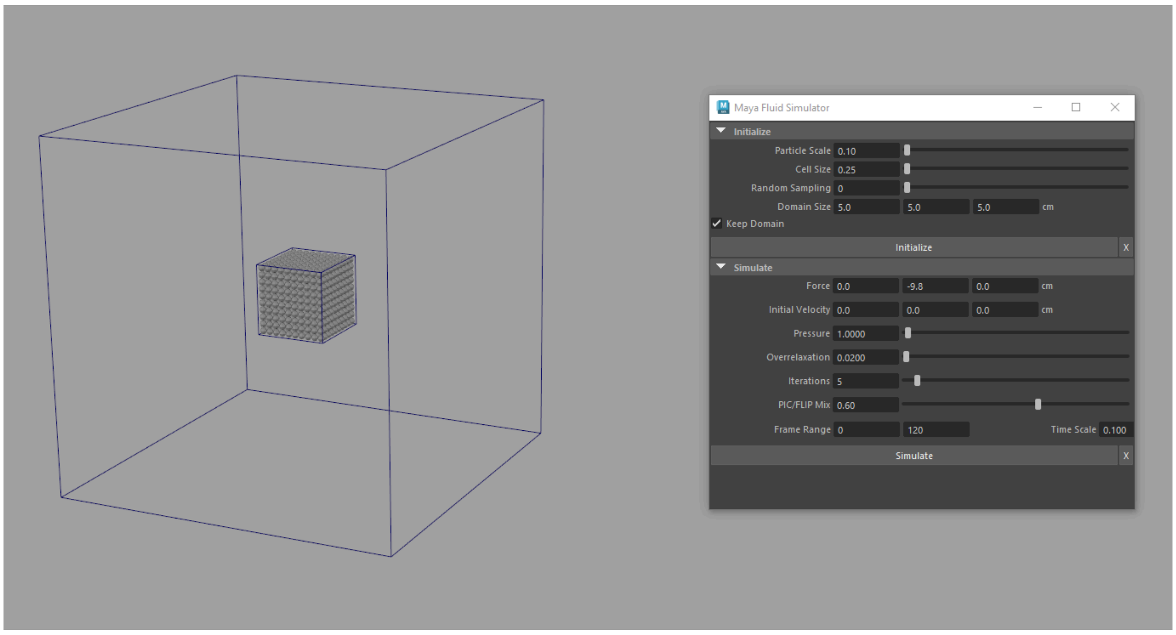
Maya Fluid Simulator

L4 Technical Arts Production
2023-2024



Particle Scale	0.10			
Cell Size	0.25			
Random Sampling	0			
Domain Size	5.0	5.0	5.0	cm
✓ Keep Domain				
Force	0.0	-9.8	0.0	cm
Initial Velocity	0.0	0.0	0.0	cm
Pressure	1.0000			
Overrelaxation	0.0200			
Iterations	5			
PIC/FLIP Mix	0.60			
Frame Range	0	120	Time Scale	0.100

1. User Manual

Maya Fluid Simulator is a PIC/FLIP tool that allows you to simulate particle fluids.

1.10 Installation

Before running the tool, you must install Numpy into your Maya distribution. Numpy is a popular choice when programming simulations as it can be 5 to 100 times faster than regular Python lists (Verma 2020). To install Numpy, run the code below in a terminal. Make sure that you edit the path to fit your Maya version.

Windows:

```
"C:\Program Files\Autodesk\Maya2023\bin\mayapy.exe" -m pip install --user numpy
```

Linux:

```
/usr/autodesk/maya2023/bin/mayapy -m pip install --user numpy
```

Mac:

```
/Applications/Autodesk/maya2023/Maya.app/Contents/bin/mayapy -m pip install
–user numpy
```

Once Numpy has finished installing, you can start Maya and load the script. When you hit run, a panel titled “Maya Fluid Simulator” should appear on the Maya top bar.

1.20 Usage

To use the tool, open the user interface by pressing *Maya Fluid Simulator -> Open Maya Fluid Simulator*. You’ll see several options and buttons appear. A video on how to use *Maya Fluid Simulator* can be found in *artefacts/videos/MFS_tutorial.mp4*

Initialize	
Particle Size (0.1)	The size of the particles (diameter)
Cell Size (0.25)	Size of cells for collision searches and velocity transfer. A larger cell size than the particle size can result in a stepped look.
Random Sampling (0)	The number of particles to randomly sample inside the source object cells. A value of 0 will evenly distribute particles within the source object.
Domain Size (5, 5, 5)	The size of the simulation domain

Keep Domain (True)	Keep the domain when re-initializing points.
Initialize | x	Create particles in the source object and create a domain object | Delete all the generated artefacts

<u>Simulate</u>	
Force (0, -9.8, 0)	The amount of global force that acts on the particles
Initial Velocity (0, 0, 0)	The amount of initial particle velocity
Pressure (0.1)	The amount of pressure divergence. Large values will force particles apart faster.
Overrelaxation (0.02)	The amount of velocity divergence. Be careful setting this value too high.
Iterations (5)	The number of iterations for solving the divergence. Higher iterations will spread particles out more, filling up more volume.
PIC / FLIP Mix (0.6)	Blending between PIC (smooth) / FLIP (splashy)
Frame Range (0, 120)	The start and end frames for the simulation
Time Scale (0.1)	The speed of the simulation
Simulate | x	Simulate the fluid | Clear the simulation

2. Code Breakdown

2.1 Navier Stokes, PIC, and FLIP

Maya Fluid Simulator relies on the Navier Stokes equations. The equations state that the divergence of the fluid (its compression) is always 0 and that the acceleration of the fluid is related to its pressure gradient, viscosity, and external forces. (Harris 2007)

$$\nabla \cdot \overline{u} = 0$$

$$\rho \frac{D\overline{u}}{Dt} = -\nabla p + \mu \nabla^2 \overline{u} + \rho \overline{F} \tag{1}$$

Figure 1. Navier Stokes Equations

There are numerous ways to implement the Navier Stokes equations into a fluid simulator, aside from SPH (Smoothed Particle Hydrodynamics), two of the most popular methods are PIC (Particle in Cell), and FLIP (Fluid Implicit Particles). Both methods combine Lagrangian simulation with Euler simulation.

The Lagrangian simulation method relies on particle movement to describe fluid motion. This is often more realistic as fluid is (fundamentally) made of particles. However, this method can be extremely computationally expensive when calculating divergence and solving the pressure gradient. The Euler method combats this computational time by storing velocity, density, and other attributes within a grid. This is often the preferred method for gas-based simulations such as fire and smoke, as it is much faster to calculate. However, the particle-like nature of the fluid is lost. (Englesson et al. 2011)

PIC and FLIP combine the particle-like nature of lagrangian simulations with the speed of Eulerian. The method is explained below.

1. Particle velocities are transferred to the grid and density is calculated. The velocity grid is then copied
2. A timestep is calculated
3. Boundaries are enforced
4. Fluid is made divergence-free
5. Velocity is transferred to the particles using the old and new velocity grids
6. Particle collisions are handled

2.2 Particle to Grid and Grid to Particle

One of the most important parts of the algorithm was transferring velocity between the particles and the grid. This was done through trilinear interpolation on a staggered grid. A staggered grid offsets the position of certain attributes so that they can be interpolated and averaged correctly. (Braley and Sandu 2009)

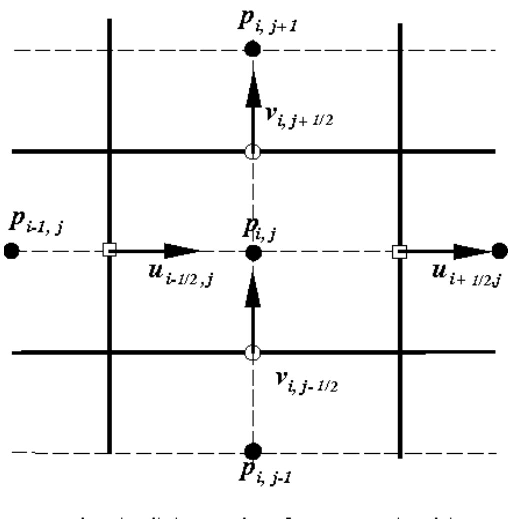


Figure 2. A 2-dimensional example of a staggered grid. Velocities are stored on the edges of cells, while pressure is stored in the centre of cells. (Kampanis and Ekaterinaris 2005)

Pressure was stored in an array the size of the domain resolution, whilst velocities were stored in arrays with a size +1 on their corresponding index.

```
self.velocity_u = np.zeros((self.resolution[0]+1, self.resolution[1], self.resolution[2]), dtype="float64")
self.velocity_v = np.zeros((self.resolution[0], self.resolution[1]+1, self.resolution[2]), dtype="float64")
self.velocity_w = np.zeros((self.resolution[0], self.resolution[1], self.resolution[2]+1), dtype="float64")
self.pressure = np.zeros((self.resolution[0], self.resolution[1], self.resolution[2]), dtype="float64")
```

The weights were calculated by finding the particle offset in cell space (*dx = x - int(x)*), and then multiplying by an *in_bounds()* check. This means any velocity values outside the domain (which don't exist), don't contribute to the trilinear interpolation.

```
c000 = (1 - dx) * (1 - dy) * (1 - dz) * self.in_bounds(i, j, k, resolution[0], resolution[1], resolution[2])
```

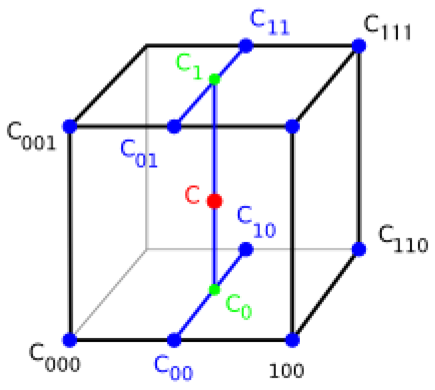


Figure 3. A visual example of tri-linear interpolation (Wikipedia 2023)

The first line is how velocity would be transferred from a particle to the grid. The second line is the inverse; grid to particle.

```
self.velocity_v[min(i, self.resolution[0] - 1)][min(j, self.resolution[1])][min(k, self.resolution[2] - 1)] +=
p.velocity[1] * c000

velocity_v += self.velocity_v[min(i, self.resolution[0]-1)][min(j, self.resolution[1] )][min(k, self.resolution[2]
-1)] * c000
```

To keep the simulation stable, velocities must be normalised by the sum of their weights. An average pressure value must also be calculated for the first frame of the simulation. This is very important for making sure the divergence calculation is correct.

2.3 Finding dt

With Lagrangian simulations, a timestep can remain constant as the particles aren't traversing any grids. In PIC and FLIP, however, the particles cannot skip over cells, as doing so may destabilise the simulation. Therefore, a timestep needs to be calculated. This concept is the CFL (Courant–Friedrichs–Lewy) condition. The technique implemented below was inspired by Robert Bridson. (Braley and Sandu 2009)

```
def calc_dt(self, particles, timescale, external_force):
    max_speed = 0

    for particle in particles:
        speed = np.linalg.norm(particle.velocity)
        max_speed = max(max_speed, speed)

    max_dist = np.linalg.norm(self.cell_size) * np.linalg.norm(external_force)

    if (max_speed <= 0):
        max_speed = 1

    return min(timescale, max(timescale * max_dist / max_speed, 1))
```

The function finds the fastest particle and calculates a timestep so that the particle would move only 1 grid cell. A timescale variable is used so users can set the visual speed of the simulation.

2.4 Enforcing Boundaries

Enforcing boundaries is extremely simple. Using the Neumann Boundary condition, all boundary velocity components that point out of the domain are set to 0. The loss of velocity is then corrected when the divergence is solved. (Englesson et al. 2011)

2.5 Solving Divergence

There are numerous ways to make the fluid velocity field divergence-free, but the easiest to implement is the Gauss-Seidel method (a modern version of the Jacobi method). Divergence is calculated by finding the velocity difference in each cell, then a pressure force is subtracted. (Müller 2022)

```
divergence = overrelaxation * (
                    (self.velocity_u[i+1][j][k] - self.velocity_u[i][j][k]) / self.cell_size[0] +
                    (self.velocity_v[i][j+1][k] - self.velocity_v[i][j][k]) / self.cell_size[1] +
                    (self.velocity_w[i][j][k+1] - self.velocity_w[i][j][k]) / self.cell_size[2]
                ) - stiffness * (self.pressure[i][j][k] - rest_pressure)
```

A border value is then calculated to spread the divergence evenly along each velocity component.

```
self.velocity_u[i][j][k] += divergence * self.in_bounds(i-1, j, k, self.resolution[0], self.resolution[1],
self.resolution[2] ) / borders

self.velocity_u[i+1][j][k] -= divergence * self.in_bounds(i+1, j, k, self.resolution[0], self.resolution[1],
self.resolution[2]) / borders
```

2.6 Particle Collisions

Two types of collisions need to be handled in the simulation: domain and particle. Handling domain collisions is relatively simple. Integrate the particle, then check if its position is outside the domain. If so, zero its velocity (across the axis), and set the particle position to the boundary edge.

```
if (x < min_x):
        x = min_x
        vx = 0
```

Negating the velocity is also an option. However, *Maya Fluid Simulator* would need to simulate more frames for the fluid to reach a rest state.

Although not required, implementing particle collisions helps stabilise the simulation and maintains the fluid volume. Generally, particle collisions are detected by checking if the distance between two particles is less than their radii combined. Particles that do collide are de-intersected and have their velocities swapped.

```
if (dist < r1+r2):
      temp_velocity = particle.velocity
      particle.velocity = other.velocity
      other.velocity = temp_velocity

      overlap = (pscale) - dist

      if (dist > 0):
            dx /= dist
            dy /= dist
            dz /= dist

      move = [dx * overlap * 0.5, dy * overlap * 0.5, dz * overlap * 0.5]

      particle.position -= move
      other.position += move
```

As the number of particles increases, the time needed to calculate particle collisions increases dramatically. To combat this, Hash mapping was implemented. Hash mapping assigns close particles to the same array, heavily reducing the number of collision tests between particles. (Rotenberg 2017)

2.7 PIC and FLIP

Although PIC and FLIP follow (essentially) the same code structure. The way they affect particle velocity is slightly different. For PIC, velocity from the grid is directly assigned to the particle velocity. For FLIP, a velocity difference is calculated between the old grid (particle to grid) and the grid after all the force and divergence calculations. The velocity difference is then added to the particle velocity. (Englesson et al. 2011)

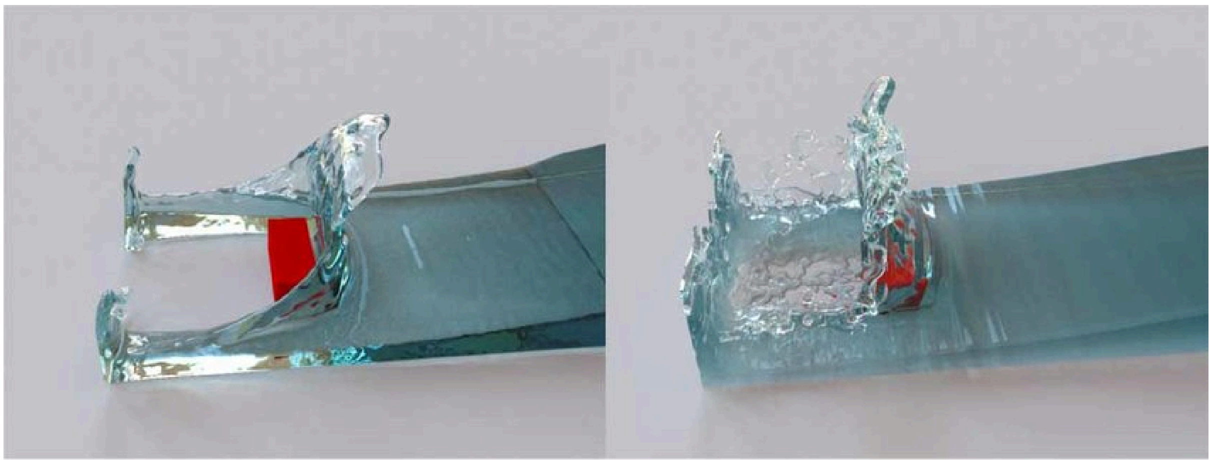


Figure 4. A visual difference between PIC (left) and FLIP (right) (Salomonsson 2011)

PIC is usually more viscous, and FLIP is usually more splashy. The purpose of the PIC/FLIP slider is so that users can choose how much viscosity they want the fluid to have. This is also why the viscosity force was left out of the code.

When interpolating the velocities from the grid, particles are often integrated backwards by their velocity. *Maya Fluid Simulator* uses simple Backward Euler integration, however, more accurate methods such as Runge-Kutta exist. (Englesson et al. 2011)

2.8 Program Structure

The code for *Maya Fluid Simulator* is split into two main sections. The tool, which takes up the first chunk of code, and the simulator, which takes up the second chunk.

The tool is what interfaces with Maya, It's built into a class so that it can have localised global variables. It deals with creating the user interface, managing particles and domains, and starting the simulation. Below are the associated functions with the tool.

class MFS_Plugin()

__init__
MFS_create_menu
MFS_popup
MFS_delete_menu
MFS_initialize
MFS_simulate
keyframe
MFS_delete
MFS_reset
get_active_object
can_simulate
mesh_to_points
is_point_inside_mesh

The simulator is not linked to Maya at all. It creates particle data types and does its simulation within the *MFS_Grid* class, only referencing the variables from the user interface. This way of working makes it very easy to know what issues are from Maya, and what are from the simulator.

class MFS_Particle()

__init__
integrate

class MFS_Grid()

__init__
particles_to_grid
average_pressure
in_bounds
get_trilinear_weights
calc_dt
apply_forces
enfource_boundaries
solve_divergence
grid_to_particles
handle_collisions_and_boundary
get_velocity
get_grid_coords
insert_particles_into_hash_table
get_particles_from_hash_table
hash_coords
clear

Further explanations of all the functions can be found within the Python code.

While implementing *Maya Fluid Simulator*, a 2-dimensional version of the code was written in javascript to help with debugging. The program can be found in *src/other/picflip2d.html*

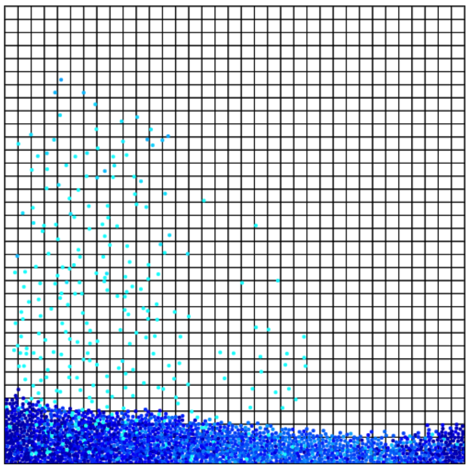


Figure 5. A 2D Javascript implementation of the simulator

2.9 Sourcing Objects

One algorithm that deserves recognition is the sourcing algorithm used in *MFS_Plugin.initialise()*. When an object is selected and initialised, it becomes a source object and its name is used for its corresponding domain and particles. The domain centre is at the centre of the source object, and its cell numbers are stored in the subdivisions of the grid.

When sourcing the particles, the tool looks at the bounds of the source object and creates a 3-dimensional point grid inside of it. Each particle is then checked to be within the source object. To check whether a point is inside a mesh, a ray is cast from the point in a random direction. The number of intersections between the ray and the source object is then counted, and if there is an odd number of intersections then the point is inside the mesh. This method only works when the source object is a closed mesh. There is also a possibility that an outside ray could touch the boundary object, however, the chances of that happening in this program are very low and can be easily dealt with by the user.

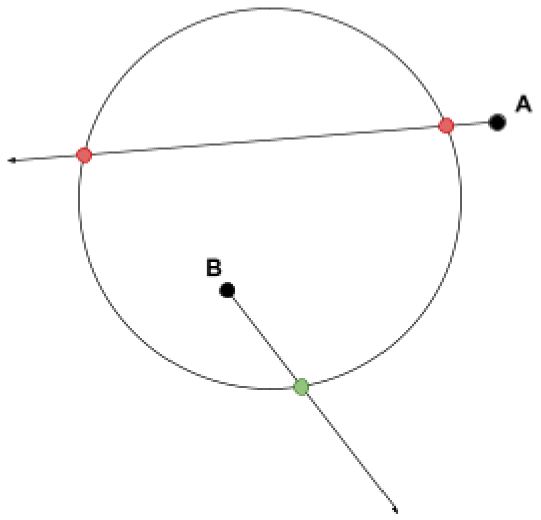


Figure 6. Point A is outside the mesh as there are even ray intersections. Point B is inside the mesh as there are odd ray intersections.

The code for testing if a point is inside a mesh is referenced below, as it is an extremely powerful function.

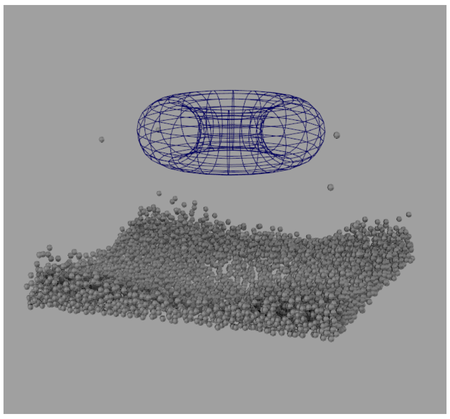
```
def is_point_inside_mesh(self, point, mesh):
    ''' is_point_inside_mesh checks if a point is inside a specific mesh.
    To do this, a ray is fired from (at a random direction) the point.
    If the ray intersects with the mesh an uneven number of times, the point is inside the mesh.

    point      : the point position
    mesh       : the mesh to check if the point is in

    This only works if the mesh is enclosed, any gaps in the geometry can lead to unstable results.

    There is also a slight chance that a ray could 'touch' the mesh, resulting in the algorithm thinking an
outside particle is inside the object.
    '''
    # Random ray direction
    direction = om.MVector(random.random(), random.random(), random.random())

    selection_list = om.MSelectionList()
    selection_list.add(mesh)
    dag_path = selection_list.getDagPath(0)

    fn_mesh = om.MFnMesh(dag_path)
    intersections = fn_mesh.allIntersections(
        om.MFloatPoint(point),
        om.MFloatVector(direction),
        om.MSpace.kWorld,
        999999,
        False
    )

    if (intersections is not None):
        num_intersections = len(intersections[0])
        return num_intersections % 2 != 0

    return False
```

3. Conclusion

3.1 Results

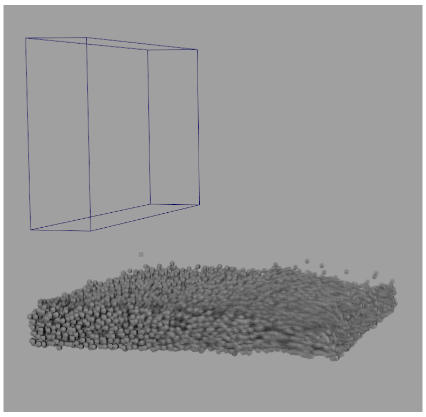
Below are some benchmark tests for *Maya Fluid Simulator*. Tests were run on Windows 10 Enterprise with an Intel i7-13700, 64.0 GB RAM, and NVIDIA GeForce RTX 4080.

<u>Default Donut</u>
Particle Scale: 0.1
Cell Size: 0.2
Domain: 5x5x5
Force: (0, -9.8, 0)
Velocity: (0, 0, 0)
Pressure: 1.0
Overrelaxation: 0.02
Iterations: 5
PIC/FLIP Mix 0.6
Frame Range: 0-120
Timescale: 0.1



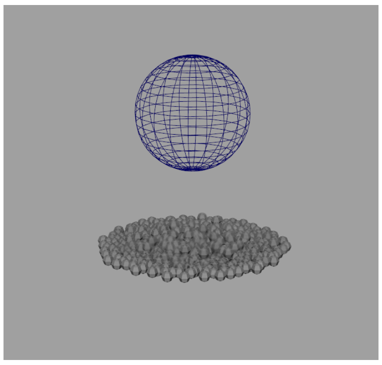
Particle Count: 4800
Simulation Time: 3 Minutes 33 Seconds

<u>Dam Break</u>
Particle Scale: 0.1
Cell Size: 0.2
Domain: 5x5x5
Force: (0, -9.8, 0)
Velocity: (0, 0, 0)
Pressure: 1.0
Overrelaxation: 0.02
Iterations: 5
PIC/FLIP Mix 0.8
Frame Range: 0-120
Timescale: 0.1



Particle Count: 15000
Simulation Time: 9 Minutes 20 Seconds

<u>Honey</u>
Particle Scale: 0.2
Cell Size: 0.4
Domain: 5x5x5
Force: (0, -9.8, 0)
Velocity: (0, 0, 0)
Pressure: 0.2
Overrelaxation: 0.0001
Iterations: 1
PIC/FLIP Mix 0.0
Frame Range: 0-120
Timescale: 0.1

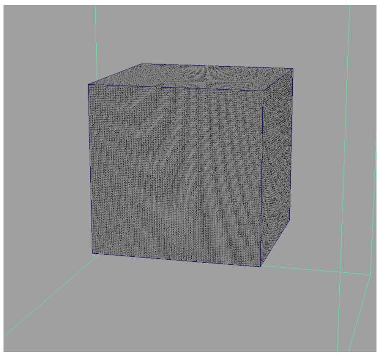


Particle Count: 520
Simulation Time: 1 Minute 8 Seconds

<u>1 Million Particles</u>
Particle Scale: 1.0
Cell Size: 5.0
Domain: 200x200x200

No Simulation was attempted
as loading the scene file
took ~20 minutes.

This is due to Maya being unable to handle a large
number of objects in the viewport.



Particle Count: 1000000
Generation Time: ~40 Minutes

Videos of the simulations can be found in */artefacts/videos*. The scene files can be found in */artefacts/scenes/*.

3.2 Limitations

There are a few limitations to the program. The first is the instability of the simulator. Due to errors in the divergence calculation, high values of overrelaxation, or too many iterations can cause the fluid to behave strangely. As well as this, particles will pop at times in the simulation. Ideally, improving the divergence calculation and removing the pressure, overrelaxation, and iteration settings altogether would make the tool a lot easier to use.

The second limitation is the time it takes to simulate the fluid. In further implementations, GPU support as well as a faster divergence calculation method would be included.

The third limitation, which is more of a problem with Maya, was not being able to have relative image paths. Although a set-project method could work, It can be quite limiting as users may need to set-project elsewhere depending on their scene. Therefore, at the present moment, *Maya Fluid Simulator* does not have any icons or images associated with it.

Other limitations are small issues that, with time, will be sorted as the code evolves.

Although the program is already quite comprehensive, plans for the future would be to implement inflow objects, colliders, and a meshing solution so that the fluids could be used more for real productions.

3.3 Conclusion

This tool was written for L4 Technical Arts Production for Computer Animation Technical Arts at Bournemouth University. There is a very high chance that the algorithms implemented are not entirely correct, therefore be weary when re-using the code. If you wish to access (or correct) the source code, you can do so here.

Reference List

Braley, C. and Sandu, A., 2009. Fluid Simulation For Computer Graphics: A Tutorial in Grid Based and Particle Based Methods.

Englesson, D., Kilby, J. and Ek, J., 2011. Fluid Simulation Using Implicit Particles Advanced Game Programming.

Guy, R., 2016. PIC/FLIP Fluid Simulation | Ryan Guy's Portfolio [online]. rlguy.com. Available from: https://rlguy.com/gridfluidsim/ [Accessed 4 May 2024].

Harris, M., 2007. Chapter 38. Fast Fluid Dynamics Simulation on the GPU [online]. NVIDIA Developer. Available from: https://developer.nvidia.com/gpugems/gpugems/part-vi-beyond-triangles/chapter-38-fast-fluid-dynamics-simulation-gpu [Accessed 7 May 2024].

Kampanis, N. and Ekaterinaris, J., 2005. 5-Figure1-1 [online]. Science Direct. Online Image. Available from: https://d3i71xaburhd42.cloudfront.net/135310e597706e8555be7693f8fcf6af06b4e333/5-Figure1-1.png [Accessed 5 May 2024].

Matthias, 2022. 18 - How to write a FLIP water / fluid simulation running in your browser [online]. www.youtube.com. Available from: https://youtu.be/XmzBREkK8kY [Accessed 4 May 2024].

Müller, M., 2022. How to write a FLIP Water Simulator [online]. Available from: https://matthias-research.github.io/pages/tenMinutePhysics/18-flip.pdf [Accessed 5 May 2024].

OpenAI, 2024. ChatGPT: Optimizing Language Models for Dialogue. Available from: https://openai.com/blog/chatgpt/. [Accessed 9 May 2024].

Rotenberg, S., 2017. Particle-Based Fluid Simulation [online]. Available from: https://cseweb.ucsd.edu/classes/wi17/cse169-a/slides/CSE169_15.pdf [Accessed 10 May 2024].

Salomonsson, F., 2011. PIC/FLIP Fluid Simulation Using a Block-Optimized Grid Data Structure [online]. Diva Portal. Available from: https://www.diva-portal.org/smash/get/diva2:441801/FULLTEXT01.pdf [Accessed 5 May 2024].

Verma, S., 2020. How Fast Numpy Really is and Why? [online]. Medium. Available from: https://towardsdatascience.com/how-fast-numpy-really-is-e9111df44347 [Accessed 7 May 2024].

Wikipedia, 2023. 3D Interpolation 2 [online]. Wikipedia. Online Image. Available from: https://upload.wikimedia.org/wikipedia/commons/thumb/9/97/3D_interpolation2.svg/220px-3D_interpolation2.svg.png [Accessed 5 May 2024].

Winkler, D., Zischg, J. and Rauch, W., 2017. Visual-comparison-of-SPH-left-and-PIC-FLIP-right-flow-simulations-the-impact-of [online]. Water Science and Technology. Online Image. Available from: https://www.researchgate.net/profile/Daniel-Winkler-4/publication/321111273/figure/fig1/AS:606985259061248@1521727971958/Visual-comparison-of-SPH-left-and-PIC-FLIP-right-flow-simulations-the-impact-of.png [Accessed 5 May 2024].

Zhu, Y. and Bridson, R., 2005. Animating sand as a fluid. ACM Transactions on Graphics, 24 (3), 965–972.