these parts are solved is highly crucial to correctly solve the overall equation.

The Navier-Stokes equations are split up into four parts; External forces, Advection, Diffusion (Viscosity) and Projection. Where the field has been evaluated at time t and we want to compute the field at a later time $t + \Delta t$. The initial state is $u_0 = u(x, 0)$. Stam in [28] suggest to solve the splitting in this order:

$$w_0(x) \xrightarrow{\text{add force}} w_1(x) \xrightarrow{\text{advect}} w_2(x) \xrightarrow{\text{diffuse}} w_3(x) \xrightarrow{\text{project}} w_4(x) \tag{13}$$

where $w_0(x) = u(x, t)$ and the solution at desired time $t + \Delta t$ is the last velocity field, $w_4(x) = u(x, t + \Delta t)$.

2.5.2 Euler equations

There are only some rare cases where viscosity is important, e.g. when simulating a high viscous fluid such as honey. But most of the time it is not as important and therefore it is often dropped to simplify the equation. Also the error of estimating the equations also introduce what can physically interpret as viscosity, see section 3.2 for reducing this. Without the viscosity term we still get a fluid like behavior. The Navier-Stokes equations without the viscosity term are called the *Euler equations* and the fluids they describe are called *inviscid*: An ideal fluid without no viscosity. The Euler equations can be written on a very compact form using the material derivative.

$$\frac{D\vec{u}}{Dt} = -\frac{1}{\rho}\nabla p + \vec{f}, \tag{14a}$$

$$\nabla \cdot \vec{u} = 0 \tag{14b}$$

The material derivative describes how a field is advected by a velocity field. In this case the velocity field $\vec{u}$ is advected with itself:

$$\frac{D\vec{u}}{Dt} = \frac{\delta\vec{u}}{\delta t} + \vec{u} \cdot \nabla\vec{u}, \tag{15}$$

thus it is sometimes called *self-advection*. When dropping the viscosity term the layout of the splitting becomes:

$$w_0(x) \xrightarrow{\text{add force}} w_1(x) \xrightarrow{\text{advect}} w_2(x) \xrightarrow{\text{project}} w_4(x) \tag{16}$$

2.5.3 External forces

The easiest part to solve for is the external force f. This is solved for with a simple forward Euler time integration:

$$w_1(x) = w_0(x) + \Delta t f(x, t) \tag{17}$$

Where the external force f is assumed to not vary considerably during the time step. This can of course be solved with an higher order time integration schemes such as *Total Variation Diminishing (TVD) Runge-Kutta*, which takes half steps to accurately estimate the position. But in most cases, the simple and easy to implement, forward Euler is sufficed. A detail description of these two can be found by Johansson in [15].

2.5.4 Advection

The advection term $-(\vec{u} \cdot \nabla)\,\vec{u}$ describes, as mention earlier, how the velocity field is moved by itself. It is the term that makes the Navier-Stokes equations nonlinear, and also the term that used a unstable solver before

Stam propose to solve it with semi-Lagrangian. Which is *unconventionally stable* due to the fact that the bi- or trilinear interpolation, which semi-Lagrangian use, guarantees that the velocity computed cannot be higher than the velocities in the field. Stam solution to the advection step is based on a technique used for solving partial differential equations known as *method of characteristics*.

For each time step, fluid particles are transported by the velocity field of the fluid itself. Thus updating the velocity at point x at time $t + \Delta t$ is to backtrace the point through the velocity field w_1 over time Δt. This defines a path to the velocity, a particle streamline $p(x, s)$. Therefore the velocity, which we seek, is taken from point x at its previous location at Δt ago. Of course no actual particles are created, they are strictly imaginary hence its name semi-Lagrangian.

$$w_2(x) = w_1(p(x, -\Delta t)) \tag{18}$$

For computing the particle streamline $p(x, s)$ we know that point x start at position $\vec{x}_p$ and ends up at grid position $\vec{x}_G$. We want to update $\vec{x}_G$'s velocity $\vec{q}_G^{n+1}$. We know that the imaginary particle x starts at position $\vec{x}_p$ with the velocity $\vec{q}_p^n$, flows along the velocity field and after time Δt ends up at position $\vec{x}_G$. Which gives that $\vec{q}_G^{n+1} = \vec{q}_p^n$. Therefore we only need to know $\vec{q}_p^n$.

From the start position $\vec{x}_p$, the imaginary particle x moves according to this differential equation:

$$\frac{d\vec{x}}{dt} = \vec{u}(\vec{x}) \tag{19}$$

and ends up at position $\vec{x}_G$ at time Δt. When we trace this backwards we assume that we start at the end position and moves with the reverse velocity field $-\vec{u}$. This can be solved using a forward Euler:

$$\vec{x}_p = \vec{x}_G - \Delta t \vec{u}(\vec{x}_G) \tag{20}$$

To get a better result Bridson [5] suggest to use a higher-order time integration such as the TVD Runge-Kutta.

When the position $\vec{x}_p$ is found, we need to find its velocity $\vec{q}_p^n$. And most likely the particle is not on any grid point, therefore must be interpolated to get the velocity. Most common to use is the Trilinear or Bilinear, depending if three or two dimension are used. As mention above it does not over shoot i.e. do not introduce extra energy to the velocity field, which cause the solution to become unstable. But it suffers from numerical dissipation that causes the fluid the seem to be more viscous than it should be, due to the linear interpolation average the velocities from the neighboring grid points.

To improve this a more accurate interpolation can be used, Bridson mention *Catmull-Rom* interpolation. Which is to fit an Hermite spline to the points. It can cause over- and undershoots therefore the result needs to be clamped. More information can be found in Bridson's book [5].

2.5.5 Diffusion

Solving the next step; the viscosity of the fluid, Stam suggest to setup the viscosity as a diffusion equation

$$\frac{\delta w_2}{\delta t} = \nu \nabla^2 w_2 \tag{21}$$

and since this equation is well known, there exist a lot of numerical procedures for solving it. One way of solving equation 21 is to discretize the

diffusion operator ∇^2 and take a explicit time step. But this procedure is unstable if the viscosity is large therefore a better way that Stam suggest is to solve the implicit form of this equation.

$$\left(\mathbf{I} - \nu\Delta t\nabla^2\right) w_3(x) = w_2(x) \tag{22}$$

where $\mathbf{I}$ is the identity matrix. Thus resulting in a sparse linear system for the unknown velocity field w_3 when the diffusion operator is discretized. And a sparse linear system can be solved efficiently with e.g. *Preconditioned Conjugate Gradient* algorithm, see Bridson [5] for more detail.

2.5.6 Projection

The projection step is the step that makes the fluid incompressible and at the same time enforce the boundary conditions; preventing the fluid of leaking into solid objects and walls. A vector field that satisfied the incompressibility condition is *divergence-free* i.e. contains no sinks or sources. Therefore to make the fluid incompressible is the same as making the velocity field divergence free. Stam uses a mathematical result called *Helmholtz-Hodge Decomposition* which means that any vector field, in our case w_3, can be decomposed into a divergence free part u and a curl free part ∇q:

$$w_3 = u + \nabla q \tag{23}$$

We can define a projection operator P that simply project w_3 onto its divergence free part $u = Pw_3$ by applying the divergence operator $\nabla\cdot$ to all terms of equation 23

$$\nabla w_3 = \nabla u + \nabla \cdot \nabla q \overset{\nabla u=0}{\Rightarrow} \nabla w_3 = \nabla^2 q \tag{24}$$

This equation is called a *Possion equation* and its solution is the scalar field q. It is used to compute the divergence free part of w_3, namely u which we are looking for.

$$u = Pw_3 = w_3 - \nabla q \tag{25}$$

The Possion equation can be solved approximately by discretizing each term, forming a sparse linear system. And use the *Preconditioned Conjugate Gradient* algorithm to find the solution q. But there is one other thing that we must consider and that is boundary conditions.

2.5.7 Boundary conditions

One of the boundary conditions, called *Dirichlet* boundary condition, states that there can be no flow into or out from a boundary surface (Solid object or walls of the grid) and can be described as follows:

$$\vec{u} \cdot \hat{n} = \vec{u}_{solid} \cdot \hat{n} \tag{26}$$

where $\vec{u}_{solid}$ describes the velocity of the boundary surface. This equation simply kills the velocity (for fluid cells adjacent to a solid object or wall) in the normal direction of the boundary surface $\hat{n}$ but leaves the tangent velocity, thus allows the fluid to flow alongside the surface.

The other boundary condition, called *Neumann* boundary condition, is used when solving the Possion equation 24 and is defined as:

$$\frac{\delta\vec{u}}{\delta\hat{n}} = 0 \tag{27}$$

describing the normal pressure derivative, and states that there shall be no change of flow along the normal $\hat{n}$ of a boundary surface. Both of these boundary conditions are important when solving the projection step. When the solution for q is found we can simply enforce the velocity field to be incompressible by projecting w_3, see equation 25 which gives:

$$w_4 = w_3 - \nabla q \tag{28}$$

2.5.8 Issue with Viscosity

As seen above the Projection step will also result in a sparse linear system, thus these two steps (Projection and Viscosity) can be combined. So called solving the *Stokes problem* but since it renders the solver more complex, a lot of people tends to avoid this approach and solve the viscosity separately, Bridson dedicates a whole chapter for this in [5]. One problem with solving the viscosity separately is that it needs a divergence free velocity field which the advection step also needs. One solution is to apply the Projection step after the advection step and after the diffusion step.

$$w_0(x) \xrightarrow{\text{add force}} w_1(x) \xrightarrow{\text{advect}} w_2'(x) \xrightarrow{\text{project}} w_2(x) \xrightarrow{\text{diffuse}} w_3(x) \xrightarrow{\text{project}} w_4(x) \tag{29}$$

but will result in having lower quality than other solutions.

2.6 Python

Python is a high-level scripting language and instead of needing to be compiled like C++ and Java, it uses a interpreter i.e the Python interpreter interpret the source code into an more efficient code for the computer and immediately execute the code, hence the name interpreted language. [32]. Python runs on Windows, Linux and Mac OS X which the later two supports it natively.

Since Python does not have any compiling step, the edit-test-debug cycle is much faster than example C++, where you need to recompile before each test. Therefore it is great for e.g. testing parameters for Improved Blobbies or testing to switch the order of the splitting method for the fluid solver and see what it does. The ability for wrapping C and C++ code into modules with Boost.Python [18] (more on this later in section 3.6). That and most of Digital Domain's Technical Directors are used to Python was the reason for us to chose Python for our scripting language to the level set and fluid framework.

Python started as a hobby project over Christmas 1989 by Guido van Rossum when he work at *National Research Institute for Mathematics and Computer Science* (CWI) in the Netherlands developing a programming language aimed at non-programmers called ABC.

> It all started with ABC, a wonderful teaching language that I had helped create in the early eighties. It was an incredibly elegant and powerful language, aimed at non-professional programmers. Despite all its elegance and power and the availability of a free implementation, ABC never became popular in the Unix/C world. I can only speculate about the reasons, but here's a likely one: the difficulty of adding new "primitive" operations to ABC. It was a monolithic, "closed system", with only the most basic I/O operations: read a string from the console,

> write a string to the console. I decided not repeat this mistake in Python. - Van Rossum [31]

Python has then grown in popularity and today is widely used in the world, at [32] there is a list of companies that use this script language. For the examples that follows we use Python 2.7.2.

2.6.1 Basics

The variables in Python is not locked to a specific type but is determine at run time. Therefore it is perfectly valid to write this in a Python Interpreter:

```
>>> # This line will be ignored by the interpreter
>>> foo = 1
1
>>> foo += 2
3
>>> foo = "Hello"
Hello
```

The structure of Python builds on indentation where functions, classes and loops are indented to show what is contained inside. Similar to C++ "{" and "}" but instead a indentation is used. For example a function that simply multiply the argument passed to it by two and return the product. Will look something like this if it is defined in C++ :

```
int foobar(int x)
{
   // multiply the argument by 2
   return x*2;
}
```

if it is defined in Python it will look like this:

```
def foobar(x):
   # multiply the argument by 2
   return x*2
```

where *def* tells the Python interpreter that a function is defined and the following word is the name of the function, in this case `foobar`. All the argument of a function is defined inside the parenthesis, without mention of what type each parameter has, just the name. The type is decided by the Python interpretor at run time thus is fully possible to pass a string as an argument to `foobar` while the C++ version of `foobar` is strictly assign to only handle integers. To make this even more complicated let us write a program that compute a sum. In C++:

```
int foobar(const int &start,const int &stop)
{
   int sum = 0;
   // Accumulate sum
   for(int x =start; x <  stop; x++)
   {
  sum += x;
   }
   return sum;
```

```
}
```

In Python this will be a little less complex:

```
def foobar(start, stop):
    sum = 0
    # accumulate sum
   for x in range(start, stop):
        sum += x
    return sum
```

In this example we use a *for loop* and compare to the C++ version Python's for loop is a little easier to read for a person not so familiar to a program language. It follows the same syntax as for *def*. Behind the scene the for loop unravels to an while loop that uses an iterator to iterate through the list which the range function builds. The dereference of the iterator is the variable *x*.

2.6.2 Classes

Python is an object oriented language and therefore also handles classes. It has similar syntax as for the *def* and the *for loop*. To show the difference with C++ here is an example of a simple class implemented in both C++ and Python:

```
// C++:
class Foo
{
   private:
     int my_spam,
         my_egg;
  public:
    Foo(const int &spam, const int &egg)
  {
            my_spam = spam;
     my_egg = egg;
  }
   int getSpam() { return my_spam; }
   int getEgg() { return my_egg; } const
};
```

```
# Python:
class PyFoo:
  def __init__(self, spam, egg):
     self._spam = spam
     self._egg = egg
  def getSpam(self):
     return self._spam
  def getEgg(self):
     return self._egg
```

The C++ class Foo has two private data members, `spam` and `egg`. But in Python; "Private" instance variables that can only be access inside the class object does not exist. Therefore a "_" in front of a variable is used to indicate that it is "Private". In the example above we can see that

`PyFoo` has two "private" variables (`_spam` and `_egg`) same as for Foo. The function named `__init__` corresponds to the constructor in C++, i.e the function which is called when the object is created. One thing that differs from `Foo` to `PyFoo` is the variable `self`. As the name applies it is the object itself that is passed when calling a certain method from `PyFoo`, similar to C++'s *this pointer*, e.g:

```
>>> # Python Interpreter
>>> foo = PyFoo(1,2)
>>> foo.getSpam()
1
>>> # is the same as
>>> PyFoo.getSpam(foo)
1
```

2.6.3 Modules

All of this can be written in the Python interpreter but when exiting all of the functions and classes we define are lost. Therefore a better way when writing longer programs is to use a text editor. Prepare all the inputs in a file and then run it through the interpreter, this is called a *script*. As the program gets longer we might want to divide the script into multiple files for better maintenance. We also might have a function defined in one program and want to use it with another but want to avoid copy/paste that definition to the other program. Python solution to this is to use the definition and statements in a file to either execute as a script or use them in a interactive instance of the interpreter. Such a file is called a *module* and has the file extension ".py". Similar to the header file of a C++ program. These modules can be imported into other modules or into the main module. For example we create an module with our previous defined function `foobar` and our class `PyFoo` in a file called `bar.py`:

```
# Module bar

def foobar(x):
   # multiply the argument by 2
   return x*2

class PyFoo:
  def __init__(self, spam, egg):
     self._spam = spam
     self._egg = egg
  def getSpam(self):
     return self._spam
  def getEgg(self):
     return self._egg

# create an extra function
def foobar2(x):
   # divides the argument by 2
   return x/2
```

To use these method we can then simply open a Python interpreter and import bar.

```
>>> import bar
>>> foo = bar.PyFoo(1,3)
>>> foo.getSpam()
1
>>> bar.foobar( foo.getSpam() )
2
```

The import calls does not import all the method into current symbol table just the module `bar`. Therefore we must use the module name to access the methods. And if we use a function often we can assign that function call to a local variable.

```
>>> # function calls in Python
>>> # can also be stored in variables
>>> spam = foo.getSpam
>>> spam
<bound method PyFoo.getSpam of
<__main__.PyFoo instance at 0x0000000002AAF108>>
>>> spam()
1
>>> # therefore to simplify the call above
>>> # we can do this
>>> foobar = bar.foobar
>>> foobar( spam() )
2
```

If we want to import a certain function or class directly into the current symbol table we can do this.

```
>>> from bar import foobar, PyFoo
>>> foo = PyFoo(1,3)
>>> foobar( foo.getSpam() )
2
```

or if we want to import all methods:

```
>>> from bar import *
```

But since this cause poorly readable code it is usually avoided. To get more info on how to use Python we recommend to read the documentations/tutorials at [10]

3 Implementation

In this section we will describe how we implemented the PIC/FLIP fluid solver and the methods which we need before we can implement it. We will also describe how to create a surface from a set of particles. And how to wrap everything into Python.

3.1 Particle Field

The particle field used in this implementation uses the particle container that Johansson presented in [15], which uses multiple arrays: one for each element in the grid to store particles that are within the proximity of its grid cell. Johansson suggested different C++ Standard Library (STL) containers *Vector*, *List* and *Deque* (Double Ended Que) to store the particles. We use the Deque container since it is memory coherent and perfect to store small number of objects like particles.

> The *Deque* is useful when storing a small number of objects and you want the ability to quickly allocate space for a couple of more objects. It will provide memory coherency because each block of elements is allocated together in memory. - Johansson [15]

Because the particles are stored on the grid, all the spatial information are maintained. For example looking up neighboring particles can be done quickly without any need of going through all the particles or building a search tree. Which would be the case if all is stored in a single array, see Disney's Partio [29]. The downside of using this data structure, as Johansson mentions, is iterating over all particles. Because we have to iterate over all the dirty grid cells, and for each dirty cell loop over its particle container to get all particles.

3.1.1 Class Interface

Apart from the methods suggest by Johansson for the particle container we also suggest to include following methods:

- `Save( filepath )`
- `Load ( filepath )`
- `Particle Local to World`
- `Particle World to Local`

The ability to save particle to disk is necessary for the fluid simulation but also helps a lot when debugging. To be able to see the particles can make a huge difference than just printing the coordinates on the screen. The position of the particles is stored in local cell coordinates, range $[0, 1]$ for each axis as in [15]. But since the cell point is defined at $(0, 0, 0)$ of the cell space and not in the center of the cell, the transfer from particle field to velocity field will be biased when using a staggered field, see figure 5. Therefore to better suit the staggered field we use an offset when transforming from local particle coordinates $\vec{x}_p$ to world coordinates $\vec{x}$. Hence the equation suggested by Johansson:

$$\vec{x} = \vec{I} + \vec{x}_p \Delta x \tag{30}$$

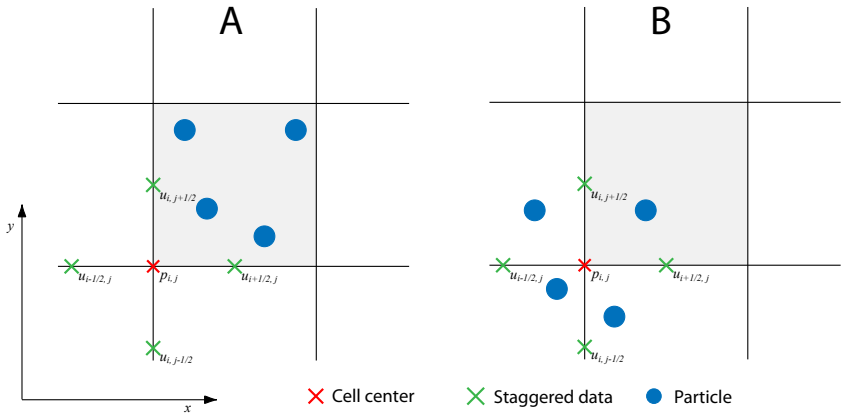


Figure 5: A: Particles no offset applied making the transfer to the staggered points to be biased. B: Particles with offset, better suits the staggered grid.

where $\vec{I}$ is the world coordinates of the cell will be slightly different:

$$\vec{x} = \vec{I} + (\vec{x}_p + \vec{o})\Delta x \tag{31}$$

where the offset $\vec{o} = (-0.5, -0.5, -0.5)$. The reason for not changing the range of the local coordinates to $[-0.5, 0.5]$ is that it would break the comparability with other functions that expects the range $[0, 1]$ e.g. move particle in [15] that only moves particles which is outside the range. One issue with these equations is that the rotation of the grid is ignored and to include this the addition between grid cell and particle position must be compute before transforming the grid cell to world coordinates:

$$\vec{x} = I(\vec{i} + (\vec{x}_p + \vec{o})\Delta x) \tag{32}$$

3.1.2 Partio

We use Walt Disney Animation Studio's open source C++ library Partio for writing to disk since it supports the most common particle formats out there (GEO, BGEO, PTC, PDB, PDA). Partio also supports reading and manipulating particles. It is licensed under the BSD license.

Partio has three different classes for handling the data: `ParticlesInfo`, `ParticlesData` and `ParticlesDataMutable` and each has different access to the data:

- ParticlesInfo: Only access basic info about the particles, attributes name, type etc.
- ParticlesData: Access the data but read only.
- ParticlesDataMutable: Full access of the data (Read/Write)

To get an attribute of a particle from Partio, the type of the attribute (`INT`, `FLOAT` or `VECTOR`), the name of the attribute, the count and the particle index must be known. The function `data(Particle Attribute, Particle Index)` and `dataWrite(Particle Attribute, Particle Index)` returns a pointer to data of the specified attribute. If the attribute is a vector,x the data will be an C++ array with three components.

The only difference between these two functions is that `data` belongs to the class `ParticlesData` and `dataWrite` belongs to `ParticlesDataMutable`.

The pointer `data` returns is constant, which means that the data it points to cannot be changed only read. A brief tutorial can be found on Partio's homepage about this, see [29].

Pros:

- Easy to use.
- Can add attributes dynamically to the particles.
- Supports SideFX's Houdini particle format as well as Autodesk's Maya and Pixar's RenderMan particle format.
- Has a standalone particle viewer.

Cons:

- Unable to remove particles.
- Updating particle attributes is a little tricky.
- Slow when writing to disk.

3.1.3 Writing to disk

Since we use Partio for the I/O part of the particle field we first must convert our particle field to a Partio object. This is simply done by first creating a Partio object with the same particle count as the field. Add all the attributes of the particles to the Partio object, then iterating over all particles and coping the data to Partio object. When that is completed, the write to disk can be executed by the Partio funciton `write`. To avoid unnecessary memory usage, the Partio object is then released.

3.1.4 Particle layout

The particle interface is similar to the interface Johansson suggested, which is that the particle must be represented by a position and a radius. But we do not need to keep track if the particle has escaped or not. We do need to know the velocity of the particle. That leaves us with this simple interface:

- `setPosition( float, float, float)`
- `getPosition()`
- `setRadius( float )`
- `getRadius()`
- `setVelocity( float, float, float)`
- `getVelocity()`

Where the Position and Velocity are stored as three component float arrays, the radius is stored as a single float.

3.1.5 Dynamically add attributes to particles

The *Techniacl Directors* (TD) at Digital Domain also wanted to dynamically add/remove other attributes to the particles. Since Partio can add/remove attributes dynamically and we already use it for disk I/O, it was also natural to use it for storing and handling the attributes. Therefore another particle interface was implemented which is similar to the interface suggested in section 3.1.4, but with some extra functions to accommodate for adding and removing particles. We also need to have a particle index for each particle so that it can be linked to the Partio data object.

- `setID( long int )`
- `getID()`
- `setPosition( float, float, float )`
- `getPosition()`
- `setRadius( float )`
- `getRadius()`
- `setVelocity( float, float, float )`
- `getVelocity()`
- `addAttribute( Partio::ParticleAtribute, void* data )`
- `removeAttribute( Partio::ParticleAtribute )`

3.1.6 Partio Particle Field

We cannot use the Particle field as it is for the dynamic particles because the interface wont match. Also since particles can be dynamically added or removed from the grid and Partio do not support removing particles from the data, *ghost particles* can show up when writing to file. Therefore we created a Particle Field that can handle this. Internally it has a Partio data object P_{data}, a garbage list g_{list}: which store the particle IDs of the removed particles, and a class, `IDHandler` that handle the ID assignment. Each particle that is created asks for an ID from the `IDHandler`.

The `IDHandler` checks if g_{list} is empty: if true; returns the particle count, if not; pops the first element in g_{list} and returns that. If a particle is removed it returns its ID to the IDHandler which place the ID in g_{list}. This makes sure that each particle gets a unique ID and old IDs are reused thus reducing ghost particles and the Partio data object from growing out of hand. This can be seen as a version of the design pattern *Mediator* discussed by Gamma et. al. in [11].

Since the ID number is directly linked to the particle count, e.g the last ID number is equivalent to the number of particles in the field. The first element in the garbage list g_{list} must have the lowest ID number, or it might happen that $ID > particle\,count$. Which will cause a crash.

We use the C++ STL container [27] *priority_queue* for g_{list}, since it uses heap sort for sorting the elements and runs in $O(nlog\,n)$. For default the first element in the priority_queue is the greatest element but can be changed to the smallest by replacing the default comparing function.

3.2 Particle-In-Cell Method

The *Particle-In-Cell* (PIC) method was invented at Los Almos National Laboratory in the early 1950s but was first mention in a report from 1963 by Harlow [12] see Bridson in [5] for more information. The major difference between this method and the Eulerian fluid solver is that it uses particles to store the quantities. The quantities are then transfered to the grid before computing external forces, pressure etc (same as for the Eulerian fluid solver). One issue with the Eulerian fluid solver is that it suffers from numerical viscosity mostly caused when solving the advection step. This is mostly due to the semi-Lagrangian method, presented by Stam in [28], because the interpolation step in the method takes a weighted average of values of the previous time step. Hence each advection step performs an average operation. As Bridson mentions in [5] is that averaging tends to smooth out sharp areas, similar to what a low-pass filter does to an image. Therefore more and more sharp details will be lost as the time progress. This is called *numerical dissipation*. Which Bridson explains in full detail in his book [5].

Since the PIC method uses particles this problem solves itself when the particles are moved along with the velocity field. To clarify this, for the semi-Lagrangian method, to update the velocity at position $\vec{x}$ we go and look for an imaginary particle which is then used to update the velocity(note that no actual particles are created for the method, they are strictly imaginary). For PIC, to update the velocity at position $\vec{x}$ we simply wait until the particle comes to us and transfer its velocity to position $\vec{x}$. The algorithm is as follows:

1. Transfer the particle values q to the grid, replacing the values that is previously stored on the grid. See section 3.2.3. And if necessary extrapolate using the method described in 3.3.1
2. Compute all other terms on the grid such as pressure, external forces etc. Same as for the Eulerian solver, see section 2.5.
3. Transfer the newly computed values q_g on the grid to the particles by interpolating, see section 3.3, replacing q on the particles.
4. Advect the particles in the grid velocity.

3.2.1 Seeding particles

Inside each cell that represents the fluid, eight particles are randomly seeded. Zhu and Bridson [38] propose to use a stratified method to get a better distribution of the particles and avoid aliasing when "the flow is underresolved at the simulation resolution". The cell is divided into 2x2x2 sub-cells and in each a particle is randomly generated. To avoid gaps/holes in the fluid and noise when running the fluid simulation Bridson in [5] propose that a fluid cell should not have less than three particles and no more than twelve. Thus when a fluid cell fails to fulfill these requirements it is reseeded; below the criteria one particle at a time is added until the criteria is met. If there is more than twelve particles, the overflow is removed.

3.2.2 Smart Reseeding

Just reseeding a particle randomly inside the cell will cause horrible artifacts and could make the fluid unstable. If a particle is generated outside the fluid, see figure 6, the fluid will start to grow, which is an undesirable effect.

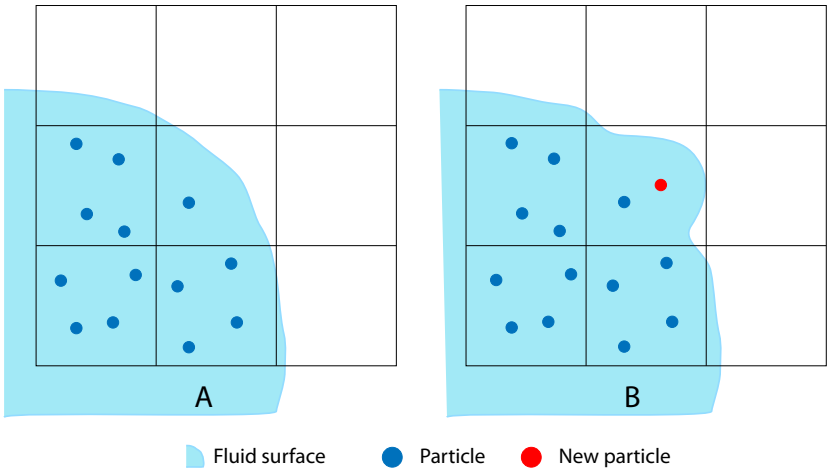


Figure 6: Shows the issue if a particle is reseeded outside the other particles.

Instead the reseeding must be contained inside the fluid to avoid this fungus-like effect. To accommodate for this effect, a bounding box is created that spans between the current particles in the cell and the walls adjacent to fluid cells.

The algorithm is as follows and is also visualized in figure 7:

1. Two bounding boxes B_c and B_p are created. One for the cell itself and one for the already existing particle(s) in the cell.

   - With $\vec{c}_{min} = (0,0,0)$ and $\vec{c}_{max} = (1,1,1)$, $B_c \in \{\vec{c}_{min}, \vec{c}_{max}\}$;
   - $\vec{p}_{min} = min_{comp}\{\vec{p_0}, ..., \vec{p_n}\}$, $\vec{p}_{max} = max_{comp}\{\vec{p_0}, ..., \vec{p_n}\}$ where $n < 3$, $\vec{p_n}$ is the coordinates for particle n.

   min_{comp} stands for taking the minimum of a set vectors component-wise, the same applies for max_{comp} but the maximum instead. $B_p \in \{\vec{p}_{min}, \vec{p}_{max}\}$

2. Each adjacent cell is checked to see if it is marked as a fluid cell (six cells in total). If a cell is a fluid cell the corresponding coordinate of B_p is assigned to B_c i.e if $c_{i+1,j,k} = fluid$ then $c_{max}[0] = p_{max}[0]$ in other words that side of B_c is scaled to the side of B_p, see figure 7.

3. A special case is needed to see if the cell only contains one particle and the adjacent cells are empty. Because that will result in $\vec{c}_{min} = \vec{c}_{max} = \vec{p}_0$ and if not taken care of the new particle will be created on top of the already existing particle, which we do not want. Therefore if this case is true B_c is expanded and the new particle will be seeded around the already existing particle instead of on top of it.

3.2.3 Particle-To-Grid Transfer

When transfer the velocity from the particle to the grid we want a robust and fast way of computing it. Due to the fact that this will be calculated for each particle, and a scene can easily have over 500 000 particles. Thus we use a simple trilinear hat function shown in equation 33 for computing

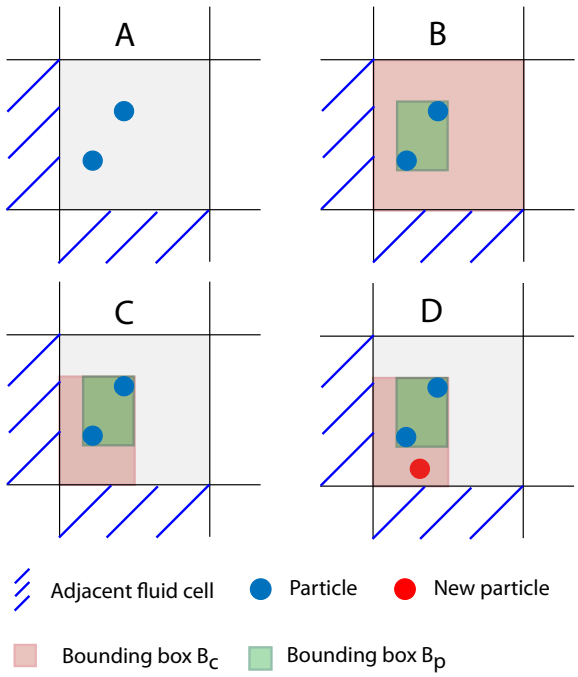


Figure 7: The smart reseeding alorithm. (A) a fluid cell with only two particles. (B) Bounding boxes B_c and B_p are created. (C) B_c is scaled to accommodate for empty cells. (D) a new particle is generated inside B_c

the total energy of the velocity reaching one grid point.

$$k(x,y,z) = h(\frac{x}{\Delta x})h(\frac{y}{\Delta y})h(\frac{z}{\Delta z}) \tag{33a}$$

$$h(r) = \begin{cases} 1-r & : \quad 0 \leq r \leq 1, \\ 1+r & : \quad -1 \leq r < 0, \\ 0 & : \quad \text{otherwise.} \end{cases} \tag{33b}$$

Bridson in [5] page 142, mention that the kernel function should be adapted to the grid spacing, if Δx is less: particles can momentarily vanish from the grid i.e. particles will not contribute to any cell. If Δx is larger much of the sharpness of the particle method will be smoothed i.e. particles will affect a larger area and lot of the high frequencies will be lost.

Calculating the velocity is done for each component of the velocity, the u component is computed as follows:

$$u_{i+1/2,j,k} = \frac{\sum_p u_p k\left(\vec{x}_p - \vec{x}_{i+1/2,j,k}\right)}{\sum_p k\left(\vec{x}_p - \vec{x}_{i+1/2,j,k}\right)} \tag{34}$$

the other two components are calculated in similar fashion, note that this equation is for a staggered grid (MAC grid) hence the half offsets. Bridson also recommends, at least for velocity, that the weight should be calculated separately for each grid point due to the non-uniform particle distribution.

We implemented two different methods for computing the velocity, a *shooting* and a *gathering* method. The shooting method is the method suggested by Bridson, in [5] page 148. which is to loop over all particles and update each grid point that is affected by the current particle. The

gathering method is to loop over the grid points instead and update it using the adjacent particles, which affects the grid point.

3.2.4 Shooting method

The transfer is calculated component-wise since we use a staggered grid (see section 2.2) hence the offsets for the different components will be different. For u component, and similar for the other components, the algorithm is as follows:

1. Reset all grid values to zero.
2. Loop over all particles.
   (a) Loop over all grid points affected by the particle.
   (b) For each grid point: compute $u_p k\left(\vec{x}_p - \vec{x}_{i+1/2,j,k}\right)$ and $k\left(\vec{x}_p - \vec{x}_{i+1/2,j,k}\right)$ and add them to the grid point (store the weight in a second buffer) .
3. Loop over all grid points and calculate equation 34.

3.2.5 Gathering method

The gathering method is similar to the shooting, since it use the same kernel and equation as the shooting method. The reason for using the gathering method is that it can run in parallel without racing problems and memory corruption. More on this in section 3.7 and the discussion.

1. Reset all grid values to zero.
2. Loop over all grid points
   (a) Calculate the velocity using equation 34 by looping over all particles that affect the grid point

3.3 Grid-To-Particle Transfer

The transfer from the grid to the particles is similar to the gathering method described above. For the u_p component, the other components are computed in a similar way. We loop over all particles and for each particle we get the eight closest grid points, since they are the only one that affects the particle. Then interpolate u_p from those points using:

$$u_p = \frac{\sum_{i,j,k} u_{i,j,k} k(\vec{x}_p - \vec{x}_{i,j,k})}{\sum_{i,j,k} (\vec{x}_p - \vec{x}_{i,j,k})} \tag{35}$$

In pseudocode the method can be described as:

1. Reset all velocity components to zero for the particles.
2. Loop over all particles.
   (a) For each particle p calculate the eight closest grid points by transform the particles position $\vec{x}_{local} \rightarrow \vec{x}_{world} \rightarrow \vec{x}_{grid}$.
   (b) Floor $\vec{x}_{grid}$ to get the closest grid point $\vec{c}_{i,j,k}$ to get the other seven is just stepping $\frac{1}{2}$ in positive direction.
   (c) Compute u_p for p using equation 35.
   (d) Repeat step 2b and 2c for the other two velocity components.

3.3.1 Extrapolation using Fast Marching Method

Extrapolation might be necessary since the velocity is only defined near the particles and not defined over the entire narrow band of the level set, which describes the surface. A brute force method is to simply to go through all the undefined points and copy the closest defined velocity. The downside to this is that it is extremely slow and the points on the surface must be defined. But this might not be the case if the radius differ from the velocity transfer and the surface reconstruction.

A better and more efficient way is to use a method called *Fast marching* proposed by Adalsteinsson et. al. in [1] which runs in $O(n \log n)$. And is used for example by Chentanez et. al. in [24] to extrapolate the velocity on the CPU. Zhu et. al. in [38] used *Fast Sweeping* [37] since it was easier to implement and marginally faster than fast marching. The reason for choosing Fast Marching instead of Fast Sweeping is that is was easier to implement interchangeable *functors* (a class in C++ that behaves like a function) when extrapolating the velocity. Thus allowing the functor to be replaced later if desired, right now it only copies the values.

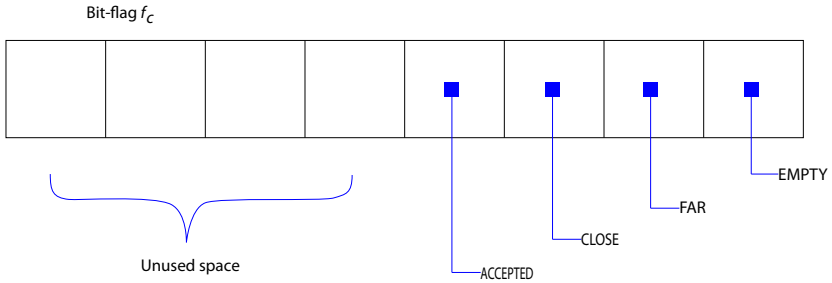


Figure 8: This shows how the different states are placed in the bit-flag f_c. The cells state can only go from right to left and a bit shift to the left is used to quickly change state.

The input to this function is a velocity field $\mathbf{F_v}$ and a level set $\mathbf{F}_{ls}$ that describe the surface of the fluid. The Extrapolation method is based on the fast marching method presented by Adalsteinsson et. al. in [1]. But instead of having three lists; Far, Close and Accepted. We only use two lists; a bit-list $\mathbf{B}$ and a Close-list $\mathbf{C}$. $\mathbf{B}$ contains a bit-flag f_c for each cell and also has the spatial information of the surrounding cells. Whereas $\mathbf{C}$ is the same as in [1]. Each bit-flag stores the state of its cell i.e. Empty, Far, Close, or Accepted, see figure 8. An Accepted list is redundant to store since all of the classifications are stored in the bit-list. The close-list takes data objects

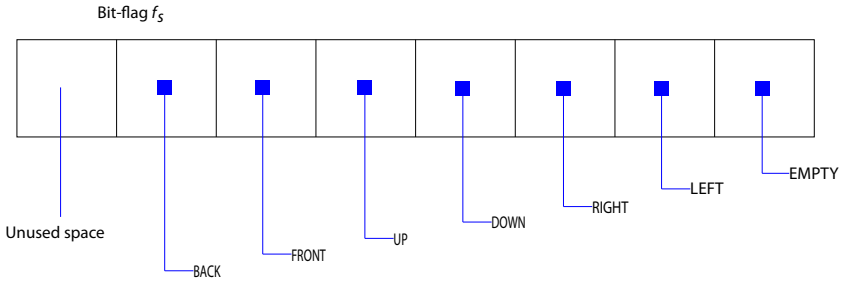


Figure 9: Layout of the bit-flag f_s

containing the two linear offsets (Block and Element) to the bit-flag position in the bit-list and the distance to the surface ϕ. To efficiently get the data object which has the smallest ϕ we use the same heap-sort technique which is suggested by Adalsteinsson et. al. in[1].

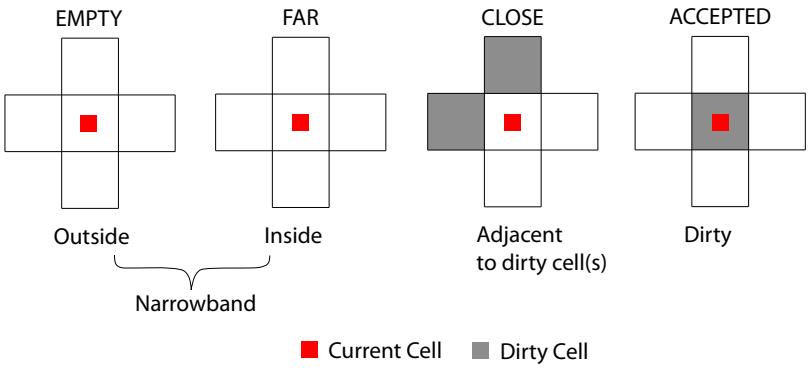


Figure 10: A 2D example of the four different states a cell can have.

1. Compute a bounding box B_{ls} around $\mathbf{F}_{ls}$, which will be the search field.
2. Loop over all cells inside B_{ls} and for each cell:
   (a) Create a bit-flag, f_s which contains the adjacent information and set it to zero, e.g turn off all flags. See figure 9 for layout of the bit-flag.
   (b) Check if $|\phi| > \gamma_{ls}$ (the width of the narrow band) if true turned on the empty flag in f_s and jump to 2d.
   (c) Check its neighbors and if any of them are dirty, turn on that position in f_s.
   (d) Classify the cell to be either Empty, Far, Close or Accepted, see figure 10, using f_s. And turn on the corresponding flag in f_c in the bit-list.
   (e) If the cell has the label "Close" also add it to C.
3. Pop the first entry of the close-list and get ϕ for each dirty neighbor for that cell.
4. The cell who has the smallest ϕ copy it is velocity, in our case the corresponding velocity component, to the corresponding position in $\mathbf{F}_v$.
5. Put the adjacent cells that is marked as "Far" in $\mathbf{C}$ and update the current cell to be accepted and the "Far" cells to "Close" by simply bit shift each f_c one step to the left.
6. If $\mathbf{C} = \emptyset$ stop else go back to 3 and repeat the steps.

3.4 Fluid-Implicit-Particle Method

Although PIC solves a lot of issues for the fluid solver, the numerical viscosity for one, it also has some disadvantages. It will lose energy over time

due to the smoothing that happens when averaging fluid quantities from particles to the grid. Then interpolate the smoothed quantities on the grid back to the particle, smoothing them even further. Thus loosing a lot of fine details for the fluid, which we desire to have.

One rather simple and clever solution to counteract this was introduced by Brackbill et. al. [4] and is called *Fluid Implicit Particle* (FLIP) method. Instead of replacing the quantities each time step by interpolating back and forth, the changes in the quantities, which are computed on the grid, are interpolated and are used to increment the particles. This method found its way to the fluid simulation by Zhu and Bridson in [38] and it produces fine and detailed fluids. And as Bridson mention in [5], page 149, that it "essentially eliminates all numerical dissipation from advection". But it still has loss of vorticity due to the first-order time-splitting which is used to solve the Navier-Stokes equations. On the contrary its easy to implement especially when we had already implemented PIC. Bellow follows the basic structure of the method described in [5] by Bridson:

1. Transfer particle values q_p (our case only velocity) to the grid $q_{i,j,k}$ as explained in section 3.2.3 using equation 34 and extrapolate on the grid if necessary as described in section 3.3.1.
2. Save the grid values $q_{i,j,k}$.
3. Compute all other terms on the grid, i.e Pressure, external forces etc to form an updated grid $q^{new}_{i,j,k}$.
4. For each particle compute the change $\Delta q_{i,j,k} = q^{new}_{i,j,k} - q_{i,j,k}$ and add it to the particle.
5. Advect the particles in the grid velocity.

3.5 Surface Reconstruction

One problem with the PIC/FLIP method, and the use of particles, is that it lacks a surface which comes for free with the Eulerian solver. Blinn in [3] introduced a method that wraps a smooth implicit surface around the particles which he calls *Blobbies*.

3.5.1 Blobbies

Given the particles position x_i *Blobbies* will be defined as following function:

$$F(\vec{x}) = \sum_i k\left(\frac{||\vec{x} - \vec{x}_i||}{h}\right) \tag{36}$$

where h is a user defined parameter which Bridson, in [5] page 85, explains as the extent of each particle. k is a smooth kernel and suggest a spline kernel:

$$k(s) = \begin{cases} (1-s^2)^3 & : \quad s < 1, \\ 0 & : \quad s \geq 1. \end{cases} \tag{37}$$

since it is cheap and simple compare to a Gaussian kernel.

The user defined parameter h should be three times larger than half of the grid spacing Δx, which we calls inter-particle spacing r. This gives that $h = 3r$ but might be tweaked to get a better surface. Blobbies creates a implicit surface that is defined as

$$F(\vec{x}) = \tau \tag{38}$$

where τ is a threshold and Bridson suggest as default values to be $\tau = k(r/h)$ which creates a sphere of radius r for a isolated particle. To sample this to a grid and define a Level Set equation 38 can be rewritten as

$$F(\vec{x}) - \tau = \phi(\vec{x}) = 0 \tag{39}$$

Implementing Blobbies is straight forward; first an empty Level Set is created, the grid spacing Δx does not need to match the grid spacing of the particle field. A bounding box is calculated around the particles and is expanded in every direction for accommodate both the narrow band width γ and h else the surface can simply be missed.

The reason for calculating a bounding box is that the level set is only defined inside the narrow band, thus is unnecessary to go through the whole grid and sample. This speeds up the sampling step greatly since the sampling will only occur close to the particles. In each sample step (grid point) equation 39 is evaluated and stored on grid if $\phi(\vec{x}) < \gamma$. A renormalization step and rebuild of the narrow band is necessary in order to fulfill equation 10. But Blobbies suffers from that smooth surfaces, which should be smooth, can be a bit uneven and clearly shows the underlying particles.

3.5.2 Improved Blobbies

Therefore an improved version of Blobbies was suggested by Zhu and Bridson in [38] that reduce this artifact and the implicit function is instead defined as:

$$\phi(\vec{x}) = ||\vec{x} - \bar{X}|| - \bar{r} \tag{40}$$

where $\bar{X}$ is a weighted average of nearby particles defined as:

$$\bar{X} = \frac{\sum_i k\left(\frac{||\vec{x}-\vec{x}_i||}{h}\right)\vec{x}_i}{\sum_i k\left(\frac{||\vec{x}-\vec{x}_i||}{h}\right)} \tag{41}$$

$\bar{r}$ is a weigthed average of nearby particle radii and is computed as:

$$\bar{r} = \frac{\sum_i k\left(\frac{||\vec{x}-\vec{x}_i||}{h}\right) r_i}{\sum_i k\left(\frac{||\vec{x}-\vec{x}_i||}{h}\right)} \tag{42}$$

Implementing *Improved Blobbies* is similar to the older method except evaluating equation 40 instead of 36. Further improvements has been made for surface reconstruction but has not been implemented see [2] and [34].

3.6 Wrap everything into Python

One of the benefits of having a script language is that the edit-test-debug cycles are really quick compare to languages that needs compiling. This enables us to quickly test different setups of the fluid or tweak the values for example improved blobbies to find the what we are after. There are downsides for script languages as well, more on this in the discussion (section 5). And the reason for choosing Python is that it is heavily used in production and it supports wrapping C/C++ code into modules relatively easy with the API *Boost.Python*, see [18]. The reason we choose Boost.Python instead of another wrapping API is that there were a few people on Digital Domain that had done some wrapping with it, thus it was already in the pipeline. Before going into how to wrap C++ code into modules we first need to go through what a *member function pointer* is in C++ since it is used in Boost.Python. And its syntax can be rather intimidating and if not *typedef* it can be confusing as well.

3.6.1 Function Pointer

There are not many C++ programmers that know that *member function pointers* (MFPs) exist and even less that actually use them. Since in most cases where a MFP can be use there is usually another way of solving it which is faster and less painful. To start of easy we will first show how a *function pointer* (FP) is declared in C++. Let us say we want to declare a FP to the function `foobar` we declared earlier in section 2.6:

```
int foobar(int x)
{
   // multiply the argument by 2
   return x*2;
}

// Declaration of function pointer
int (*my_func_ptr)(int);
// To make it more clear we can also use typedef
typedef int (*MyFuncPtrType)(int);

MyFuncPtrType my_func_ptr;
```

To assign the FP to the function `foobar` we do like this:

```
// assign the function to the pointer
my_func_ptr = foobar;
// call the function
(*my_func_ptr)(21);
```

3.6.2 Member function pointer

Since most of the function in C++ belongs to a class. There also exist MFPs to these member functions and are similar to the function pointer, to the surface anyway. Underneath they become really complex since they deal with issues like multiple inheritance and virtual classes. This results in that the MFP can vary in size depending on what compiler is used, since different compilers tackle the issues with MFPs differently. To get a full explanation on these pointers please see Clugston's article at [7] which explains this in detail, while we will just present the syntax of these pointers. Therefore let us show how to declare a MFP to the class we presented earlier, `Foo`:

```
// C++:
class Foo
{
  private:
     int my_spam,
          my_egg;
  public:
    Foo(const int &spam, const int &egg)
  {
     my_spam = spam;
     my_egg = egg;
  }
   int getSpam() { return my_spam; }
   int getEgg() { return my_egg; } const
```

```
};

// member function pointer
int (Foo::*my_memfunc_ptr)();
// with typedef
typedef int(Foo::*FooMFPtrType)();
FooMFPtrType my_memfunc_ptr;
```

Notice a special operator `::*` and the class name are used in the declaration. This MFP will only work for `Foo::getSpam` and not `Foo::getEgg` since the last is constant. For the const function `Foo::getEgg` the declaration will be:

```
// const member function pointer
int (Foo::*my_constmemfunc_ptr)() const;
// with typedef
typedef int(Foo::*FooConstMFPtrType)() const;
FooConstMFPtrType my_memfunc_ptr;
```

To make the MFP point to a function we assign it like this:

```
 my_memfunc_ptr = &Foo::getSpam;
```

Then to call the function two special operators are used:

```
Foo* x(1, 2);

(x->*my_memfunc_ptr)();

// if the class is on the stack
// the ".*" operator is used:

Foo y(1, 2);

(x.*my_memfunc_ptr)();
```

3.6.3 Boost.Python

The Boost.Python API was developed to allow seamless wrapping of C++ classes and function into Python modules without making any change to the actual C++ code. And since we can make the interface similar in Python as it is in C++, we can easily setup an script in Python to run a fluid simulation and tweak our parameters. Then if we want we can easily convert the Python script to a C++ program. It also makes it easier for people to switch between the C++ implementation and the Python implementation of the fluid framework.

For wrapping a function to Python using Boost.Python is straight forward and has some similarities with how you define a function in Python. For exposing the function `foobar` we used in previous example:

```
// Pyclone.cpp
#include <boost/python.hpp>

BOOST_PYTHON_MODULE(Pyclone)
{
    using namespace boost::python;
```

```
    def("foobar", foobar, (arg_("x")) );
}
```

We can then import the module we call `Pyclone` into python after loading the compiled *dynamic linked library* (DLL) into Python.

```
>>> import Pyclone
>>> Pyclone.foobar(21)
42
```

What we pass to the function *def* is first the name of the function (it does not need to be the same name as the C++ function) the second argument is a function pointer to the function we want to expose. The last argument, which is not necessary, is the name of the argument for `foobar`, this is just to name the argument in Python so it match the name of the argument in C++. So when the help command is called in Python for the function, the argument name will be `x` instead of `object1` (the default name if the last argument is left out). Before we start to explain how to wrap classes into Python it is worth mentioning that to put each class/function wrapper into separate files. Instead of writing all in the module file (Pyclone.cpp), especially when you wrap more than 30 different classes like we did. Not only thus this reduce compile time [8] it can also be good if the function/class is templated e.g. a templated vector class would look something like this:

```
// Vector3t.h
template<typename Type>
class Vector3t{ ...};

// PyVector.h
#include <Vector3t.h>
#include <boost/python.hpp>

using namespace boost::python;
template<typename Type>
void vector_class_wrap(const char* ClassName)
{
   typedef VectorType Vector3t<Type>;
   class_<VectorType> ...;
}

// Pyclone.cpp
#include <PyVector.h>

BOOST_PYTHON_MODULE(Pyclone)
{
   // define a vector class that take integers
   vector_class_wrap<int>("Vector3i");
   // define a vector class that take doubles
   vector_class_wrap<double>("Vector3d");
}
```

since Python does not support template classes each variation must be explicitly declared in the module. The example above also shows how to wrap template classes. Which is not explained in the tutorial [8]. Let us explain this in more detail. First of all we need a class to expose, for this example we use the previous class `Vector3t`

```
// Vector3t.h
template<typename Type>
class Vector3t
{
  private:
  // private variable not important for this example
  ...

  public:
     // constructor
     Vector3t(Type x, Type y, Type z,
    bool transpose = false) { ... }
     // member functions
     Vector3t operator+(const Vector3t& other) {...}
     Vector3t dot(const Vector3t& other){ ... }
     Vector3t dot(const Type &x,
    const Type &y,
    const Type &z ){ ... }
    ...
};
```

we now need to wrap this class so that we can use it in Python. For this we use the Boost.Python function `class_` which is a template class that can take four template arguments, three of them which has a default value.

`class_<T, bases, Heldtype, NonCopyable>`:

- `T`: the class we want to expose.
- `bases`: as the name applies if the class we want to wrap is derived from another class (classes).
- `Heldtype`: Specify the type which is embedded in the Python object, must be T, a class derived from `T` or a pointer to `T`.
- `NonCopyable`: Specify if Python is allowed to make an instance of `T` (copies). This can be good to suppress if the class contains huge amount data e.g. grid of some sort.

the class has three different constructors but we will only show one, to see the other two please see [8] or [19]. To wrap `Vector3t` with the use of Boost.Python's `class_` we need to know the different arguments it takes. We know `T` (`T` = `Vector3t`), the class is not derived hence we can use the default value for `bases`, namely `boost::python::bases<>`. We will use `boost::shared_ptr` [17] for storing the object and the parameter is after `bases`. The `Heldtype` will be `boost::shared_ptr<Vector3t<Type> >`. We want Python to be able to copy the object therefore we can leave the last argument to it is default value.

```
// PyVector.h
#include <Vector3t.h>
#include <boost/python.hpp>

using namespace boost::python;
template<typename Type>
void vector_class_wrap(const char* ClassName)
```

```
{
   typedef VectorType Vector3t<Type>;
   class_<VectorType,
  bases<>,
  boost::shared_ptr<VectorType >
  > (ClassName);
}
```

3.6.4 Expose classes

If we compile the module we will be able to see and create the Vector class with its default constructor. Since this is automatically exposed to Python via Boost.Python, this can be disabled if we want to. But we cannot use any functions since we have not wrapped any. The next step is simply to wrapped its function and constructor. Since we want to use a `shared_ptr` to make sure that we do not get any memory leaks, we will have to wrap the construct manually. See [8] for the normal way of wrapping a constructor. We also use template classes which can be an issue with the way suggested in the tutorial.

```
// PyVector.h
#include <Vector3t.h>
#include <boost/python.hpp>
#include <boost/shared_ptr.hpp>

using namespace boost::python;

// creating a wrapper for the constructor
// since it has a parameter with a default value
// we need to create two wrapper functions

template<typename Type,
        class VectprType>
boost::shared_ptr<VectorType> Vector3t1a(Type x,
           Type y,
           Type z)
{
   return boost::shared_ptr<VectorType>(
         new VectorType(x,
            y,
            z) );
}

template<typename Type,
        class VectprType>
boost::shared_ptr<VectorType>
     Vector3t1b(Type x,
      Type y,
      Type z,
      bool transpose)
{
   return boost::shared_ptr<VectorType>(
       new VectorType(x,
          y,
```

```
          z,
          transpose) );
}

template<typename Type>
void vector_class_wrap(const char* ClassName)
{
   typedef VectorType Vector3t<Type>;
   class_<VectorType,
  bases<>,
  boost::shared_ptr<VectorType >
  > (ClassName)
  .def("__init__",
    make_constructor(&Vector3t1a<Type,
      VectorType>,
      default_call_policies(),
      (arg_("x"),
      arg_("y"),
      arg_("z"))) )
  .def("__init__",
    make_constructor(&Vector3t1b<Type,
      VectorType>,
      default_call_policies(),
      (arg_("x"),
      arg_("y"),
      arg_("z"),
      arg_("transpose") ) ) )

  ;
}
```

A little explanation might be needed for this code example, since `Vector3t`'s constructor has an argument with a default value we need to explicitly declare the both versions, one without the default value and one with it. The `.def` function is similar to `def` but instead used for member functions, it can take both function pointer and member function pointer. The `make_constructor` speaks for itself, it simply creates a constructor from the proxy function `Vector3t1a` and assign it to the Python constructor `__init__`. For wrapping member function works in in similar fashion:

```
// PyVector.h
#include <Vector3t.h>
#include <boost/python.hpp>
#include <boost/shared_ptr.hpp>

using namespace boost::python;
...

template<typename Type>
void vector_class_wrap(const char* ClassName)
{
   typedef VectorType Vector3t<Type>;
   class_<VectorType,
  bases<>,
  boost::shared_ptr<VectorType >
```

```
  > (ClassName)
  // constructor declarations
  ...
  // expose a function
  .def("add",
       &VectorType::operator+,
       (arg_("other") ) )
  ;
}
```

and if we want to use Python's "+" operator instead of calling the function `add` we can simply replace it with `__add__` which is Python's `operator+`. Similar exist for "−", "/", "×". If we want the operator "+ =" we can add an "i" i.e. `__iadd__`.

```
.def("__add__",
     &VectorType::operator+,
     (arg_("other") ) )
```

Next function we want to expose is `dot` but it exist two functions called `dot` i.e overloaded functions. We cannot pass the member function as for the function `operator+`. We first need to specify which of the two functions we mean, instead we will have to declare a member function pointer and pass that as the argument .

```
// PyVector.h
#include <Vector3t.h>
#include <boost/python.hpp>
#include <boost/shared_ptr.hpp>

using namespace boost::python;
...

template<typename Type>
void vector_class_wrap(const char* ClassName)
{
   typedef VectorType Vector3t<Type>;

   VectorType (VectorType::*dot1)(
      const VectorType& other) =
        &VectorType::dot;

   VectorType (VectorType::*dot2)(
        const Type& x,
        const Type& y,
        const Type& z) =
  &VectorType::dot;

   class_<VectorType,
  bases<>,
  boost::shared_ptr<VectorType >
  > (ClassName)
  // constructor declarations
  ...
  // expose a function
```

```
  ...
  .def("dot", dot1,(arg_("other")))
  .def("dot", dot2,(arg_("x"),
      arg_("y"),
      arg_("z")))

  ;
}
```

The overload mechanism for member functions works as well in Python. And in this example we have declared two MFPs `dot1` and `dot2`. If any function has default values we can either create a function for each version of it, similar what we did for the constructor, or use Boost.Python's macro that does this for us called `BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS`. Let us say that `Vector3t` has a function, `my_func` that has four argument, three of them has default values. Instead of creating four proxy functions that handles each case, we can use the Boost.Python macro:

```
// PyVector.h
...
BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS(my_funcs,
          my_func,
          1,
          4)

template<typename Type>
void vector_class_wrap(const char* ClassName)
{
   typedef VectorType Vector3t<Type>;

   ...

   class_<VectorType,
  bases<>,
  boost::shared_ptr<VectorType >
  > (ClassName)
  // constructor declarations
  ...
  // expose a function
  ...
  .def("my_func", my_funcs( (arg_("a1"),
          arg_("a2"),
          arg_("a3"),
          arg_("a4"))) )
  ;
}
```

where the last two arguments, of the macro, specify minimum and maximum of arguments of the function `my_func`. Similar macro also exists for functions.

3.6.5 Call Policies

There are also call policies for functions that returns a pointer, or a passing a reference to a function. Python garbage collection can accidentally delete

an object that another object depends on, thus resulting in that the program crashes when trying to access that specific object. The reason for this is that Python does not know that these two objects are connected and this must explicitly been set in the wrapper module. A good example of this is described by Guzman and Abrahams on the tutorial page [8]. A list of these predefined call policies written by Guzman and Abrahams from the same tutorial page is shown below:

- **with_custodian_and_ward:** Ties lifetimes of the arguments
- **with_custodian_and_ward_postcall:** Ties lifetimes of the arguments and results
- **return_internal_reference:** Ties lifetime of one argument to that of result
- **return_value_policy<T>** with T one of:
  - **reference_existing_object:** naive (dangerous) approach
  - **copy_const_reference:** Boost.Python v1 approach
  - **copy_non_const_reference:**
  - **manage_new_object:** Adopt a pointer and hold the instance

These calls can be combined with each other creating rather complex call policies to prevent accidentally deletion. These calls are placed as the last argument of the function `.def`.

3.7 Parallelism

The multi-core processors have put new demands on the software developers. To fully utilize these new multi-core monsters, programs must be able to run in parallel and with it a new set of issues has arisen. For example a design pattern that is trivial to implement in serial can become a nightmare when trying to implement in parallel as Lee describes in [16]. But when done right can speed up a implementation significantly.

3.7.1 Threads

The most popular approach for parallel programing, or concurrent programming as Lee calls it, is *threads*. A thread is a sequential process that shares memory with other threads and is highly adopted in modern operating systems [16]. A process launched by the OS can contain multiple threads that can either be run on a single core in a sequential order or be divided between multiple cores to be executed in parallel. The most common problem with this is what is called a *race problem*, [6], when two threads access the same memory location. Depending on which thread writes to the memory first, will affect how the outcome will be.

For example a program "a" does a simple incrementation and is threaded with n number of threads; $a = \{t_0, t_1, ..., t_n\}$. One thread t_0 reads the current sum s from the memory and increment s by one, $s^{t_0} = s + 1$. But before it writes s^{t_0} back to memory t_1 intervenes and reads s and also increment it by one; $s^{t_1} = s + 1$ and writes it back to memory before t_0. When t_0 now writes s^{t_0} to the memory location it will replace s^{t_1}. If the sum s was equal to 1 from the start after this execution s would be equal to 2 and not 3 as expected. These race problems can be extremely hard to find when debugging. There are tricks to deal with these race problems,

one is what is called a *lock*. Which prevents other threads to write to the same memory location as the thread that holds the lock. And these threads have to wait until the thread who holds the lock release it. One issue with using locks is that a *deadlock* [16] can occur causing the program to crash.

3.7.2 Parallel

All methods presented here in section 3 except the shooting method in section 3.2.4 are implemented to also run in parallel and follows a similar design pattern. Which is to divide the workload to work on each block independently and distribute them to each thread. Thus guaranties that no more than one thread writes to each block at a time, and avoids racing problems and other issues when multiple threads tries to write to the same memory location. Therefore all of the methods mentioning above is designed with this rule.

The reason for the shooting method is not run in parallel is that it breaks this rule. When transfer from particle to grid a particle affects multiple grid points (cells) and it is no guaranty that all of them lies in the same block. Two threads can try to update the same grid point at the same time which can cause memory corruption. Therefore the gathering method is used when run i parallel. Since it has no problem when two threads reads from the same memory location (only causes a small bottleneck) which can occur on the edges of the blocks.

One other operation that must be atomic is in the Partio Particle Field (section 3.1.6) and that is the call to the `IDHandler`. For example two threads, t_0 and t_1, asks for an ID at the same time when reseeding, and there is only one ID in the garbage list g_{list}. Both threads runs a separate instance of the `IDHandler` and checks if g_{list} is empty and since no one has taken the last ID it is not empty. The thread t_0 pops the last ID thus clearing g_{list}. When t_1 tries the same thing the program will crash since t_1 tries to retrieve and clear an object that do not exist in memory anymore. That is why this operation must be atomic, and can be done with locks, i.e. one thread holds the lock until it has a ID from the `IDHandler` then release the lock, giving it to another thread which is waiting for an ID.

The API we used for threading the program is Boost.Thread [20], and is described on the web page as:

> **Boost.Thread** enables the use of multiple threads of execution with shared data in portable C++ code. It provides classes and functions for managing the threads themselves, along with others for synchronizing data between the threads or providing separate copies of data specific to individual threads. - Boost [20]

And as the quote describes it is a good API for threading and has a lot of good functions, e.g. locks for preventing race problems, easy to get data back from the threads and so on. There is other API:s that can be used for threading programs, one popular is intel's *Threading Building Blocks* (TBB) [14].

4 Results

This section will present the results from tests performed with our implementation of PIC/FLIP and surface reconstruction. We will also show results using our Python module. The hardware used is 8 Intel Xeon 2.40 GHz CPUs, with 24 GB RAM, each CPU has 4 cores and 8192 kB cache. To visualize our fields we use SideFX's Houdini but all data is generated with our implementation, i.e. the surfaces shown here are generated using Improved Blobbies and not with Houdini's own surface reconstruction.

4.1 Fluid implementation

Here follows the different tests we did with our implementation of the PIC/FLIP method.

4.1.1 Benchmark

Our fluid tests are performed on a scene containing particles that forms a sphere centered at the middle of the grid and the radius of the sphere is $r = \frac{w}{2}$. The test consist of running the fluid simulation one frame (i.e. one iteration), and measure the time. Each grid size are run with 0, 2, 4, 6 and 8 threads, where 0 threads stands for running it in serial. We tested both the PIC method and the FLIP method, the results can be seen in the tables below. The level set resolution, which we use for the surface reconstruction, is twice the size as the particle field i.e. for particle field with size 64^3 a level set with size 128^3 is used.

Particles	$w \times h \times d$	0 threads	2 threads	4 threads	6 threads	8 threads
46758	32^3	2.791 s	1.563 s	0.947 s	0.772 s	0.668 s
375794	64^3	21.283 s	11.868 s	6.751 s	5.118 s	4.320 s
3009586	128^3	179.017 s	94.846 s	52.597 s	40.320 s	32.677 s
24091194	256^3	1533.330 s	862.151 s	517.800 s	409.979 s	353.789 s

Table 1: One frame of PIC

Particles	$w \times h \times d$	0 threads	2 threads	4 threads	6 threads	8 threads
46758	32^3	2.722 s	1.582 s	0.923 s	0.749 s	0.648 s
375794	64^3	21.328 s	12.275 s	7.219 s	5.510 s	4.670 s
3009532	128^3	194.329 s	104.868 s	61.681 s	49.150 s	40.726 s
24091423	256^3	1542.160 s	866.895 s	519.885 s	408.224 s	353.722 s

Table 2: One frame of FLIP

4.1.2 FLIP breakdown

To get a better understanding of what is happening and where most of the time is spent on the fluid simulation, we present a breakdown for the FLIP method in table 3. Where the time for each step in the method is presented and also the total time. The scene is the same as for the previous tests with the grid size of 128^3 and total number of particles used is 3009611.

4.2 Partio conversion

We also measured the time for converting particles stored on the grid to a Partio object, which is done before writing to disk.

Step	8 threads	0 threads
Particle to grid	1.403 s	9.734 s
External forces	0.426 s	0.425 s
Extrapolation	0.055 s	0.057 s
Incompressible	1.039 s	1.727 s
Grid to Particle	0.441 s	1.325 s
Advection	1.214 s	4.022 s
Surfacing	21.057 s	156.458 s
Save to disk	15.804 s	16.276 s
Total	41.499 s	190.071 s

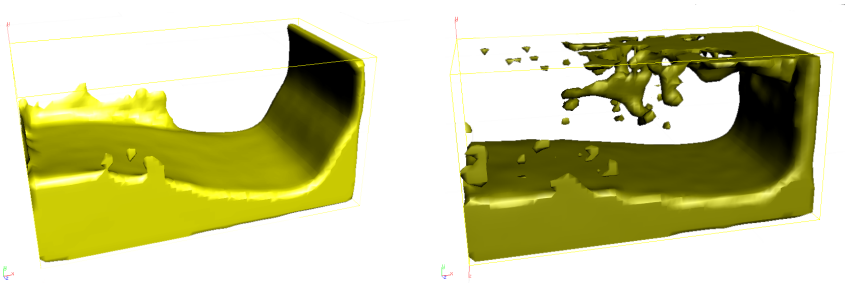
Table 3: FLIP breakdown, one frame, Grid size: 128^3, Particles: 3009611

Particles	0 threads	1 thread	2 threads	4 threads	6 threads	8 threads
2527880	0.326 s	0.482 s	0.340 s	0.256 s	0.222 s	0.206 s
20199656	3.038 s	3.810 s	2.554 s	1.856 s	1.712 s	1.604 s

Table 4: Partio conversion

4.2.1 Different fluid test

Figure 11 shows the difference between PIC and FLIP when testing the dam breaking scenario. The grid size for both are $128 \times 64 \times 64$. The other test, figure 12 shows how the collision between a PIC fluid and solid sphere. There is only one way coupling between these two, hence the fluid does not affect the solid sphere. It has the same resolution as the dam breaking test.



(a) Dam breaking test using PIC (b) Dam breaking test using FLIP

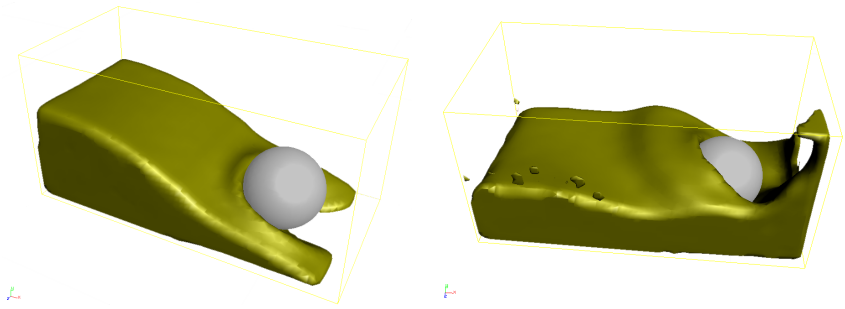
Figure 11: Dam Breaking test, with the PIC and FLIP method. A clear difference can be seen between these two methods.

4.2.2 Surface reconstruction

The following two figures 14 and 13 shows the result from our implementation of Improved Blobbies, the particle field and the level set has the same resolution. Where h is set to $3\Delta x$, the particle radius is fixed to Δx and the search radius for each particle is $3 \times 3 \times 3$ grid cells.

4.3 Python benchmark

We also tested running the same test as above with the PIC method, but execute it from a Python script. The result can be seen in table 5. But due to the long execution time we only tested for two different resolutions.



(a) (b)

Figure 12: Boundary condition test with solid object, one way coupling i.e. the solid object is not affected by the fluid

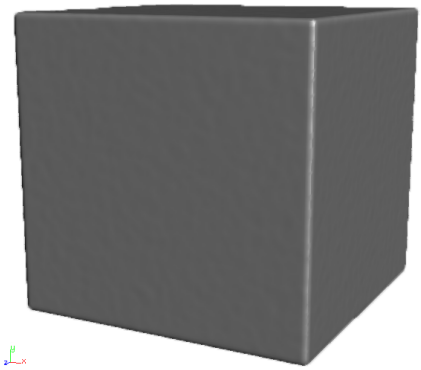
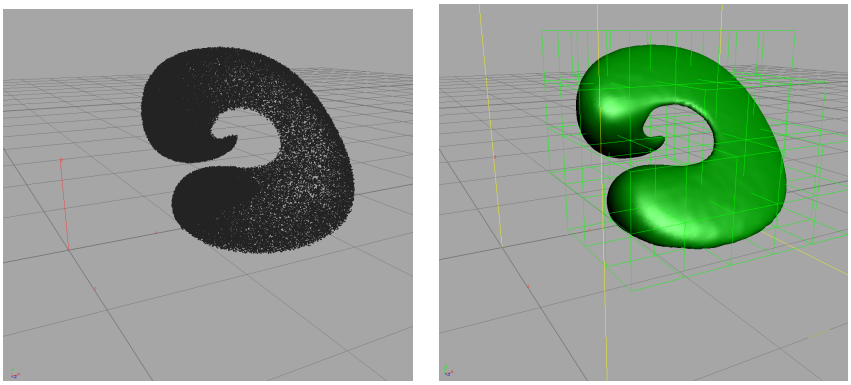


Figure 13: Surface reconstruction using Improved Blobbies. Small bumps can be seen on the "flat" surfaces on the box



(a) The underlying particles (b) A surface wrapped around the particles

Figure 14: figure from a Enright test we did, the surface is constructed using Improved Blobbies. The green boxes in 14(b) shows the dirty blocks and where the yellow box shows the actual size of the grid. The level set has the same resolution as the particle field.

In table 6 we present different iteration method that can be used to loop through a field. The test consisted of iterating through a fully alocated scalar field and print its values.

Particles	$w \times h \times d$	0 threads	2 threads	4 threads	6 threads	8 threads
46777	32^3	50.760 s	25.653 s	13.702 s	9.670 s	8.014 s
375835	64^3	405.277 s	206.720 s	106.751 s	72.629 s	55.243 s

Table 5: One iteration of PIC using Python

$w \times h \times d$	While loop	For-loop	Optimized for-loop	Wrapped for loop
32^3	0.299198 s	0.207685 s	0.210562 s	0.008156 s
64^3	2.176877 s	1.598500 s	1.685689 s	0.063166 s
128^3	18.351317 s	13.765721 s	13.870668 s	0.501051 s
256^3	147.008148 s	103.344957 s	104.519476 s	3.938097 s

Table 6: Different iteration method that can be used in the wrapped Python module

5 Discussion

The following section will discuss the difference between using PIC and FLIP, which of the way of storing attributes works best and also the use of Partio for both attribute manipulation and saving to disk. We will also discuss the script language Python. What advantages and disadvantages there are with having script language support.

5.1 PIC vs FLIP

Since the surface is not necessary for the PIC/FLIP method as it is for a Eulerian fluid simulation. Simulation can be run on a sparser grid than a Eulerian fluid and achieve similar result. It also benefits that the resolution of the surface can be decide after the simulation is finished, within a certain span. For example you cannot run a simulation at 8^3 and create a surface using a level set with the resolution 256^3 and expect it to look good and have a lot of fine detail.

We mention in our implementation that PIC suffers from numerical dissipation causing a lot of details to be smoothed out. FLIP also has its own issues, one is that it can develop noise. Since we have around eight particles in each cell, which gives that the particles have more degrees of freedom than the grid. Results in that particles velocity can cancel each other in one time step causing the particles to vanish from the grid. In the next time step pop up, causing velocity fluctuation on the grid. Bridson suggest, in [5], to reduce this artifact by blending a small amount of the PIC update with FLIP and hopefully not introduce to much numerical dissipation. Since PIC do not have this problem due to the quantities are interpolated from the grid and not incremented.

As seen in figure 11 FLIP is more turbulent than the PIC method, and a lot of the times when we tried the FLIP method it showed tendencies to become unstable. We have not had the time to inspect this further to see what cause this. If there is an implementation error somewhere or if there is something with the kernel that introduce more energy to the simulation.

Looking at the benchmark tables our implementation scales really well with the number of threads used. And what takes up most of the time is the surface reconstruction and saving the particles to disk. It is quite understanding that the surface reconstruction takes time since it spends most of the time looking up values in the grid. Each evaluated grid point in the level set checks all adjacent particles in a search area of $3 \times 3 \times 3$

cells. Which gives a time complexity of $O(n \times m^3)$ where n is number of grid points evaluated and m is the number of grid points in the search area in one dimension.

One issue we noticed when using reseeding to prevent holes and noise in the fluid is that it can add volume to the fluid causing it to grow. For example a small chunk of fluid is poured on top of a sphere. When it starts to run along the sides of the sphere, the particle count will become low in some parts (a cell will have less than three particles) and new particles will be created to fill these holes. Thus the fluid volume will start to grow, and engulf the whole sphere.

One solution might be to create particles for the surface reconstruction only, i.e. the created particles are not added to the simulation. But with this comes new issues, e.g. how to store these help particles and to keep track of them without slowing down the rest of the simulation. And this does not solve the issue with having to many particles in a cell. Since if we only remove particles, the fluid will start to loose volume instead.

5.2 Three ways of storing attributes

A particle that just store the position, radius, velocity and unable to add other attributes (*non-dynamic particle*) takes 24 bytes (192 bits) of memory with no quantization. Johansson [15] showed that if we use Semi-Compact implementation we can represent the radius and position with 63 bits thus the total amount of memory per particles becomes 159 bits $\approx$ 20 bytes. When using particles that can add attributes (Dynamic and semi dynamic) we use an ID to keep track of the, an int64 (64 bits) to be precise. This brings the total amount of memory per particle to be at minimum 223 bits $\approx$ 28 bytes for the semi-dynamic particle. Since it stores the radius, position and the velocity on the particle same as the non-dynamic. For the dynamic particle, all of its attributes are stored in the Partio data object hence we cannot use a semi-compact representation. This leaves us to store all values as floats. The minimum amount of memory for these particles will be 256 bits = 32 Bytes.

For the scene we used to test the PIC and FLIP method at resolution 128^3 using around 3 million particles. The total amount of memory for the particles will be: $159 * 3\text{M} = 477\,\text{Mbit} \approx 455$ MB for the non-dynamic particles. For the semi dynamic particles the total amount of memory will be $223 * 3\text{M} = 669\,\text{Mbit} \approx 638$ MB. And for the dynamic particles: $256 * 3\text{M} = 768$ MBit $\approx$ 732MB. Using particles that can dynamically add and remove attributes but stores the most important attributes on the particle will take ruffly 28.7% more space than non-dynamic particles. and if we use dynamic particles the increase of memory use will be around 37.9%.

The total execution time will also be affected if we use dynamic particles since an extra memory look up will be necessary to get the right attribute. For example we want to know a particle's velocity, for non-dynamic and semi-dynamic we access it directly. But for the dynamic particle we first must retrieve the ID then use it to look up the attribute in the Partio data object. Therefore to get a good trade of for speed and flexibility we would recommend to use semi-dynamic particles. But the dynamic particles are not fixed to only be used as fluid particles they can easily be used as particles in a particle level set.

The best way would probably be to store only position, radius and the ID on the particle and add other attributes if necessary using Partio.

5.3 Disney's open source particle I/O

Apart from the surface reconstruction, the next step that takes up most of the time for the fluid simulation is the "Save to disk" step. Looking at table 4 and 3 we can see that the Partio conversion is only a fraction of the total time for the step. The rest is spent on saving the particles to disk. Which we can see in table 3 that the difference between using 8 threads and running it in serial is not that much. Using Partio for handling the I/O part has its advantages as well as disadvantages. Since it supports a lot of particle file formats we do not need to implement any thing ourself but the downside to this is that we instead need to convert our particle field to a Partio object every time we want to save our particles to disk. And the second is that we cannot optimize the step further without writing our own particle input/output method which we then need to maintain.

This also affect the attribute manipulation since Partio does not support removing particles from the data object only adding to it. Which force us to use a Partio conversion even when we use a Partio object for handling the attributes in the Particle field. If we just printed the Partio object for the Particle field we could have particles that had been removed from the particle field still be visible if we load it into Houdini. One solution for this would be if Disney added lazy removal of the particles i.e. have a flag (bit) that if true (1) the particle exist and if false (0) the particle is deleted.

5.4 Python script

If we compare the two PIC method tests in table 1 and table 5 we can see that the Python script is much slower to execute than the C++ implementation, the C++ implemetation is 11 times faster than the Python script. This mostly due to that the C++ implementation is highly optimized during compilation using intel's compiler version 12, where as Python interpret the code on the fly when executing the script. Therefore it cannot spend to much time on optimizing the function calls. Another is that there is a lot of function member pointers that must be resolved during the script execution. These might be the reason for the Python script to be so slow. And as of now only works for testing parameters, and different setups. Not to do any actual simulation due to that it is to slow.

We can also see from table 6 that iterations in python are slow compared to the wrapped for loop. The slowest loop is the while loop in Python since it uses the wrapped iterators to loop through the field and therefore must evaluate the pointers to the iterators each time. The different between the for-loop and the "Optimized" for-loop, is that the "Optimized" for-loop avoids using the dot operator. Instead it stores the function calls to local variables. Which actual takes longer time then the normal for-loop, and the reason for calling it an optimized for-loop is that on python's wiki [33] it is suggested that this runs faster than the normal for-loop. But in our case this is false, so do not trust everything that is written on that wiki, but try it yourself. The absolute fastest way to iterate through the field is to write a function in C++ that does this and simply wrap it to Python. Therefore to avoid bottlenecks in the Python script we need to have as much functions as we can to be executed inside wrapped functions. But if we hide to much functions, the flexibility of using a python script will be rather limited. And looses its similarity with the C++ implementation thus makes it harder to translate between the two languages. Since the Boost.Python API is quite new to us, we might have missed a lot of ways to optimize the Python module. Ideally would be if we could use the Python script for both testing

and simulating without getting to much time penalty.

6 Conclusion

We implemented Python script support for the fluid framework as well as other frameworks that uses Digital Domain's DB-Grid as base. Which turns out that it is to slow for actual simulation but good for testing due its quick edit-test-debug cycle.

We have shown how to implement a PIC/FLIP method on a sparse block-optimized grid data structure, DB-Grid, and take advantage of the inherent parallelism that comes with this grid. And the results show that it scales really well with the use of multiple threads.

We also discussed the use of having the ability to dynamically add/remove attributes for the particles. And show that a fully dynamical particle will be flexible but will use more memory and will be slower than the more specialized particle. The best way will probably be to have a hybrid of these two as mention in the discussion.

6.1 Future work

Even if we got a fluid simulation running there is still lot to do and to improve before it can be used in production. Looking at table 3 we can see that the things needed to be optimized is the save to disk part and surface reconstruction. One thing to further analyze how the algorithm behaves is to use a similar methodology as O'Neill et.al. did in [25] where they optimize Dreamworks fluid simulation. Where they analyze the programs performance in overall system use, how often the program performs I/O operations, and its overall memory foot print.

6.1.1 Substitute Partio

Optimizing the I/O part can be hard since it most depends on what hard drive is used, and due to the writing to disk is perform sequentially. But what we can do is to write our own particle I/O that better fits our grid structure, which will make the conversion step unnecessary.

6.1.2 Automate C++ to Python

Wrapping of our C++ tools, involves a lot of repetitive and tedious work and can be easily be automated to some degree with a script of some kind. Similar to the way the automated documentation generator *Deoxygen* works [30]. We do not want to expose all functions in a class to the user, therefore some human intervention will be necessary to have, for example to expose a class could look like something like this:

```
// wrap class
/* \#Expose_class
*/
class SomeClass
{
...

// wrap function
/* \#Expose_func
*/
void some_function() { ... }
```

```
// skip function
void other_function() { ... }

};
```

similar to Dyoxygen's \Brief and not to collide with any Deoxygen commands, we could use \# in front of the commands for the wrapper script instead.

6.1.3 Python module

As of now we bake everything into one module *Pyclone* and even if we have separate all of the classes into separate files for the wrapping for improving the compile time as mention in section 3.6.3. The compile time is still high, around 30+ min. A better way might to divide them into smaller modules and then make a package of them instead.

6.2 Surface reconstruction

The Improved Blobbies method generates a better result than Blobbies but it still has visible bumps on flat surfaces. To improve the surface we could implement a method presented by Yu and Turk in [36]. Where they use anisotropic kernels instead of isotropic kernels to reconstruct the surface. Their results looks promising, and the method seems to produces very smooth surfaces.

References

[1] D. Adalsteinsson and J. A. Sethian. The fast construction of extension velocities in level set methods. *J. Comput. Phys.*, 148:2–22, January 1999.

[2] Bart Adams, Mark Pauly, Richard Keiser, and Leonidas J. Guibas. Adaptively sampled particle fluids. In *ACM SIGGRAPH 2007 papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[3] James F. Blinn. A generalization of algebraic surface drawing. *ACM Trans. Graph.*, 1:235–256, July 1982.

[4] J U Brackbill and H M Ruppel. Flip: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions. *J. Comput. Phys.*, 65:314–343, August 1986.

[5] Robert Bridson. *Fluid Simulation for Computer Graphics*. A K Peters/CRC Press, September 2008.

[6] Liang T. Chen. The challenge of race conditions in parallel programming.

[7] Don Clugston. Member function pointers and the fastest possible c++ delegates. http://www.codeproject.com/KB/cpp/FastDelegate.aspx. Retrived 2011-08-18,.

[8] Joel de Guzman and David Abrahams. Boost.python: Tutorial introduction. http://www.boost.org/doc/libs/1_47_0/libs/python/doc/tutorial/doc/html/index.html. Retrived 2011-08-18.

[9] Mathieu Desbrun and Marie-Paule Gascuel. Smoothed particles: a new paradigm for animating highly deformable bodies. In *Proceedings of the Eurographics workshop on Computer animation and simulation '96*, pages 61–76, New York, NY, USA, 1996. Springer-Verlag New York, Inc.

[10] Python Software Foundation. The python tutorial. http://docs.python.org/tutorial/index.html. Retrived 2011-08-17.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education.

[12] F. H. Harlow. The particle-in-cell method for numerical solution of problems in fluid dynamics. 1963.

[13] Francis H. Harlow and J. Eddie Welch. Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with Free Surface. *Physics of Fluids*, 8(12):2182–2189, 1965.

[14] Intel. Threading building blocks. http://software.intel.com/en-us/articles/intel-tbb/. Retrived 2011-08-05.

[15] John Johansson. Efficient implementation of the particle level set method. Master's thesis, Linkoping University, Media and Information Technology, 2010.

[16] Edward A. Lee. The problem with threads. *Computer*, 39:33–42, May 2006.

[17] Boost C++ libraries. Boost shared_ptr class template. http://www.boost.org/doc/libs/1_47_0/libs/smart_ptr/shared_ptr.htm. Retrived 2011-08-18.

[18] Boost C++ libraries. Boost.python. http://www.boost.org/doc/libs/1_47_0/libs/python/doc/. Retrived 2011-08-17.

[19] Boost C++ libraries. Boost.python: boost/python/class.hpp. http://www.boost.org/doc/libs/1_37_0/libs/python/doc/v2/class.html. Retrived 2011-08-18.

[20] Boost C++ libraries. Boost.thread. http://www.boost.org/doc/libs/1_47_0/doc/html/thread.html. Retrived 2011-08-01.

[21] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '03, pages 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[22] Ken Museth. An efficient level set toolkit for visual effects. In *SIGGRAPH 2009: Talks*, SIGGRAPH '09, pages 5:1–5:1, New York, NY, USA, 2009. ACM.

[23] Ken Museth and Michael Clive. Cracktastic: fast 3d fragmentation in "the mummy: Tomb of the dragon emperor". In *ACM SIGGRAPH 2008 talks*, SIGGRAPH '08, pages 61:1–61:1, New York, NY, USA, 2008. ACM.

[24] M. Mller N. Chentanez. Real-time eulerian water simulation using a restricted tall cell grid. SIGGRAPH 2011. ACM, 2011.

[25] John J. O'Neill, Charles Congdon, Ram Ramanujam, Silviu Borac, and Ron Henderson. A performance optimization study for the dreamworks animation fluid solver. http://software.intel.com/en-us/articles/a-performance-optimization-study-for-the-dreamworks-animation-fluid-solver/. Retrived 2011-08-24.

[26] Stanley J. Osher and Ronald P. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Springer, 1 edition, October 2002.

[27] C++ reference. priority_queue. http://www.cplusplus.com/reference/stl/priority_queue/. Retrived 2011-08-23.

[28] Jos Stam. Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99, pages 121–128, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[29] Walt Disney Animation Studio. Partio: Particle i/o api. http://www.disneyanimation.com/technology/partio.html.

[30] Dimitri van Heesch. Deoxygen. http://www.stack.nl/ dimitri/doxygen/. Retrived 2011-08-22.

[31] Guido van Rossum. Foreword for "programming python" (1st ed.). http://www.python.org/doc/essays/foreword/. Retrived 2011-08-17.

[32] Guido van Rossum. Python programming language official website. http://www.python.org/. Retrived 2011-08-17.

[33] Python wiki. Performance tips. http://wiki.python.org/moin/PythonSpeed/PerformanceTips. Retrived 2011-08-25.

[34] Brent Warren Williams. Fluid surface reconstruction from particles. MSc thesis, 2008.

[35] Magnus Wrenninge. Field3d: An open source file format for voxel data. https://sites.google.com/site/field3d/.

[36] Jihun Yu and Greg Turk. Reconstructing surfaces of particle-based fluids using anisotropic kernels. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, pages 217–225, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[37] Hongkai Zhao. A fast sweeping method for eikonal equations. In *Mathematics of Computaion*, volume 74, pages 603–627, 2004.

[38] Yongning Zhu and Robert Bridson. Animating sand as a fluid. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 965–972, New York, NY, USA, 2005. ACM.