MPM-based 3D Liquid Simulation - Final Project report

CSIT6000L: Advanced Digital Design

Contents

Introduction

Context and Goals

Physical simulation is an important part of computer graphics. It’s different from the physics subject because we aim not to implement every formula in the virtual world. What we focus on is to imitate the phenomena and material attributes in the real world with a relatively small cost. This often requires many tricks and assumptions——we summarize them as models then keep refining them to gradually achieve real-time high performance simulation.

In this project, we aim to have a comprehensive understanding of modern physics simulation mainly on liquids. Then we expect to provide interactive demos for user to play with different parameters and see what will happen——inspired by the former assignment OpenSCAD——and offline-rendered demos to show how vivid the scene created by those simple particles can be.

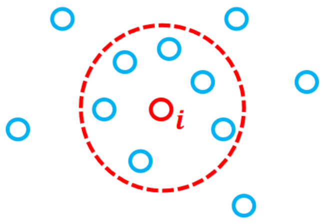
Theory Overview

Basically, to simulate a type of liquid or fluid we want to simulate their advection and projection. Advection refers to the transport or movement of a property (such as density, temperature, or velocity) by the bulk motion of the fluid. The advection equation is typically expressed as

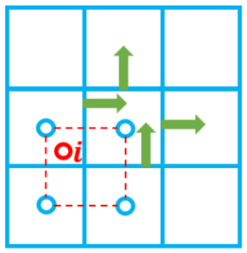
$$\frac{\partial \phi}{\partial t} + (v \cdot \nabla)\phi = 0$$

, where ϕ is the property being advected, t is time, v is the fluid velocity, and ￼ is the gradient operator. Projection in fluid simulation is a step used to enforce incompressibility in the flow field. Incompressibility means that the fluid density remains constant. This is a common assumption for many fluid simulations, especially those based on the Navier-Stokes equations. The projection step corrects the fluid velocity field to ensure that it remains divergence-free, meaning the fluid is incompressible. The projection step typically involves solving a Poisson equation to correct the velocity field, ensuring that it satisfies the continuity equation ($\nabla \cdot v = 0$) for each grid, which enforces incompressibility.

As stated, liquid/fluid is a continuous field and we can’t compute for every point. Therefore, to simply the information storation and computation, there are two classical views. Lagrangian and Eulerian views are two fundamental approaches used in fluid dynamics to describe and analyze the motion of fluid particles. These perspectives provide different ways of looking at and understanding fluid flow.



Lagrangian approach focuses on tracking individual fluid particles as they move through space and time. In the Lagrangian framework, you follow specific particles and observe how their properties (such as velocity, density, and temperature) change over time. Each fluid element has its trajectory, and its behavior is described by a set of ordinary differential equations (ODEs) that govern the motion of individual particles. It's just like the process of releasing dye into a river and tracking the movement of individual dye particles as they flow downstream.



While Eulerian approach focuses on fixed points in space and observes how fluid properties vary at those fixed points over time. It's like placing sensors at various points in a river to measure the velocity of the water at those locations over time. Unlike particle-based Lagrangian approach, it's a bit more difficult to understand how we update the information of a grid.

$$\frac{\partial u}{\partial t} + (u \cdot \nabla) u = -\frac{\nabla p}{\rho} + \mu \Delta u + f$$

Here, the terms represent the following. u : Velocity vector field of the fluid (a function of space and time); t : Time; ∇: Gradient operator; p : Pressure; ρ: Density of the fluid; μ : Kinematic viscosity (ratio of dynamic viscosity to density); f : External force per unit mass acting on the fluid. Δ :Laplacian Smoothing.

Breaking down the terms:

Transient Term $\frac{\partial u}{\partial t}$: Describes how the velocity field changes with time.

Convective Term $(u \cdot \nabla)u$: Represents the advection or transport of velocity by the fluid motion.

Pressure Term $-\frac{\nabla p}{\rho}$: Represents the pressure gradient force, driving fluid from regions of high pressure to low pressure.

Viscous Term $\mu \Delta u$: Accounts for the effects of viscosity, causing velocity gradients to smooth out over time.

External Force Term f : Includes any external forces acting on the fluid per unit mass.

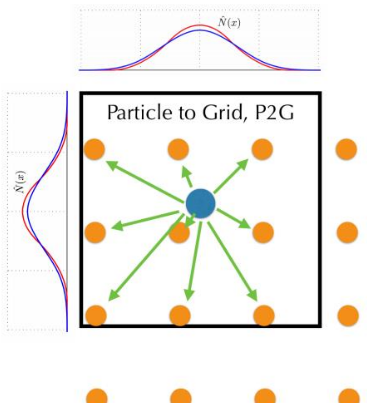
Through the updating function, we can update the information of all units at each timestep, which constructs the process of simulation.

Both methods have their disadvantages. For particle-based methods (Lagrangian), tracking individual particles through space and time is relatively straightforward. Particles naturally follow the flow, making it easy to simulate the advection process. But ensuring incompressibility (divergence-free) in a Lagrangian framework is more challenging. Maintaining a uniform distribution of particles while preventing compression or expansion is not inherently guaranteed.   For grid-based methods (Eulerian), tracking fluid elements as they move through a fixed grid can be computationally challenging, especially in turbulent flows where the grid might need frequent adjustments. Enforcing incompressibility in an Eulerian framework is more straightforward. The continuity equation can be explicitly satisfied by adjusting the grid and solving for the pressure field.
A common strategy is to combine the strengths of both approaches through hybrid methods. For example, the Material Point Method (MPM) combines Lagrangian particles with an Eulerian grid. This allows for accurate advection of material properties using particles, while the grid ensures a regular structure that facilitates efficient projection to enforce incompressibility. In each step, we scatter the information of each particle to its neighbor grids. Then we can take advantage of Eulerian methods. Next we propagate the updated information back to the particles and conduct advection. That’s the mainstream of MPM. In the next Section we’ll bring more details for the steps in MPM.

Methodology

Particle to Grid

When treating the P2G process, we scatter each particle’s momentum to its 9 grid neighborhoods. It is obvious that the particle will affect the grid point much more if it is closer to that grid. In Particle-in-Cell computing method[1], we have 3 several types of B-Spline Kernels that can be used.



Linear

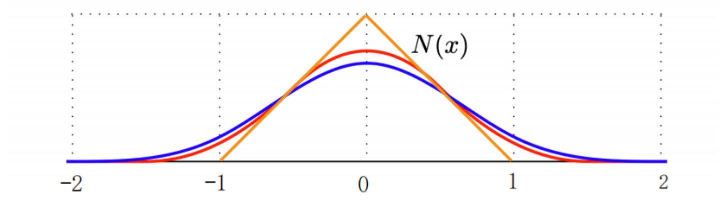
$$N(x) = \begin{cases} 1-|x| & 0 \leqslant |x| < 1 \\ 0 & 1 \leqslant |x| \end{cases}$$

Quadratic

$$N(x) = \begin{cases} \frac{3}{4}-|x|^2 & 0 \leqslant |x| < \frac{1}{2} \\ \frac{1}{2}\left(\frac{3}{2}-|x|\right)^2 & \frac{1}{2} \leqslant |x| < \frac{3}{2} \\ 0 & \frac{3}{2} \leqslant |x| \end{cases}$$

Cubic

$$N(x) = \begin{cases} \frac{1}{2}|x|^3-|x|^2+\frac{2}{3} & 0 \leqslant |x| < 1 \\ \frac{1}{6}(2-|x|)^3 & 1 \leqslant |x| < 2 \\ 0 & 2 \leqslant |x| \end{cases}$$



Since the Quadratic method has a smooth curve and requires only quadratic calculation, which does not cause numerical instability, we use the quadratic method to scatter particle’s momentum to grid.

Simulating Elastoplastic Material

Deformation Recovery Force contains two parts: elastic part and plastic part.

$$\mathbf{F}_p = \mathbf{F}_{p,\textsf{elastic}}\mathbf{F}_{p,\textsf{plastic}}$$

But the Potential Energy only depend on elastic force:

$$\psi_p^n = \psi(\mathbf{F}_{p,\textsf{elastic}})$$

To handle the elasticity, we use Corotated model [3], which in general consists of the following two formulas:

1. Potential Energy of Elastic Force (F)

   $\psi(\mathbf{F}) = \mu \sum_i(\sigma_i - 1)^2 + \frac{\lambda}{2}(J-1)^2$. σ_i are singular values of $\mathbf{F}$.

2. Derivative of Potential Energy

   $\mathbf{P}(\mathbf{F}) = \frac{\partial\psi}{\partial\mathbf{F}} = 2\mu(\mathbf{F}-\mathbf{R}) + \lambda(J-1)J\mathbf{F}^{-T}$

The J is the volume ratio which we will illustrate in the next section. The λ, μ are the Lamé Parameters, which can be regard as the coefficient of material viscoelasticity.

The main idea of plasticity is that when scaling exceeds a certain limit, no recovery will be performed. Therefore, to handle the plasticity, we maintain elastic deformation in a specific range:

$$\Sigma_{elastic} = max(min(\Sigma, B_{upper}), B_{lower})$$

The Σ is the singular value from SVD (Singular value decomposition) of Fp. This singular value is also called *scaling matrix*, which represents the deformation part of the linear transformation. B_{upper} is the plasticity upper bound. B_{lower} is the plasticity lower bound.

Grid Momentum

The whole formula of grid momentum:

$$(m\mathbf{v})_i^{n+1} = \sum_p w_{ip}\{m_p\mathbf{v}_p^n + [m_p\mathbf{C}_p^n - \frac{4\Delta t}{\Delta x^2}\sum_p V_p^0\mathbf{P}(\mathbf{F}_p^{n+1})(\mathbf{F}_p^{n+1})^T](\mathbf{x}_i - \mathbf{x}_p^n)\}$$

The calculation of grid's momentum contains two parts: **APIC momentum** and **particle elastic impulse**. The APIC (Affine Particle-in-cell method) [2] proposed a C matrix that contains more information of the movement of a particle: Basic movement, Scaling and Shearing. Let i denotes the index of grid, p denotes the particle, w_{ip} denotes the weight of particle from Quadratic B-Spline Kernels, m_p denotes the mass of the particle v_p denotes the velocity of particle, x denotes the position of particle, the **APIC momentum** is calculated using the following formula:

$$w_{ip}[m_p\mathbf{v}_p^n + m_p\mathbf{C}_p^n(\mathbf{x}_i - \mathbf{x}_p^n)]$$

Let P denotes the Derivative of Potential Energy in the Simulating Elastoplastic Material, the **particle elastic impulse** can be calculated by using this formula:

$$\Delta t\mathbf{f}_{ip} = -w_{ip}\frac{4\Delta t}{\Delta x^2}\sum_p V_p^0\mathbf{P}(\mathbf{F}_p^{n+1})(\mathbf{F}_p^{n+1})^T](\mathbf{x}_i - \mathbf{x}_p^n)$$

Let's look at how these parameters in the **particle elastic impulse** are calculated. Local velocity field is not constant, so the material keeps deforming. The new deformation gradients can be evaluated using the following formula.

$$\mathbf{F}_p^{n+1} = (\mathbf{I} + \Delta t \nabla \mathbf{v})\,\mathbf{F}_p^n$$

We use APIC's C matrix as an approximation of ∇v to transfer more information from grid to particle. Therefore, the deformation gradients can be calculate using this:

$$\mathbf{F}_p^{n+1} = (\mathbf{I} + \Delta t \mathbf{C}_p^n)\mathbf{F}_p^n$$

Let J_p denotes the volume ratio of a particular, we have

$$J_p = V_p^n / V_p^0 = \det(\mathbf{F}_p^n)$$

The deformation gradient equation can be transformed as follows and obtains a representation using the volume rate J.

$$\begin{aligned} \mathbf{F}_p^{n+1} &= (\mathbf{I} + \Delta t \mathbf{C}_p^n)\mathbf{F}_p^n \\ \Rightarrow \det(\mathbf{F}_p^{n+1}) &= \det(\mathbf{I} + \Delta t \mathbf{C}_p)\det(\mathbf{F}_p^n) \\ \Rightarrow J_p^{n+1} &= (1 + \Delta t\,\mathbf{tr}(\mathbf{C}_p^n))\,J_p^n \end{aligned}$$

Grid Operations

The grid operations can be summarized as involving two key computational steps within the Material Point Method (MPM) framework, “Grid Velocity Calculation” and “Boundary Condition Application.”

Grid Velocity Calculation: The velocity of each grid node is updated based on the momentum and mass associated with that node. This is expressed mathematically as $\vec{v}_i^{(n+1)} = \left(\left(\overrightarrow{mv}\right)_i^{(n+1)}\right) / \left(m_i^{(n+1)}\right)$, where $\left(\overrightarrow{mv}\right)_i^{(n+1)}$ represents the momentum at grid node i after the grid gathers momentum from all surrounding material points that is weighted based on the distance between particles and the grid node. $m_i^{(n+1)}$ is the mass at node i, and it is calculated using the distribution of surrounding particles' mass. The velocity is essentially the ratio of the node's momentum to its mass at the next time step n+1.

Boundary Condition Application: The updated velocity at the next time step n+1, $\vec{v}_i^{(n+1)}$, at each grid node is then modified according to the specified boundary conditions using a boundary condition function. This function adjusts the velocity to conform to physical constraints, such as solid walls where the velocity might be set to zero to simulate an immovable boundary, or other types of boundaries, such as free-slip boundary conditions. In these conditions, the material is allowed to slide along the boundary without friction. The normal component of the velocity (perpendicular to the boundary) is set to zero to prevent penetration, while the tangential component (parallel to the boundary) is unchanged, allowing the material to “slip” along the boundary. This is often used in fluid dynamics simulations where the fluid can move freely along the boundaries but cannot cross through them.

Grid to Particle

The grid-to-particle phase in the Material Point Method (MPM) is a critical part of the simulation process, following right after the grid operations. Once the grid has been updated with the latest velocities and forces, such as gravity, have been applied, the simulation needs to transfer this updated information back to the particles. This transfer ensures that the individual particles, which represent the material being simulated, are moved and deformed according to the calculated physics on the grid.

Firstly, we begin with updating the velocity of each particle. The particle velocity is a key factor in determining how a particle will move in the next time step. To update the velocity, we take a weighted sum of the velocities from the grid nodes. The weighted velocity is based on how close each grid node is to the particle; the closer the node, the more it contributes to the particle’s new velocity. This makes sense because we expect the grid areas closest to the particle to have the most influence on its movement. Specifically, the calculations of weights use the same method, as weights calculations in P2G, mentioned above. Here is the general formula:

$$v_p^{n+1} = \sum_i w_{ip} v_i^{n+1}$$

After we have the new velocity for each particle, we need to figure out how quickly the velocity is changing in different directions around each particle. This is where we calculate the particle velocity gradient. It gives us a picture of how the velocity varies from point to point in the material, which is important for understanding how the material is stretching or compressing. We compute the velocity gradient as a sum of the outer product of the weighted grid node velocities and the displacement of the grid nodes from the particle position, divided by the square of the grid spacing. Here is the formula:

$$C_p^{n+1} = \frac{a}{\Delta x^2} \sum_i w_{ip} \, v_i^{n+1} (x_i - x_p^n)^T$$

Mathematically, the velocity gradient C for a particle is a matrix and can be represented as:

$$C = \begin{bmatrix} \frac{\partial v_x}{\partial x} & \frac{\partial v_x}{\partial y} & \frac{\partial v_x}{\partial z} \\ \frac{\partial v_y}{\partial x} & \frac{\partial v_y}{\partial y} & \frac{\partial v_y}{\partial z} \\ \frac{\partial v_z}{\partial x} & \frac{\partial v_z}{\partial y} & \frac{\partial v_z}{\partial z} \end{bmatrix}$$

Each entry in this matrix is a partial derivative, and together they describe how the particle’s velocity changes in space. For instance, the term $\partial v_x / \partial x$ tells us how much the x-component of velocity changes as the particle moves in the x-direction, and similarly for the other terms. In MPM, this matrix is used to update the deformation gradient of each particle in P2G at the next time step.

The final step in the grid-to-particle phase is to update the position of the particles. Each particle moves according to its velocity, which we’ve just calculated. By multiplying the velocity by the time step, we get the distance the particle moves during that time step. Then we add this distance to the particle’s current position to get its new position. This step physically moves the particle through the simulation space and is crucial for tracking the flow of the material over time.

Summary

Each of these steps is interconnected, ensuring that the particles are updated in a consistent and accurate manner based on the information from the grid. This cycle of updates continues, with the grid providing the medium for calculations and the particles carrying the material properties, allowing us to simulate complex material behaviors.

Implementation

Core Algorithm

The core algorithm is implementation of MPM.

1. P2G

Firstly, we should do particle to grid process using Particle-in-Cell's Quadratic method. We store the weight in a weight list and calculate the grid momentum with weight. We also calculate the grid mass using weight for grid velocity calculation.

```
Xp = self.F_x[p] / self.dx
base = int(Xp - 0.5)
fx = Xp - base
w = [0.5 * (1.5 - fx) ** 2, 0.75 - (fx - 1) ** 2, 0.5 * (fx - 0.5) ** 2]
```

Secondly, we calculate the APIC's momentum and particle elastic impulse.

The deformation gradient uses this formula:

$$\mathbf{F}_p^{n+1} = (\mathbf{I} + \Delta t \mathbf{C}_p^n)\mathbf{F}_p^n$$

Code:

```
# Deformation gradient update moved to the front (elastic deformation) (hack!)
self.def_grad[p] = ((ti.Matrix.identity(float, n: 3) + dt * self.C[p]) @
                    self.def_grad[p])  # deformation gradient update
```

Since the material will become harder when compressed, we define the h:

```
# Hardening coefficient: material harder when compressed
h = ti.exp(10 * (1.0 - self.F_Jp[p]))  # Plastic change effect the h, or the h is always 1
```

In Material DreamWorks, we should have a parameter H to control the elasticity.

Code:

```
if is_elastic_object:  # elastic object, make it softer
    h = H  # 0.1 ~ 1.0
```

We should make the μ and λ bigger, in order to make the material harder:

```
mu, la = mu_0 * h, lambda_0 * h
```

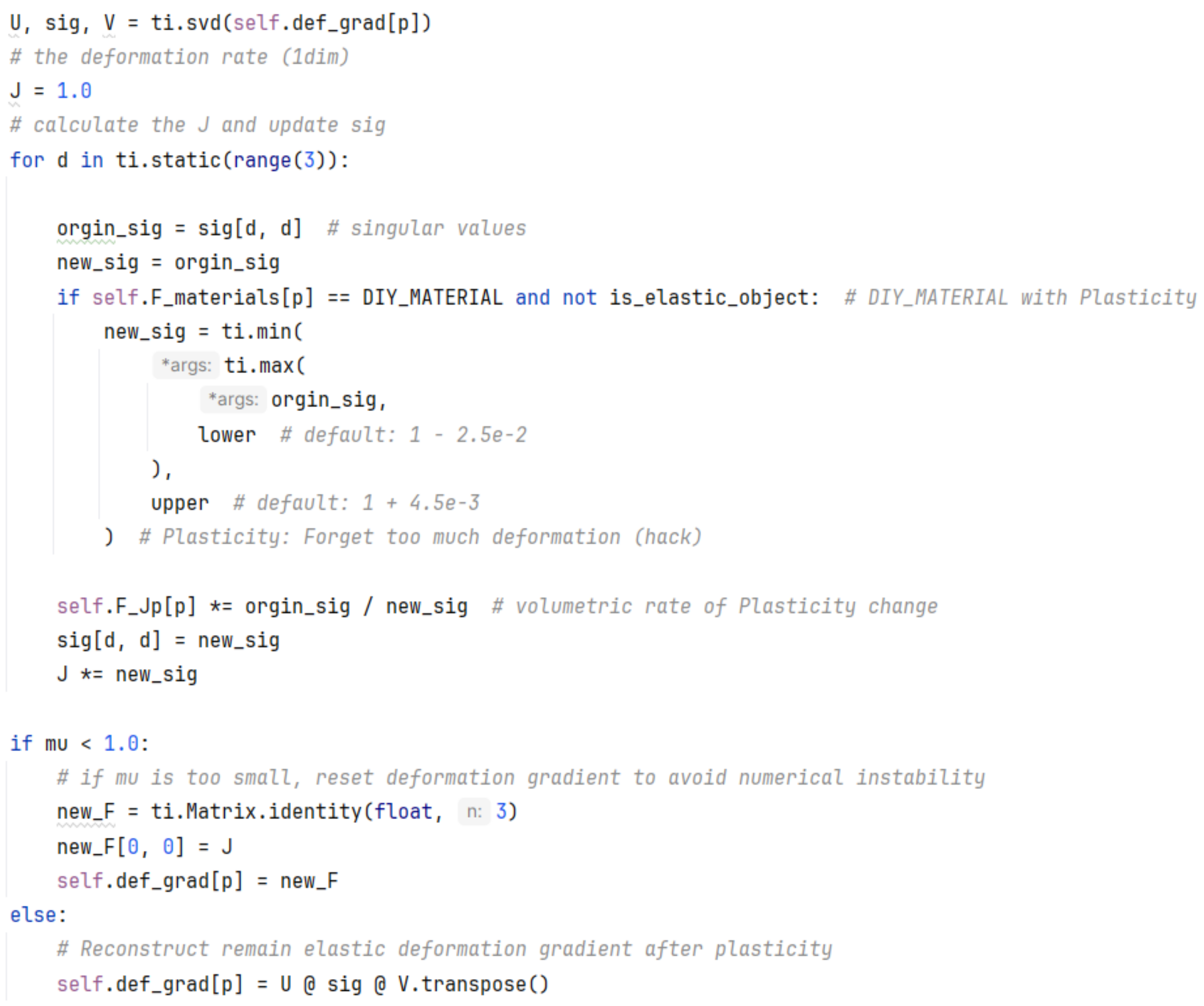
After we handle the plasticity, we reconstruct the deformation gradient. Then, we should calculate the P(F) of Corotated model and form the particle elastic impulse. At first, we calculate the "stress_part1 = P(F) * F.transposed()". Then, we create "stress = 4*Δt / Δx**2 * V * P(F) * F.transposed()". Finally, we add part of APIC's momentum with "stress" and multiply the distance to get the APIC's momentum and particle elastic impulse.

Code:

```
# Corotated used elastic F_dg
# stress_part1
# = P(F) * F.transposed()
# = (2μ(F - R) + λ(J - 1)J * F.transposed().inverse()) * F.transposed()
# = 2μ(F - R)* F.transposed() + λ(J - 1)J
stress_part1 = (
        2 * mu * (self.def_grad[p] - U @ V.transpose()) @ self.def_grad[p].transpose() +
        ti.Matrix.identity(
            float, n: 3
        ) * la * J * (J - 1)
)
# stress
# = 4*Δt / Δx**2 * V * P(F) * F.transposed()
# = 4*Δt / Δx**2 * V * stress_part1
stress = (-dt * self.p_vol * 4) * stress_part1 / self.dx ** 2
affine = self.p_mass * self.C[p] + stress

# 3d Grid momentum
for offset in ti.static(ti.grouped(ti.ndrange(*self.neighbour))):
    dpos = (offset - fx) * self.dx
    weight = 1.0
    for i in ti.static(range(self.dim)):
        weight *= w[offset[i]][i]
    self.F_grid_v[base + offset] += weight * (self.p_mass * self.F_v[p] + affine @ dpos)
    self.F_grid_m[base + offset] += weight * self.p_mass
```

Thirdly, we should handle the plasticity. We do SVD of deformation gradient, clamp the singular value between the adjustable lower boundary and upper boundary, and reconstruct the remain elastic deformation gradient after plasticity. It is very important that we reset deformation gradient to avoid numerical instability when the μ is too small.

Code:

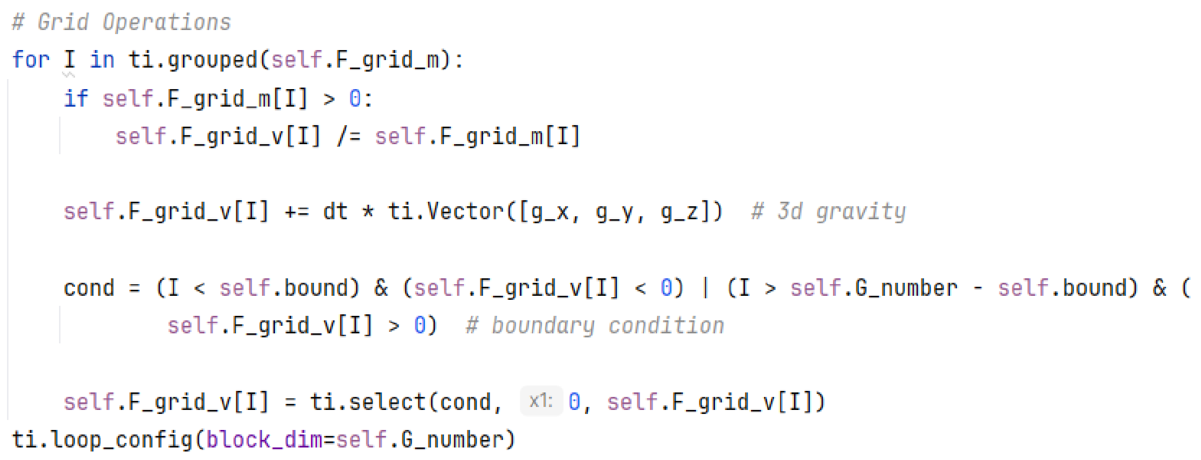


```
U, sig, V = ti.svd(self.def_grad[p])
# the deformation rate (1dim)
J = 1.0
# calculate the J and update sig
for d in ti.static(range(3)):

    orgin_sig = sig[d, d]  # singular values
    new_sig = orgin_sig
    if self.F_materials[p] == DIY_MATERIAL and not is_elastic_object:  # DIY_MATERIAL with Plasticity
        new_sig = ti.min(
            *args: ti.max(
                *args: orgin_sig,
                lower  # default: 1 - 2.5e-2
            ),
            upper  # default: 1 + 4.5e-3
        )  # Plasticity: Forget too much deformation (hack)

    self.F_Jp[p] *= orgin_sig / new_sig  # volumetric rate of Plasticity change
    sig[d, d] = new_sig
    J *= new_sig

if mu < 1.0:
    # if mu is too small, reset deformation gradient to avoid numerical instability
    new_F = ti.Matrix.identity(float,  n: 3)
    new_F[0, 0] = J
    self.def_grad[p] = new_F
else:
    # Reconstruct remain elastic deformation gradient after plasticity
    self.def_grad[p] = U @ sig @ V.transpose()
```

2. Grid Operations

In this part, we calculate the velocity of grid. Secondly, we add some influence of gravity. At the end, we check the boundary conditions to set the velocity to zero when the particle hit the boundary and its velocity is not zero.

Code:



```
# Grid Operations
for I in ti.grouped(self.F_grid_m):
    if self.F_grid_m[I] > 0:
        self.F_grid_v[I] /= self.F_grid_m[I]

    self.F_grid_v[I] += dt * ti.Vector([g_x, g_y, g_z])  # 3d gravity

    cond = (I < self.bound) & (self.F_grid_v[I] < 0) | (I > self.G_number - self.bound) & (
            self.F_grid_v[I] > 0)  # boundary condition

    self.F_grid_v[I] = ti.select(cond,  x1: 0, self.F_grid_v[I])
ti.loop_config(block_dim=self.G_number)
```

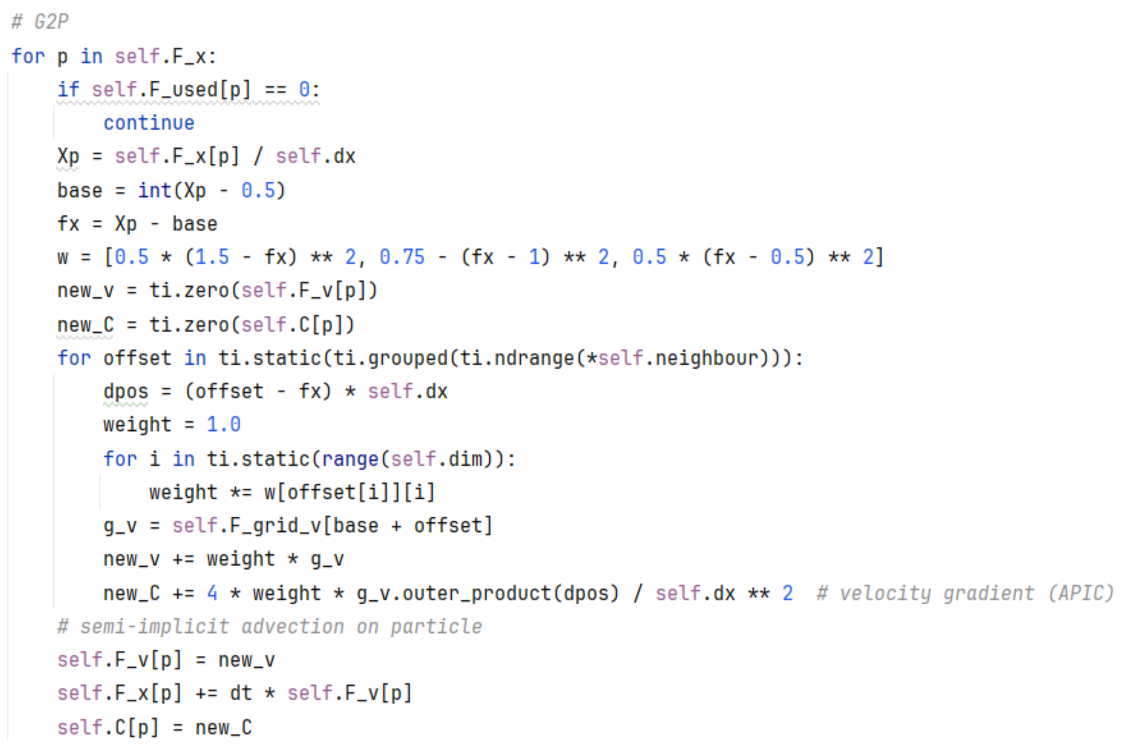
3. G2P

Firstly, we should gather the grid into particle reusing the Particle-in-Cell’s Quadratic method.

Secondly, after gathering the basic movement, we should calculate the APIC’s C matrix.

Finally, we perform the semi-implicit advection on particle to make the particle move to the next position.

Code:

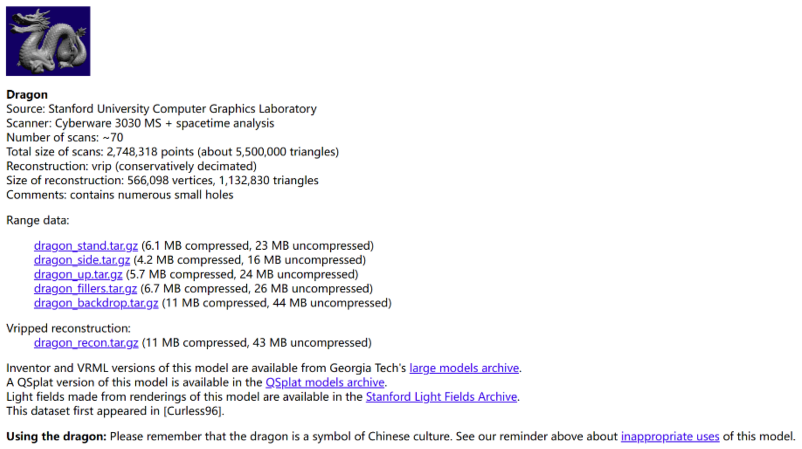


```
# G2P
for p in self.F_x:
    if self.F_used[p] == 0:
        continue
    Xp = self.F_x[p] / self.dx
    base = int(Xp - 0.5)
    fx = Xp - base
    w = [0.5 * (1.5 - fx) ** 2, 0.75 - (fx - 1) ** 2, 0.5 * (fx - 0.5) ** 2]
    new_v = ti.zero(self.F_v[p])
    new_C = ti.zero(self.C[p])
    for offset in ti.static(ti.grouped(ti.ndrange(*self.neighbour))):
        dpos = (offset - fx) * self.dx
        weight = 1.0
        for i in ti.static(range(self.dim)):
            weight *= w[offset[i]][i]
        g_v = self.F_grid_v[base + offset]
        new_v += weight * g_v
        new_C += 4 * weight * g_v.outer_product(dpos) / self.dx ** 2  # velocity gradient (APIC)
    # semi-implicit advection on particle
    self.F_v[p] = new_v
    self.F_x[p] += dt * self.F_v[p]
    self.C[p] = new_C
```

Dragon Splatting

Based on the core algorithm, now we can create some cool effects combined with our simulation solver and other 3D creation tools like Houdini. In Chinese culture, dragon is the ruler of water. And we want to make a simple demo representing the special relationship between dragon and water.

First, we downloaded the mesh model from Stanford 3D scanning repository. Then convert it to point cloud in Houdini.



Dragon
Source: Stanford University Computer Graphics Laboratory
Scanner: Cyberware 3030 MS + spacetime analysis
Number of scans: ~70
Total size of scans: 2,748,318 points (about 5,500,000 triangles)
Reconstruction: vrip (conservatively decimated)
Size of reconstruction: 566,098 vertices, 1,132,830 triangles
Comments: contains numerous small holes

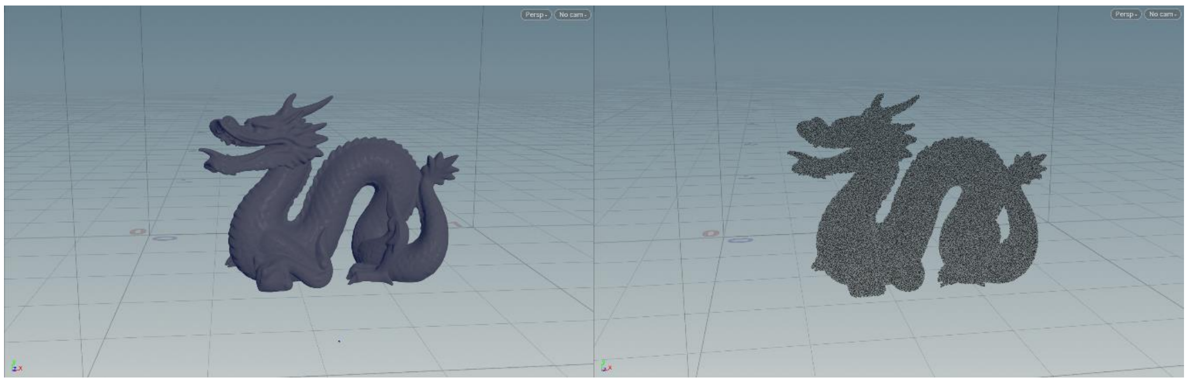
Range data:

dragon_stand.tar.gz (6.1 MB compressed, 23 MB uncompressed)
dragon_side.tar.gz (4.2 MB compressed, 16 MB uncompressed)
dragon_up.tar.gz (5.7 MB compressed, 24 MB uncompressed)
dragon_fillers.tar.gz (6.7 MB compressed, 26 MB uncompressed)
dragon_backdrop.tar.gz (11 MB compressed, 44 MB uncompressed)

Vripped reconstruction:
dragon_recon.tar.gz (11 MB compressed, 43 MB uncompressed)

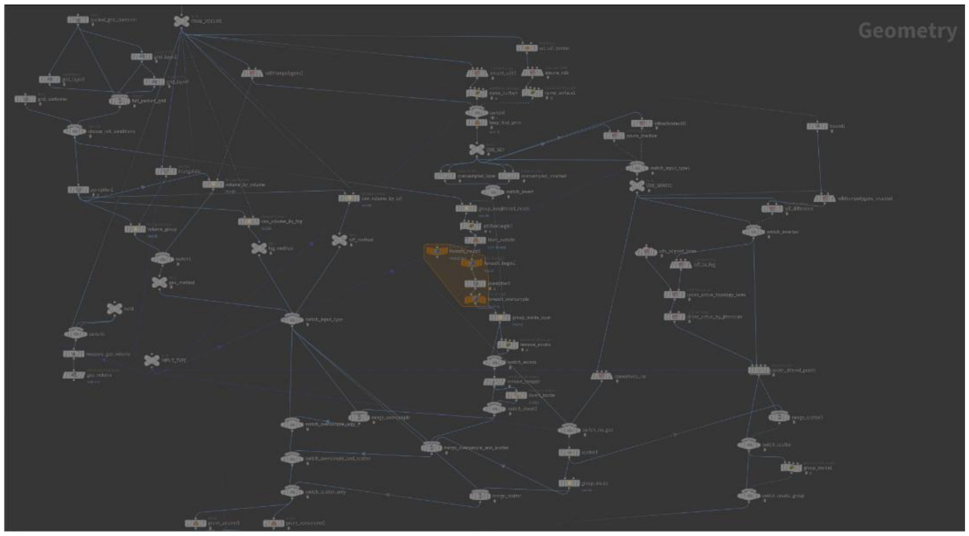
Inventor and VRML versions of this model are available from Georgia Tech's large models archive.
A QSplat version of this model is available in the QSplat models archive.
Light fields made from renderings of this model are available in the Stanford Light Fields Archive.
This dataset first appeared in [Curless96].

Using the dragon: Please remember that the dragon is a symbol of Chinese culture. See our reminder above about inappropriate uses of this model.



We made tradeoff for the density of point cloud to achieve efficient computation as well as a good look for the final rendering. Finally, we set the separation distance between each pair of points to 0.003.

The processing operation can be done by visualized programming thanks to all kinds of nodes provided by Houdini, like blueprints in Unreal Engine. Mesh to point cloud operation can be done by the below nodes.

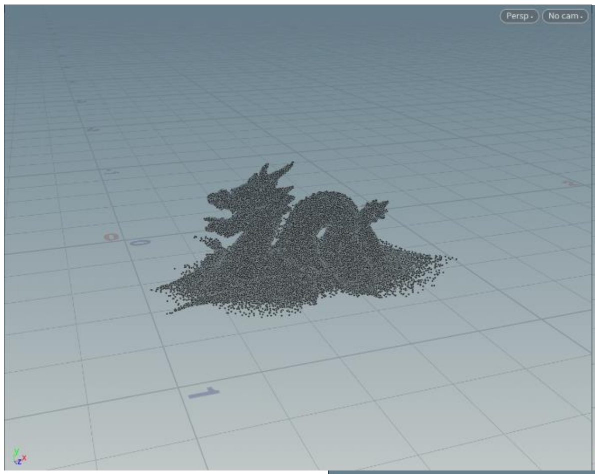
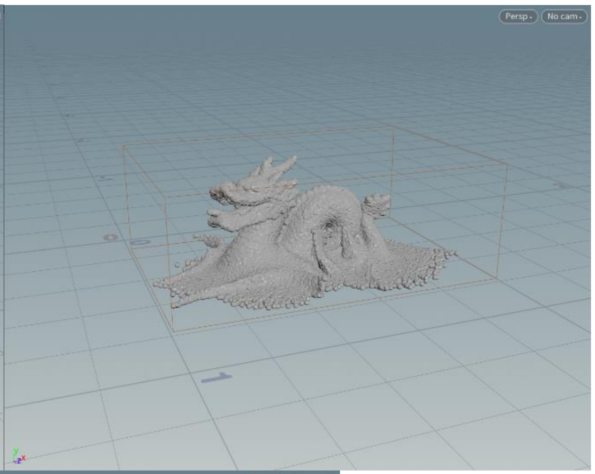
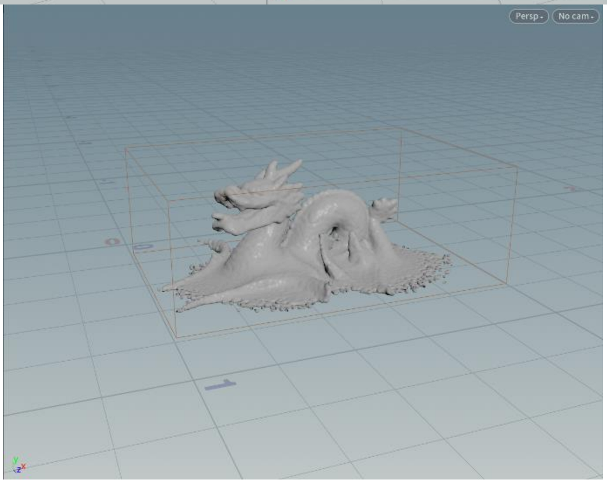


Then we could load the model data from the exported .ply file using the below codes. We just use the position attribute to initialize a bunch of particles, assigning their materials, velocities, elastic deformation variables and so on. We fine-tuned the parameters and solved the simulation mainly using the core algorithm as illustrated in the former section. We stored the solved result to a .ply file for 120 frames, which is for rendering a 5-second 24fps video. We then loaded the solved .ply files to Houdini.

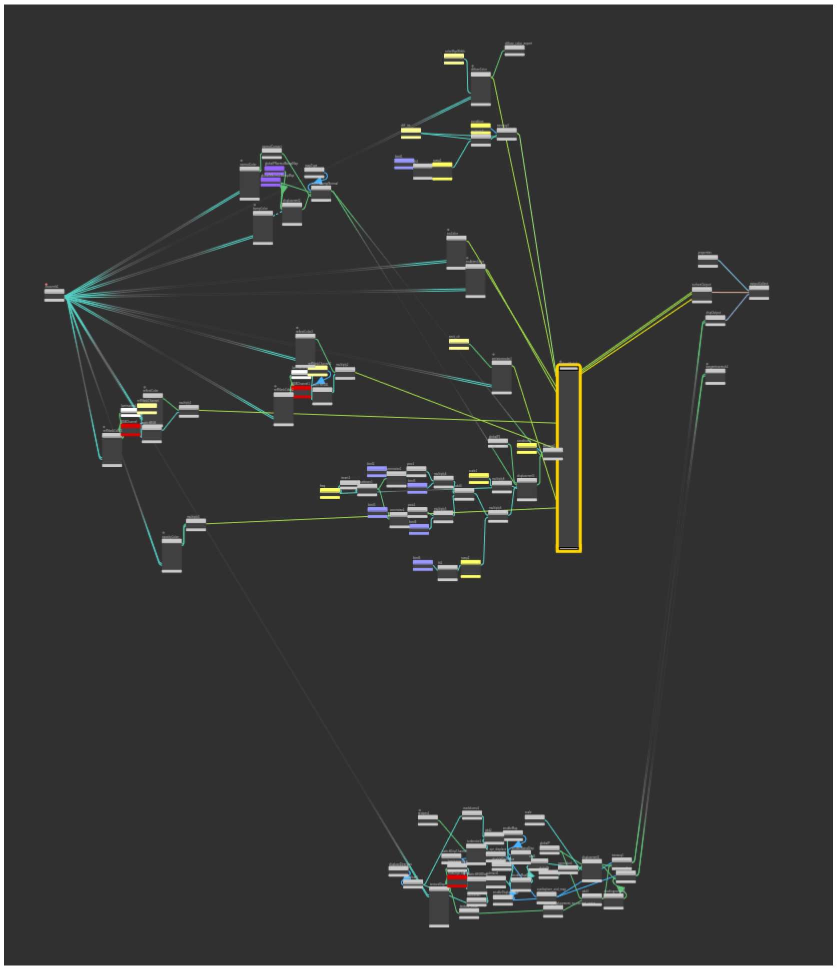
```
class PlyImporter:
    def __init__(self, file):
        self.file_name = file
        plydata = PlyData.read(self.file_name)
        data = plydata['vertex'].data
        self.count = plydata['vertex'].count
        self.np_array = np.array([[x, y, z] for x, y, z in data])

    def get_array(self):
        return self.np_array

    def get_count(self):
        return self.count

    def multiply(self, mul):
        self.np_array *= mul
```

```
def save_ply(frame1):
    series_prefix = "dragon_solved.ply"
    num_vertices = n_particles
    np_pos = np.reshape(x.to_numpy(), (num_vertices, 3))
    writer = ti.tools.np2ply.PLYWriter(num_vertices=num_vertices)
    writer.add_vertex_pos(np_pos[:, 0], np_pos[:, 1], np_pos[:, 2])
    writer.export_frame_ascii(frame1+1, series_prefix)
```

We tried many methods to convert the point clouds to mesh and make the meshes look smooth and natural. Here are some examples from the workflow.

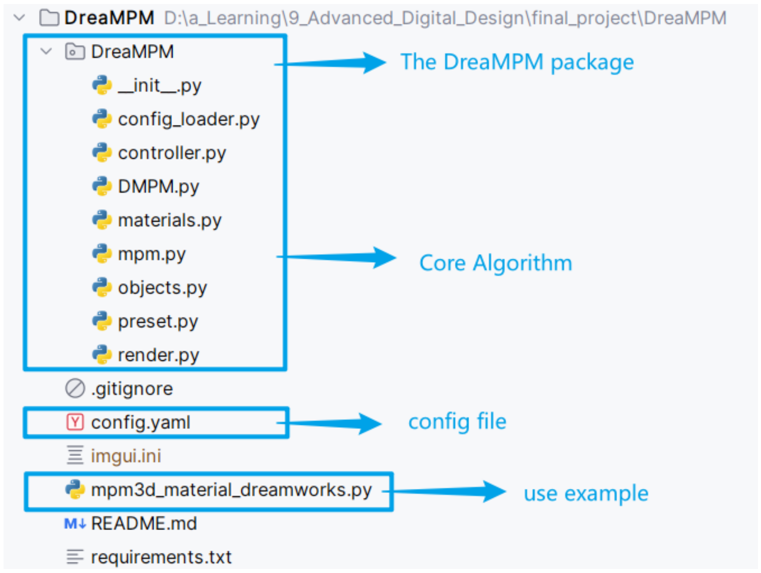




Last but not least, we picked appropriate materials to make the rendering output like natural water, though it’s still far from perfect. We gained inspiration from the course, lighting model and voxel rendering. We add uniform volume for water body——namely the part below the surface——to enable the thicker water from the viewing angle to behave more blue and the strategy works. It took an average of 7-8 minutes to render one frame. The Houdini source projects and the video are attached in our submission.

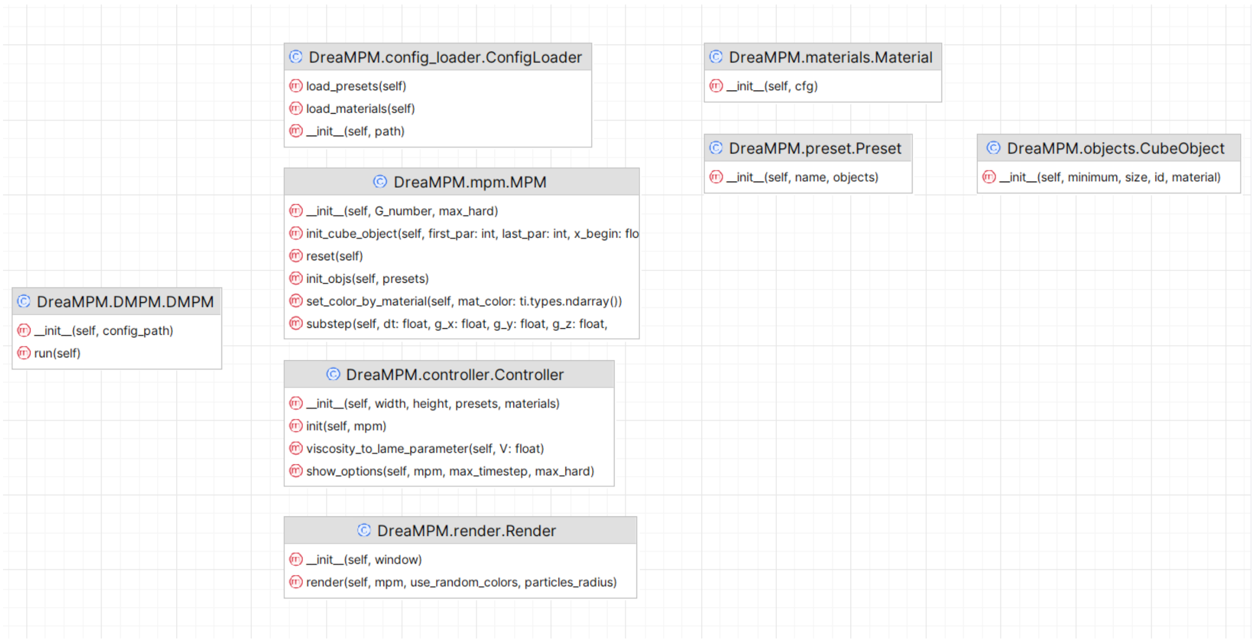


Material DreamWorks – DreaMPM

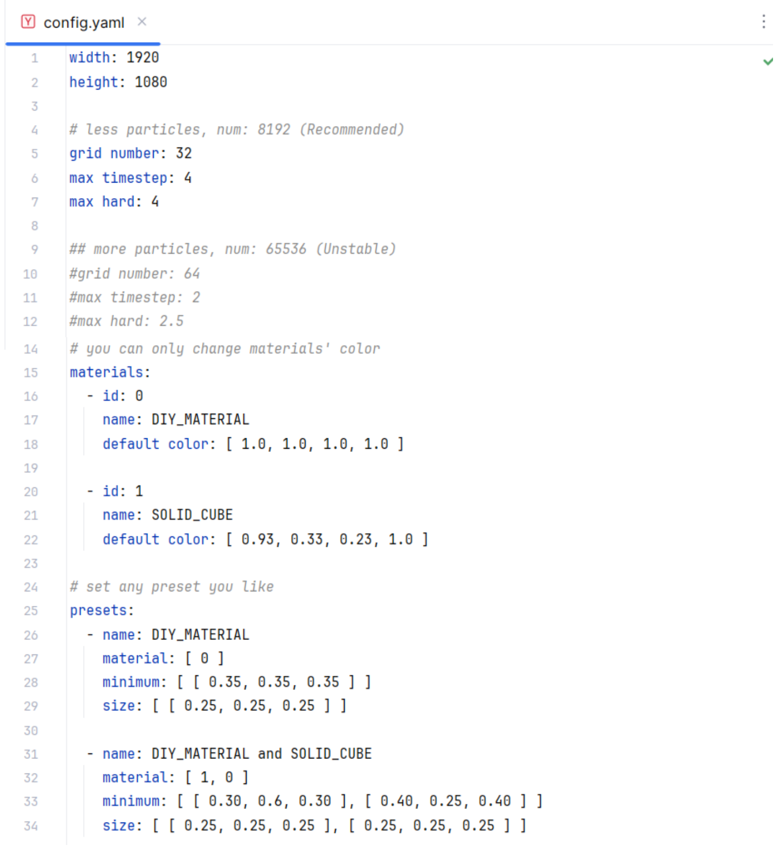
DreaMPM is a 3D real-time material DreamWorks based on MPM (Material Point Method) that we created using the Taichi language. The entire DreaMPM program can be viewed here. The project directory of DreaMPM is shown below.


```
DreaMPM D:\a_Learning\9_Advanced_Digital_Design\final_project\DreaMPM
  DreaMPM
    __init__.py
    config_loader.py
    controller.py
    DMPM.py
    materials.py
    mpm.py
    objects.py
    preset.py
    render.py
  .gitignore
  config.yaml
  imgui.ini
  mpm3d_material_dreamworks.py
  README.md
  requirements.txt
```

We encapsulated the core algorithm as the MPM class and created ConfigLoader, Controller and Render to help with configuration file loading, material property control and rendering. The entire application is encapsulated in a DMPM class. The following diagram shows the internal structure of the DreaMPM package.

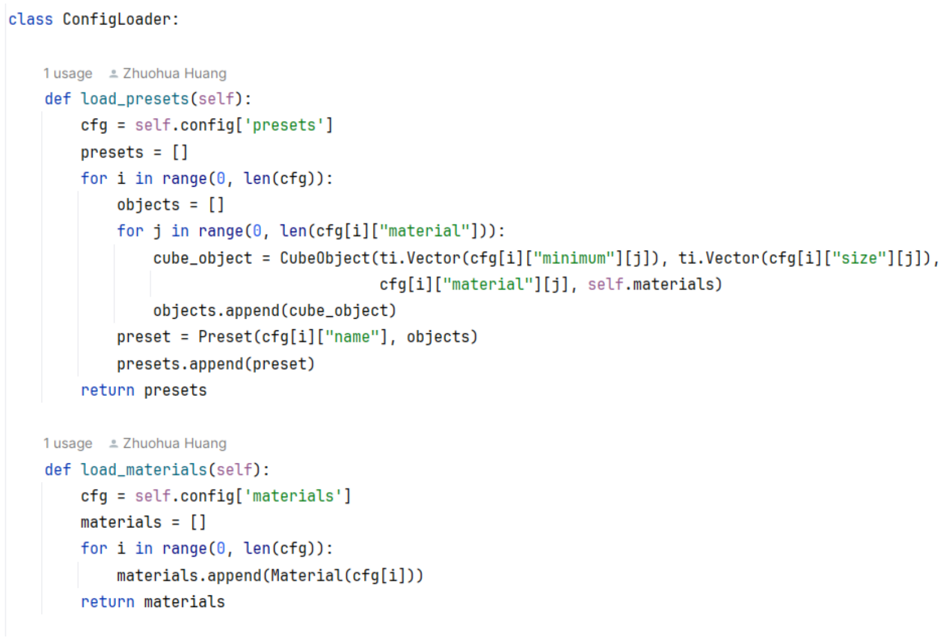

```
DreaMPM.config_loader.ConfigLoader
  load_presets(self)
  load_materials(self)
  __init__(self, path)

DreaMPM.materials.Material
  __init__(self, cfg)

DreaMPM.preset.Preset
  __init__(self, name, objects)

DreaMPM.objects.CubeObject
  __init__(self, minimum, size, id, material)

DreaMPM.mpm.MPM
  __init__(self, G_number, max_hard)
  init_cube_object(self, first_par: int, last_par: int, x_begin: flo
  reset(self)
  init_objs(self, presets)
  set_color_by_material(self, mat_color: ti.types.ndarray())
  substep(self, dt: float, g_x: float, g_y: float, g_z: float,

DreaMPM.DMPM.DMPM
  __init__(self, config_path)
  run(self)

DreaMPM.controller.Controller
  __init__(self, width, height, presets, materials)
  init(self, mpm)
  viscosity_to_lame_parameter(self, V: float)
  show_options(self, mpm, max_timestep, max_hard)

DreaMPM.render.Render
  __init__(self, window)
  render(self, mpm, use_random_colors, particles_radius)
```

The “config.yaml” file is where you can make DreaMPM parameter configurations. In the configuration file you can customize the initial window size, different particle parameters, the initial properties of the materials and different preset scenes. The following figure shows the “config.yaml” file.


config.yaml

```yaml
width: 1920
height: 1080

# less particles, num: 8192 (Recommended)
grid number: 32
max timestep: 4
max hard: 4

## more particles, num: 65536 (Unstable)
#grid number: 64
#max timestep: 2
#max hard: 2.5
# you can only change materials' color
materials:
  - id: 0
    name: DIY_MATERIAL
    default color: [ 1.0, 1.0, 1.0, 1.0 ]

  - id: 1
    name: SOLID_CUBE
    default color: [ 0.93, 0.33, 0.23, 1.0 ]

# set any preset you like
presets:
  - name: DIY_MATERIAL
    material: [ 0 ]
    minimum: [ [ 0.35, 0.35, 0.35 ] ]
    size: [ [ 0.25, 0.25, 0.25 ] ]

  - name: DIY_MATERIAL and SOLID_CUBE
    material: [ 1, 0 ]
    minimum: [ [ 0.30, 0.6, 0.30 ], [ 0.40, 0.25, 0.40 ] ]
    size: [ [ 0.25, 0.25, 0.25 ], [ 0.25, 0.25, 0.25 ] ]
```

The "ConfigLoader.py" load the configuration into instance of Material class, CubeObject class and Preset class. After loading, the ConfigLoader contains the hyperparameter, two types of material and several preset scenes.



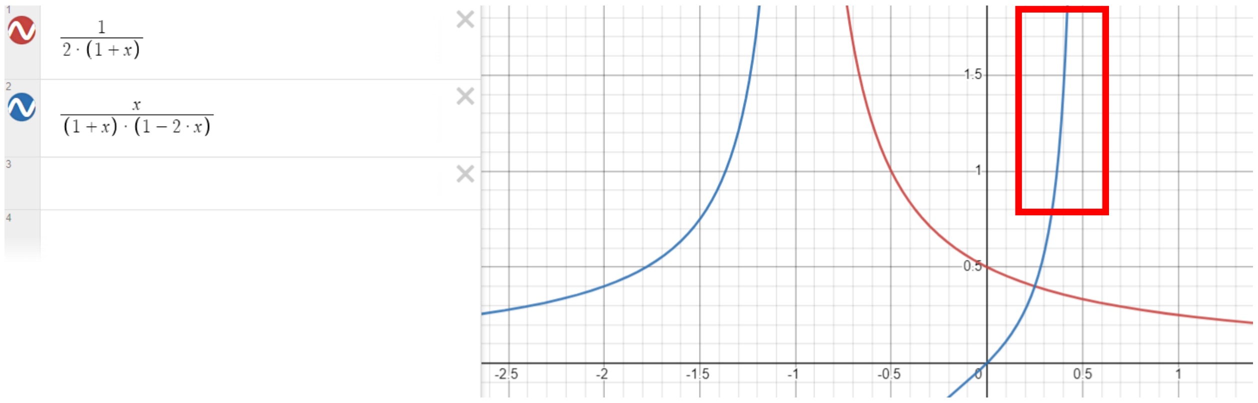
```python
class ConfigLoader:

    1 usage  Zhuohua Huang
    def load_presets(self):
        cfg = self.config['presets']
        presets = []
        for i in range(0, len(cfg)):
            objects = []
            for j in range(0, len(cfg[i]["material"])):
                cube_object = CubeObject(ti.Vector(cfg[i]["minimum"][j]), ti.Vector(cfg[i]["size"][j]),
                                         cfg[i]["material"][j], self.materials)
                objects.append(cube_object)
            preset = Preset(cfg[i]["name"], objects)
            presets.append(preset)
        return presets

    1 usage  Zhuohua Huang
    def load_materials(self):
        cfg = self.config['materials']
        materials = []
        for i in range(0, len(cfg)):
            materials.append(Material(cfg[i]))
        return materials
```

In order to control the nature of DIY material, we should change the Lamé Parameters. Typically, the Lamé Parameters can be controlled using Young's modulus and Poisson's ratio. The following formulas are the control method.

$$\lambda = E / (2 * (1 + nu))$$

$$\mu = E * nu / ((1 + nu) * (1 - 2 * nu))$$

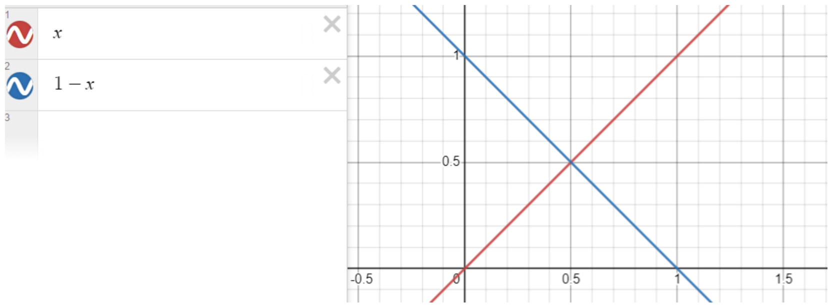
However, this method is easy to explode. We can find the problem in the function image below.



After going through several tests, we propose approximate equations. We control Lamé Parameters using one parameter V ranged [0, 1]. Let E denotes the Young's modulus, we have:

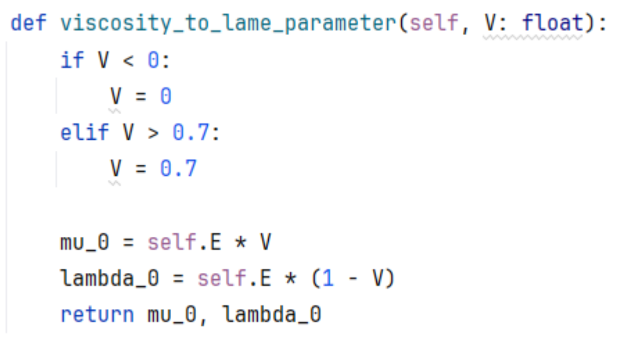
$$\lambda = E * V$$
$$\mu = E * (1 - V)$$

The low value of V means Viscosity and the high value of V means Stiffness. The following figure shows the function image of these equations.



Both Lamé Parameters λ and μ can become 0, when the V is 0 and 1. The maximum value of λ and μ can be any number, when we change the Young's modulus E.

The implementation of our proposed method is shown below.



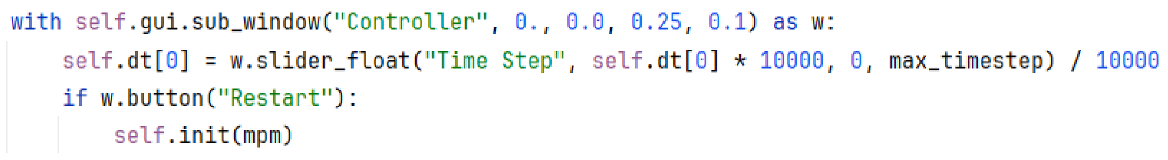
```python
def viscosity_to_lame_parameter(self, V: float):
    if V < 0:
        V = 0
    elif V > 0.7:
        V = 0.7

    mu_0 = self.E * V
    lambda_0 = self.E * (1 - V)
    return mu_0, lambda_0
```

The “Controller” class utilizes the taichi.ui and provides 5 control bars for user to customize the scene:

1. Controller:

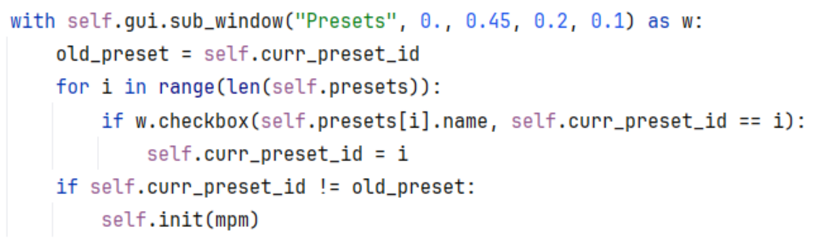
    a) Time Step: This value can make the time slow down or speed up. If the time step is 0, the scene will pause.

    b) Restart: This button can restart the scene.



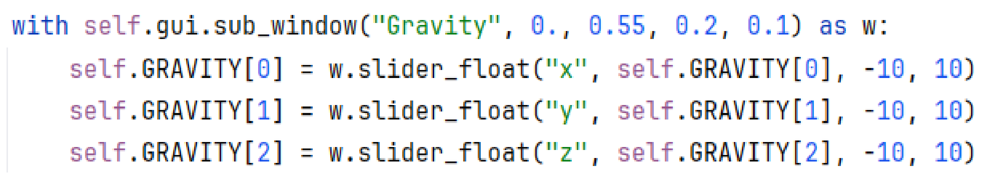
```python
with self.gui.sub_window("Controller", 0., 0.0, 0.25, 0.1) as w:
    self.dt[0] = w.slider_float("Time Step", self.dt[0] * 10000, 0, max_timestep) / 10000
    if w.button("Restart"):
        self.init(mpm)
```

2. Presets: We give 2 preset scenes. The first one is the DIY material cube. The second one is the DIY material cube interacts with a solid cube.



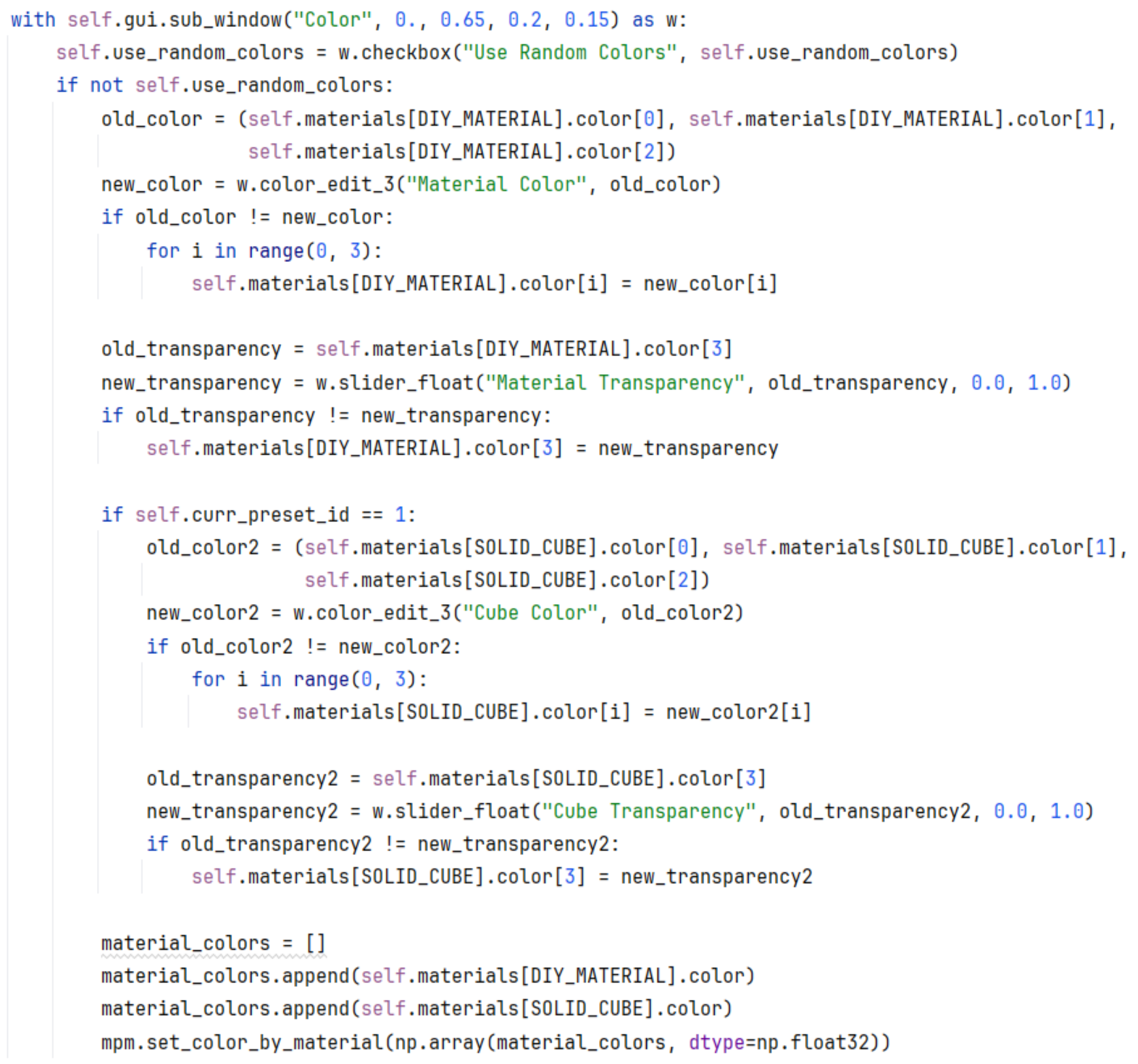
```python
with self.gui.sub_window("Presets", 0., 0.45, 0.2, 0.1) as w:
    old_preset = self.curr_preset_id
    for i in range(len(self.presets)):
        if w.checkbox(self.presets[i].name, self.curr_preset_id == i):
            self.curr_preset_id = i
    if self.curr_preset_id != old_preset:
        self.init(mpm)
```

3. Gravity: The gravity can be controlled by changing its x, y and z values.



```python
with self.gui.sub_window("Gravity", 0., 0.55, 0.2, 0.1) as w:
    self.GRAVITY[0] = w.slider_float("x", self.GRAVITY[0], -10, 10)
    self.GRAVITY[1] = w.slider_float("y", self.GRAVITY[1], -10, 10)
    self.GRAVITY[2] = w.slider_float("z", self.GRAVITY[2], -10, 10)
```

4. Color: The Color can change the color of your DIY material also the solid cube. The transparency can also be changed.

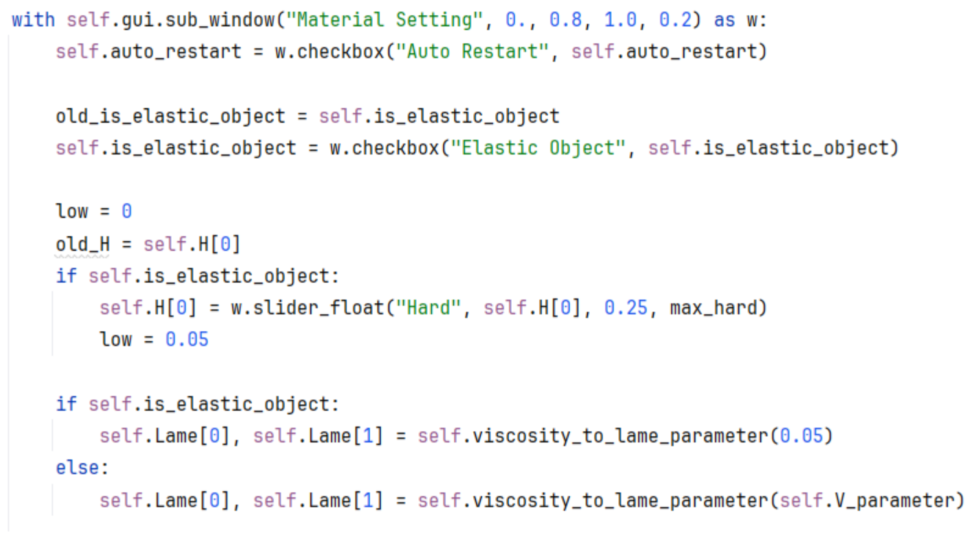


```python
with self.gui.sub_window("Color", 0., 0.65, 0.2, 0.15) as w:
    self.use_random_colors = w.checkbox("Use Random Colors", self.use_random_colors)
    if not self.use_random_colors:
        old_color = (self.materials[DIY_MATERIAL].color[0], self.materials[DIY_MATERIAL].color[1],
                     self.materials[DIY_MATERIAL].color[2])
        new_color = w.color_edit_3("Material Color", old_color)
        if old_color != new_color:
            for i in range(0, 3):
                self.materials[DIY_MATERIAL].color[i] = new_color[i]

        old_transparency = self.materials[DIY_MATERIAL].color[3]
        new_transparency = w.slider_float("Material Transparency", old_transparency, 0.0, 1.0)
        if old_transparency != new_transparency:
            self.materials[DIY_MATERIAL].color[3] = new_transparency

        if self.curr_preset_id == 1:
            old_color2 = (self.materials[SOLID_CUBE].color[0], self.materials[SOLID_CUBE].color[1],
                          self.materials[SOLID_CUBE].color[2])
            new_color2 = w.color_edit_3("Cube Color", old_color2)
            if old_color2 != new_color2:
                for i in range(0, 3):
                    self.materials[SOLID_CUBE].color[i] = new_color2[i]

            old_transparency2 = self.materials[SOLID_CUBE].color[3]
            new_transparency2 = w.slider_float("Cube Transparency", old_transparency2, 0.0, 1.0)
            if old_transparency2 != new_transparency2:
                self.materials[SOLID_CUBE].color[3] = new_transparency2

        material_colors = []
        material_colors.append(self.materials[DIY_MATERIAL].color)
        material_colors.append(self.materials[SOLID_CUBE].color)
        mpm.set_color_by_material(np.array(material_colors, dtype=np.float32))
```

5. Material Setting

a) Elastic Object and Hard

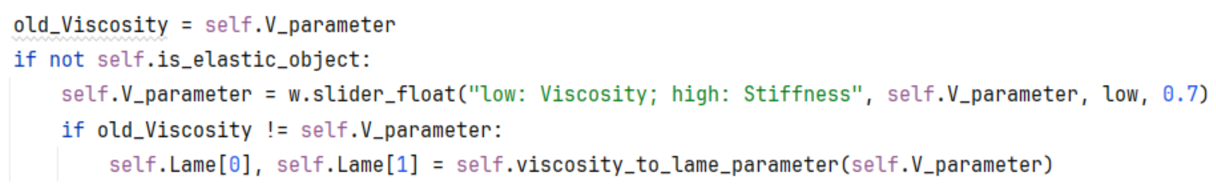
We can select Elastic Object, then the “Hard” value can control the hardness of object. In this situation, the DIY material doesn’t have any plasticity.



```python
with self.gui.sub_window("Material Setting", 0., 0.8, 1.0, 0.2) as w:
    self.auto_restart = w.checkbox("Auto Restart", self.auto_restart)

    old_is_elastic_object = self.is_elastic_object
    self.is_elastic_object = w.checkbox("Elastic Object", self.is_elastic_object)

    low = 0
    old_H = self.H[0]
    if self.is_elastic_object:
        self.H[0] = w.slider_float("Hard", self.H[0], 0.25, max_hard)
        low = 0.05

    if self.is_elastic_object:
        self.Lame[0], self.Lame[1] = self.viscosity_to_lame_parameter(0.05)
    else:
        self.Lame[0], self.Lame[1] = self.viscosity_to_lame_parameter(self.V_parameter)
```

b) V parameter

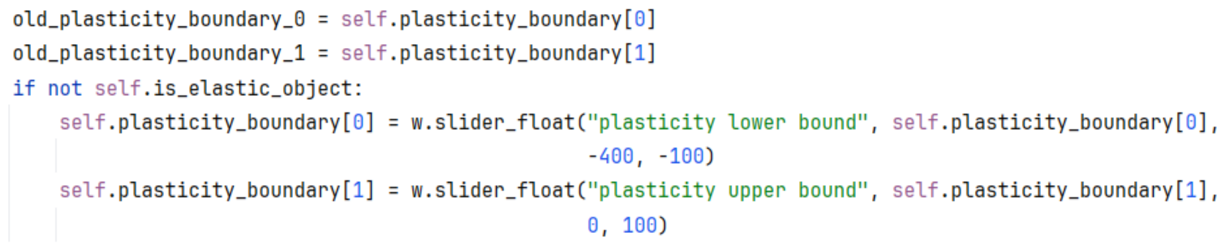
The V parameter will be passed to "viscosity_to_lame_parameter" function and get the Lamé Parameters.



```python
old_Viscosity = self.V_parameter
if not self.is_elastic_object:
    self.V_parameter = w.slider_float("low: Viscosity; high: Stiffness", self.V_parameter, low, 0.7)
    if old_Viscosity != self.V_parameter:
        self.Lame[0], self.Lame[1] = self.viscosity_to_lame_parameter(self.V_parameter)
```

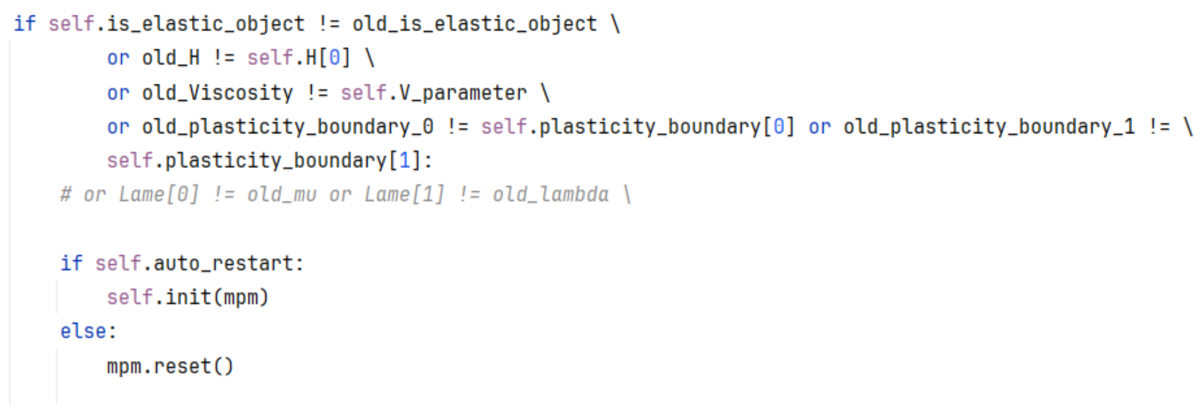
c) Plastic boundary

The Plastic boundary contains the lower boundary and upper boundary. These values will affect the plasticity of the material.



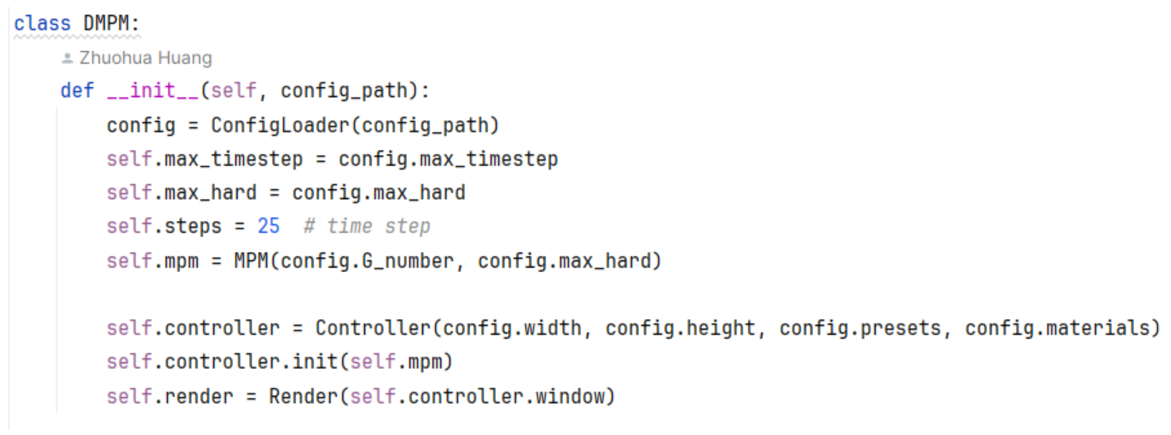
```python
old_plasticity_boundary_0 = self.plasticity_boundary[0]
old_plasticity_boundary_1 = self.plasticity_boundary[1]
if not self.is_elastic_object:
    self.plasticity_boundary[0] = w.slider_float("plasticity lower bound", self.plasticity_boundary[0],
                                                 -400, -100)
    self.plasticity_boundary[1] = w.slider_float("plasticity upper bound", self.plasticity_boundary[1],
                                                 0, 100)
```

Finally, the whole scene will initialize or reset if any parameter in the control bar has changed.



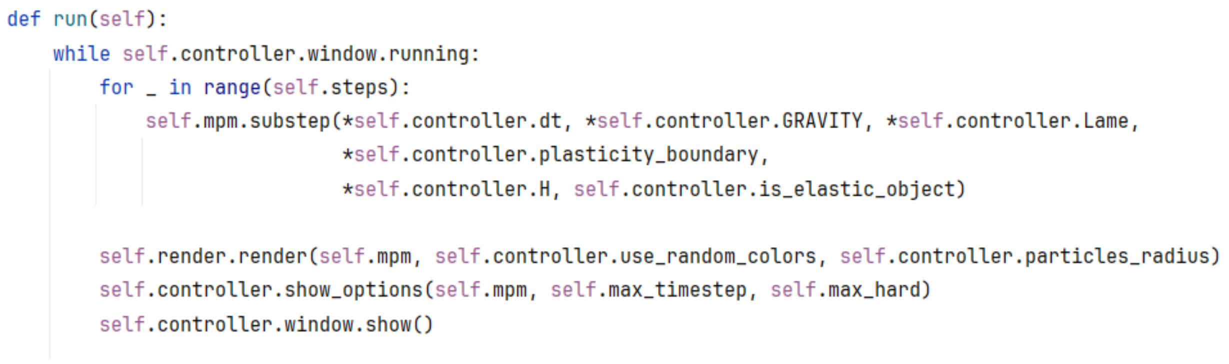
```
if self.is_elastic_object != old_is_elastic_object \
        or old_H != self.H[0] \
        or old_Viscosity != self.V_parameter \
        or old_plasticity_boundary_0 != self.plasticity_boundary[0] or old_plasticity_boundary_1 != \
        self.plasticity_boundary[1]:
    # or Lame[0] != old_mu or Lame[1] != old_lambda \

    if self.auto_restart:
        self.init(mpm)
    else:
        mpm.reset()
```

The DMPM class has the entire process. The constructor of DMPM will call ConfigLoader to initialize the hyperparameter and create MPM instance, Controller instance, and Render instance.



```
class DMPM:
    Zhuohua Huang
    def __init__(self, config_path):
        config = ConfigLoader(config_path)
        self.max_timestep = config.max_timestep
        self.max_hard = config.max_hard
        self.steps = 25  # time step
        self.mpm = MPM(config.G_number, config.max_hard)

        self.controller = Controller(config.width, config.height, config.presets, config.materials)
        self.controller.init(self.mpm)
        self.render = Render(self.controller.window)
```

In the run() function, for each step, we pass the parameters in the Controller to the core algorithm MPM.substep and get the new position of particle. Finally, we call Render.render() to render all the particle.



```
def run(self):
    while self.controller.window.running:
        for _ in range(self.steps):
            self.mpm.substep(*self.controller.dt, *self.controller.GRAVITY, *self.controller.Lame,
                             *self.controller.plasticity_boundary,
                             *self.controller.H, self.controller.is_elastic_object)

        self.render.render(self.mpm, self.controller.use_random_colors, self.controller.particles_radius)
        self.controller.show_options(self.mpm, self.max_timestep, self.max_hard)
        self.controller.window.show()
```

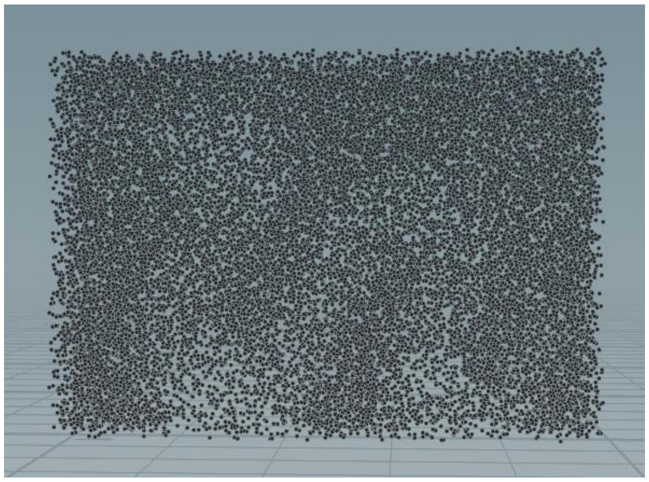
Drifting Boxes

In this demonstration, we employ a clever technique to create a waterfall effect. Initially, when the simulation begins, water particles are all generated but are strategically positioned outside the camera's view. This water particles generation trick provides that there is no need for additional space allocation during the simulation, enhancing the efficiency of the simulation. As time progresses, these water particles are gradually moved back into the target area. This technique not only simulates the appearance of a visually compelling waterfall from an unseen source but also ensures computational efficiency by avoiding the need for real-time particle generation.

To run the demo, use command "python fluidFlow.py"

Flow in the Dragon

As enlightened by the instructor, we decided to fuse our work to create a scene that injects water into a container with the shape of a dragon. A point cloud is natively filled in the interior part, thus it's difficult to create such a container. We simplified the problem by inversing the volume of the bounding box and created a container with a hollow part in the shape of a dragon. Then we removed the top part to allow water injection. You may barely see a dragon in the point cloud in the below image.



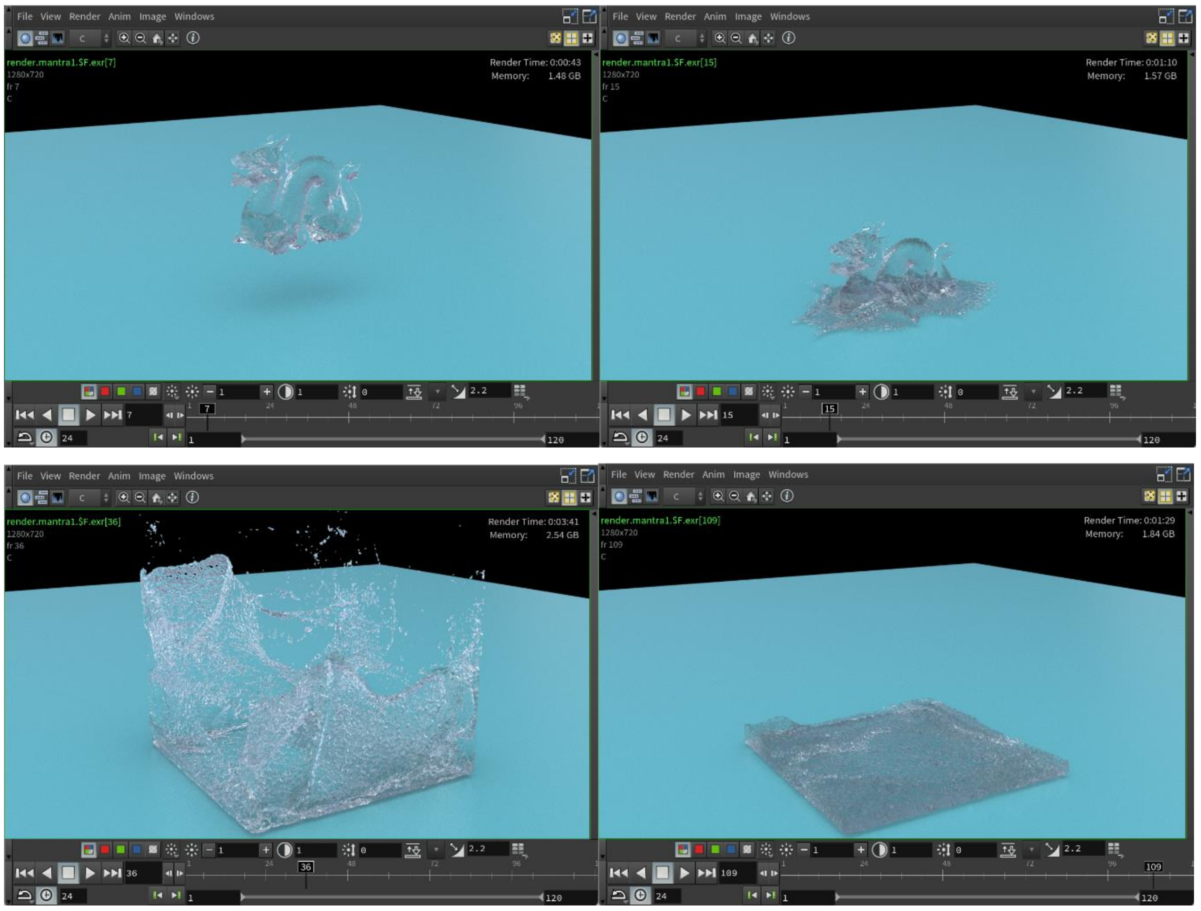
With the experience gained from the former experiments, we created a new material for the container that will hardly deform. Then we added water flow using the trick similar to "Drifting boxes". The water will gradually fill in the hollow part of the container as expected. However, we noticed that the hollow part will not be perfectly filled. Possible reasons include the container shape being rather irregular and the viscosity between different materials being somehow hard to control.

Results

In this Section we provide some visual results to help illustrate our work.

Dragon Splatting

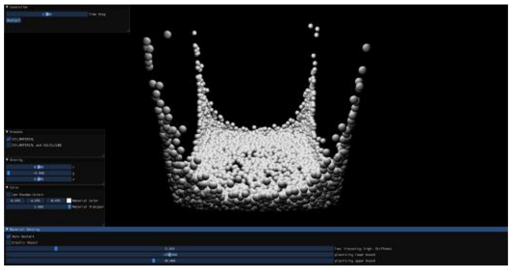
You may check any intermediate results directly in our Houdini project. Here are several representative renderings.

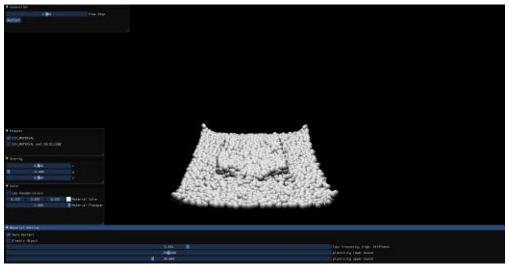


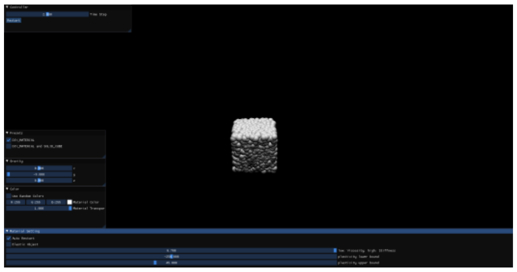
Material DreamWorks – DreaMPM

The DreaMPM successfully show 5 control bar. We can change the material's color and transparency. When we change the V parameter, the material can have different nature.

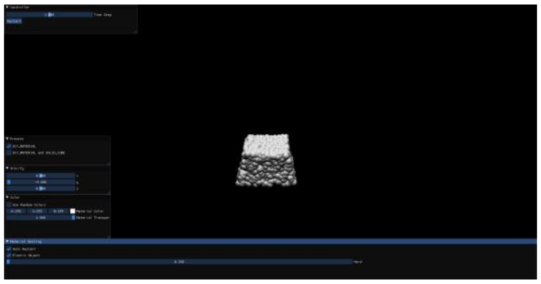
In the preset scene 1 “DIY_MATERIAL”, the DIY material can be a elastoplastic object that can range from liquids like water to sticky liquids like honey to solid objects as hard as stone.


Liquid

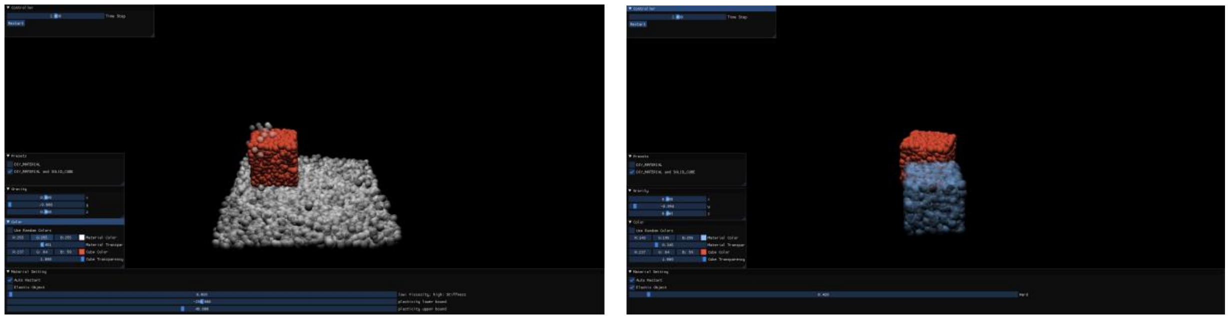

Sticky Liquid


Solid Object

When click the “Elastic Object”, the DIY material can become a flicking elastic object.

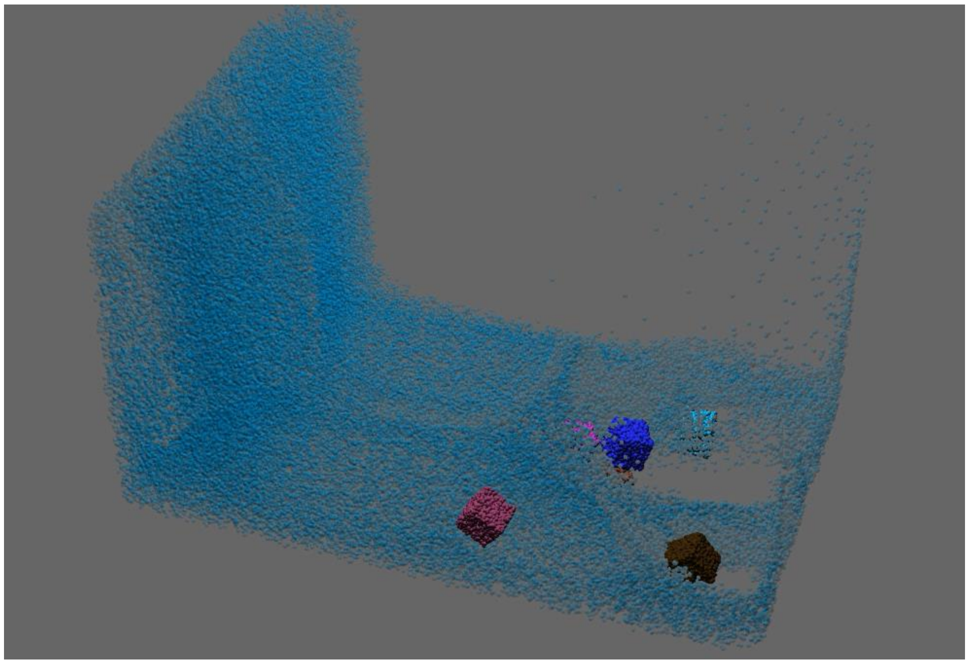

Elastic Object

In the preset scene 2 “DIY_MATERIAL and SOLID_CUBE”, we have a red cube that can interact with your DIY material.


Scene 2

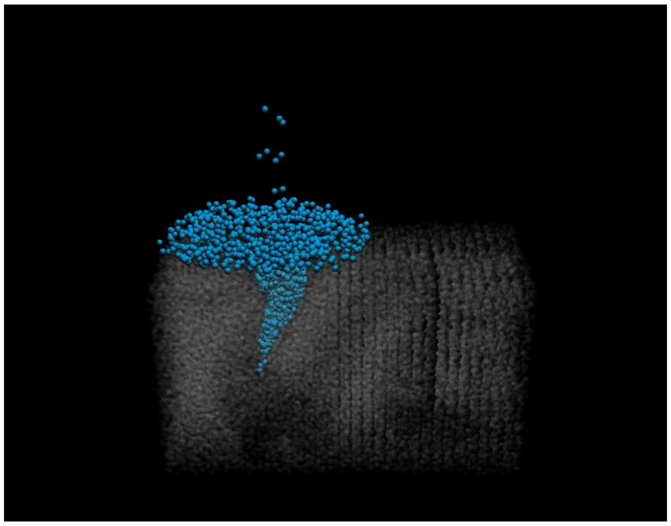
Drifting Boxes

Simulating some boxes are flushed away by a generated fluid flow.



Flow in the dragon

It’s an experimental attempt and the results are not visually interesting.



Discussion

In this comprehensive project, we explored MPM-based 3D Material Simulation, with a primary focus on liquid behavior, combining theoretical principles with practical implementation. Our journey through this project entailed a deep dive into the Material Point Method (MPM), its core algorithms, and its application in simulating various complex materials and phenomena. Our simulations, ranging from simple liquid movements to complex interactions with solid objects, demonstrate the versatility of the MPM technique in physics simulations.

The synthesis of Lagrangian and Eulerian views in MPM allowed us to capture the dynamic simulation of liquids and other materials with remarkable visual effects and efficiency. We demonstrated this through various applications, such as Drifting Boxes, Material DreamWorks and Dragon Splatting, which not only demonstrated the technical aspects of MPM but also its artistic and creative potential.

In conclusion, we consider how simulations based on the Material Point Method (MPM) can be useful in various fields, including both entertainment and scientific research. The versatility and robustness of MPM make it an invaluable tool in the evolution of digital design and simulation. Future work may focus on optimizing these methods further, exploring valuable real-time applications, and extending the technique to more complex scenarios. This project demonstrates an example of how computer simulation and digital art work together to expand the limits of what can be achieved in 3D simulation.

Member contributions

Member	Contribution	Ratio
SHAO Guocheng	Completed the MPM solver. Completed the workflow for I/O of solved results. Implemented “Dragon Splatting” and “Flow in the dragon”.	33.3%
HUANG Zhuohua	Completed the MPM solver. Proposed material property control function. Implemented Material DreamWorks – DreaMPM.	33.3%
SHEN Hengshuo	Completed the MPM solver. Proposed the idea for generating consecutive water flow and implemented “Drifting Boxes”	33.3%

References

1. Harlow, F.H. (1964) The Particle-in-Cell Computing Method for Fluid Dynamics. Methods in Computational Physics, 3, 319-343.

2. Chenfanfu Jiang, Craig Schroeder, Andrew Selle, Joseph Teran, and Alexey Stomakhin. 2015. The affine particle-in-cell method. ACM Trans. Graph. 34, 4, Article 51 (August 2015), 10 pages. https://doi.org/10.1145/2766996

3. Chenfanfu Jiang, Craig Schroeder, Joseph Teran, Alexey Stomakhin, and Andrew Selle. 2016. The material point method for simulating continuum materials. In ACM SIGGRAPH 2016 Courses (SIGGRAPH '16). Association for Computing Machinery, New York, NY, USA, Article 24, 1–52. https://doi.org/10.1145/2897826.2927348