Sand and Water Mixing Simulation Using a Multi-Grid MPM

Anjali Thakrar, Ethan Buttimer
CS 284B, Prof. James O'Brien
University of California, Berkeley
e-mails: ath@berkeley.edu, ethanbuttimer@berkeley.edu

I. INTRODUCTION / PROPOSAL GOALS

We set out to implement a multi-species particle interaction using a 2D Material Point Method (MPM). Specifically, we aimed to simulate the mixing of sand and water, following the methods presented in the paper "Multi-species simulation of porous sand and water mixtures" [Tampubolon et. al. 2017]. These methods were primarily centered around modeling different interactions between particles, using the grids as a structure to more efficiently update masses, velocities, and forces, before transferring this data back to the particles and updating their locations. Further, we planned to extend this paper by producing new absorbance effects by adding buoyancy terms and capillary action forces.

II. ACCOMPLISHMENTS

For our implementation, we used the Taichi Graphics library, which is a high performance programming language built on Python and used for graphics applications. Our starting point was an open-source repository implementing a single-grid MPM, based on the paper "High-Performance MLS-MPM Solver with Cutting and Coupling (CPIC)," [Hu 2018].

We first extended this codebase to store particles in two grids — one containing the sand particles, and the other containing water particles. For simplicity, the particles are rendered in the Taichi GUI interface as small square sprites. Before trying to get the grids to interact, we were able to render the water and sand particles as separate simulations on the same screen. We then implemented the water-sand particle interaction relationships in order to render a realistic simulation. However, we couldn't get these interaction forces to work properly given the underlying code describing the water and sand separately (see the "preflame" branch on *284b_mpm*).

Therefore, we attempted to start mostly from scratch to build a multi-grid implementation more closely following the Flame repository (but in Taichi/Python rather than C++). By the end, most of the MPMSolver code was our own, with only some of the underlying Taichi configuration, GUI set-up, and solver initialization remaining from our starting code. The resulting code from this last attempt was the most readable and follows the equations in the paper quite closely. Therefore, we've chosen to include a walkthrough of how this latest code runs, even though there is still something wrong with the cohesion of the particles.

III. CODE WALK-THROUGH

The code itself can be found here, in the files scripts/mpm128_2grid.py and scripts/mpm_solver.py

In all formulae:

- Superscripts n and n+1 indicate the current and next timesteps
- Superscripts s and w indicate sand and water, and alpha indicates when an equation applies to both media types
- Subscripts p and i indicate particles and grid cells

Refer to the Table of notation for variable definitions not discussed directly.

Variable	Where	Species	Meaning
$\mathbf{g}$	-	-	gravitational constant
c_E	-	-	drag coefficient
m_p^α	particles	both	particle mass
$V_p^{\alpha 0}$	particles	both	initial particle volume
$\mathbf{x}_p^{\alpha,n}, \mathbf{x}_p^{\alpha,n+1}$	particles	both	particle position
$\mathbf{v}_p^{\alpha,n}, \mathbf{v}_p^{\alpha,n+1}$	particles	both	particle velocity
$\mathbf{F}_p^n, \mathbf{F}_p^{n+1}$	particles	matrix	deformation gradient
$\mathbf{F}_p^{sE,n}, \mathbf{F}_p^{sE,n+1}$	particles	sand	sand elastic deformation gradient
$\mathbf{F}_p^{sP,n}, \mathbf{F}_p^{sP,n+1}$	particles	sand	plastic deformation gradient
$J_p^{w,n}, J_p^{w,n+1}$	particles	water	determinant deformation gradient
$\phi_p^{s,n}$	particle	sand	water saturation
$c_{C_p}^{s,n}$	particle	sand	cohesion
v_{cp}^s	particle	sand	volume correction scalar
$(\nabla \mathbf{v})_p^\alpha$	particles	matrix	grid-based velocity gradient
$m_i^{\alpha,n}$	grid	both	grid node mass
$\mathbf{v}_i^{\alpha,n}$	grid	both	rasterized velocity
$\mathbf{v}_i^{\alpha,n+1}$	grid	both	final grid velocity
$\mathbf{x}_i^{\alpha,n}$	grid	both	Cartesian grid node locations
$\hat{\mathbf{x}}_i^{\alpha,n+1}$	grid	both	grid positions moved by $\mathbf{v}_i^{\alpha,n+1}$
$\phi_i^{w,n+1}$	grid	mixed	water saturation
$\mathbf{f}_i^\alpha$	grid	both	internal forces

Table 1. **Table of notation.**

A. Scene set-up and initialization

In the file *mpm128_2grid.py*, we initialize custom-defined ParticleGroup objects that can be configured to have a particular size, material, position, and velocity. This is defined in *particle_group.py*.

Next, we initialize a GUI and the MPMSolver, defined in mpm_solver.py (linked above). In the initialization, we declare our global simulation parameters: the number of particles (with this taichi implementation, the number of particles must remain constant), the grid resolution, the time step size, and constants for sand plasticity and water pressure. There are also several Taichi fields to hold per-particle attributes. These include position, velocity, affine velocity field, material, deformation gradients, cohesion, and cached particle-to-grid scattering weights. The velocity, mass, force grids for each particle type are also declared as Taichi fields. Together, these fields hold all of the particle and grid data that change throughout the simulation. Finally, the initialization also prepares the solver to hold information on the explicit forces of gravity and a mouse-controller attractor, as well as the parameters of the particle sources (from ParticleGroup objects) placed at the beginning of the simulation.

These particle sources are placed using the *read_particle_groups()* function, and all of the particle data are initialized when *reset()* is called. Particles are initially randomly scattered within the desired source areas.

For each frame of the simulation, the GUI first checks for user input and adjusts forces accordingly. The solver runs its *substep()* function *n_supsteps* times and finally draws the particles.

B. Particle to Grid (P2G)

Inside *MPMSolver.substep()*, the first step is transferring particle data to the grids. The *initialize_mass_and_velocity_grids()* function iterates through all particles. For each one, the position of the particle on the grid is computed, then we iterate through the 4x4 patch of grid cells surrounding the particle. For each of these grid cells, the distance of the particle from the center of the cell is computed, then used to calculate a Quadratic B-Spline weight (and the derivative of the weight) assigned to that grid cell. Cells closer to the particle receive higher weights, with a steep falloff. These weights are cached at this point to again be used in multiple steps. The weights are used to transfer the mass and velocity of the particle to the surrounding grid cells, according to the following formulae:

$$m_i^{\alpha,n} = \sum_p w_{ip}^{\alpha,n} m_p^{\alpha}$$

$$v_i^{\alpha,n} = \frac{1}{m_i^{\alpha,n}} \sum_p w_{ip}^{\alpha,n} m_p^{\alpha} (v_p^{\alpha,n} + C_p^{\alpha,n}(x_i^{\alpha,n} - x_p^{\alpha,n}))$$

where w indicates the weight linking a given particle and grid cell, C is the affine velocity field local to the particle, and the x term is the difference in particle and grid cell positions. In addition to the existing mass and velocity of the particles, internal stress forces on the particles, calculated as energy derivatives, are also transferred to a grid for each material type. The overall formulae for these forces is given by:

$$f_i^s(\hat{x}^s) = -\frac{\partial \psi^s}{\partial \hat{x}^s} = -\sum_p V_p^0 (\frac{\partial \psi^U}{\partial F^w} F_p^{sE}(\hat{x}^s))(F_p^{sE,n})^\top \nabla w_{ip}^{s,n}$$

$$f_i^w(\hat{x}^w) = -\frac{\partial \psi^w}{\partial \hat{x}^w} = -\sum_p V_p^0 (\frac{\partial \psi^w}{\partial J^w} J^w(\hat{x}^w)) J^{w,n} \nabla w_{ip}^{w,n}$$

The sand energy derivatives (the second term in the first summation above) are based on the elastic potential energy indicated by an elastic deformation gradient matrix. First calculating the singular value decomposition (SVD) of this matrix allows for a more straightforward computation of the energy derivative. We solved for the ψ term, the elastic potential energy density, using the Ducker-Prager formulation as described in the paper "Drucker-Prager Elastoplasticity for Sand Animation" [Klar 2016]. We specifically implemented the constitutive model, as described by this equation in section 6.2 of the paper:

$$\psi^s(F^s) = \tilde{\psi}^s(\varepsilon) = \mu tr(\varepsilon^2) + \lambda 2 tr\varepsilon$$

If we model water as nearly-incompressible, the energy derivative of water can be computed as:

$$\sigma^w = p_w I, p_w = k(\frac{1}{J^{w\gamma}} - 1)$$

Where J_w is the determinant of the deformation gradient, k is the bulk modulus of the water (a constant, called *water_hardness* in the code), and gamma penalizes deviations from incompressibility. We implemented this formula in the *energy_derivative_water()* function.

These energy derivatives are used along with the weight gradient and deformation gradient to implement equations 16 and 17 (above) in the function *explicit_force_grid()*

C. Grid Updates

Next comes the grid update stage, which we implemented in the function *momenta_exchange()*. For non-interacting grid cells (only water or sand, not both) this simply means applying the external force of gravity and the internal stress forces to the velocity grid. For cells with interaction, it becomes more complicated. We followed Section 4.2 of the main paper, except that we implemented forward Euler rather than implicit integration, avoiding the non-linearity of equation 22 by using the existing velocities of the current time step in our explicit force calculations rather than implicitly using the future velocities.

The discrete interaction terms (i.e. drag) for sand and water, respectively, are computed as:

$$d_{ij}^s(\hat{x}) = -c_E m_i^s m_j^w (v_i^{s,n+1} - v_j^{w,n+1})$$

$$d_{ji}^w(\hat{x}) = c_E m_i^w m_j^w (v_i^{s,n+1} - v_j^{w,n+1})$$

where C_e is a drag coefficient dependent on the water saturation and the m terms are the mass of sand and water in a given cell. The velocity terms are being solved for in the resulting coupled system:

$$M = \begin{pmatrix} M^{s,n} & \\ & M^{w,n} \end{pmatrix}$$

$$v = \begin{pmatrix} v^s \\ v^w \end{pmatrix}$$

$$f(\hat{x}(v^{n+1}) = \begin{pmatrix} f^s(\hat{x}^s) \\ f^w(\hat{x}^w) \end{pmatrix}$$

$$(M + \nabla t D)v^{n+1} = Mv^n + \Delta t(Mg + f(\hat{x}(v^{n+1})))$$

where D contains the discrete interaction coefficients, M contains the masses of sand and water in the cell along its diagonal, Mg is the gravitational force, the f term is the stress force, and Δt is the length of the substep. Since we used v^n rather that v^{n+1} in the force term (i.e. we use forward Euler integration), the system is linear and can be solved directly by taking the inverse of the $(M + \Delta t D)$ term and multiplying it by the RHS. This isolates the v^{n+1} term, which is used to update the velocity grids.

Also during this step, the volume fraction of water in each

grid cell is computed and stored in the field grid_saturation.

The grid update step ends with the enforcement of boundary conditions. In our implementation, grid cell velocities are simply set to zero for the three layers of cells along the border. Ideally, friction along the boundary would be defined so that velocities parallel to the border would not be fully killed.

D. Grid to Particle (G2P)

First, the saturation (wetness) of sand particles is computed in the function *update_saturation()* by transferring the saturation grid values to particles using the standard weighting scheme.

$$\phi_p^{s,n+1} = \sum_i w_{ip} \phi_i^{w,n+1}$$

Next, the new velocities computed on the grid can be transferred back to the particles using the same weights from before. In the function *update_velocity_grid()*, the weighted sum of velocities is computed over the 4x4 patch of grid cell surrounding each particle, corresponding with the following equation:

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

In the same loop, we update the affine velocity matrix used in P2G according to the equation:

$$C_p^{\alpha,n+1} = \sum_i w_{ip}^n v_i^{\alpha,n+1} ((\frac{4}{h^2})(x_i^{\alpha,n} - x_p^{\alpha,n}))^\top$$

In the *update_particles()* function, the first thing that happens is simply moving the particle positions based on their new velocities, according to the following equation:

$$x_p^{\alpha,n+1} = x_p^{\alpha,n+1} + \Delta v_p^{\alpha,n+1}$$

Then, the deformation gradients are updated. For water, the following equation is used, implemented in *update_gradient_water()*:

$$J_p^{w,n+1} = (I = \Delta t * tr(\nabla v_p^{w,n+1})) J^{w,n}$$

For sand, this equation is used, implemented in *update_gradient_sand()*:

$$\hat{F}_p^{sE,n+1} = (I + \Delta t \nabla v_p^{s,n+1}) F^{sE,n}$$

The main difference in the above equations is that only the determinant of the elastic deformation gradient needs to be used for water, rather than the matrix itself for sand. The paper argues that this approach offers greater stability.

For sand in particular, the functions *update_cohesion_sand()* and *apply_plasticity_sand()* are called, implementing saturation-dependent cohesion with Drucker-Prager volume correction (Sections 4.3.3-4.3.4). In particular, cohesion is updated according to:

$$c_{CP}^{x,n+1} = c_{CP}^{s,0} (1 - \phi_p^{w,n+1})$$

where ϕ is the particle saturation computed earlier (in the grid stage), and $c^s 0$ is an initial cohesion constant (effectively setting the maximum possible cohesion).

The computed cohesion is then used to apply plasticity to the sand particles to create clumping effects and additional stiffness for wet sand. The *project_sand()* function performs the Drucker-Prager plastic flow and yield condition to determine how the deformation gradients should be updated for the sand. This follows Equation 6 in the paper:

$$c_F tr(\sigma^s) + \|\sigma^s - \frac{tr(\sigma^s)}{d} I\|_F \leq c_C$$

The Drucker-Prager projection itself also happens here, and follows Equation 27:

$$\varepsilon^{sE,n+1} = P(\hat{\varepsilon}^{sE,n+1} + \frac{v_{cp}^n}{d} I)$$

In *apply_plasticity_sand()*, the determinant of the elastic deformation gradient is determined before and after the projection in order to determine volume change. The volume correction term V_c is adjusted to counteract this. This follows Equation 26:

$$v_{cp}^{s,n+1} = v_{cp}^{s,n} + \log(det(F^{sE,n+1})) - \log(det(\hat{F}^{sE,n+1}))$$

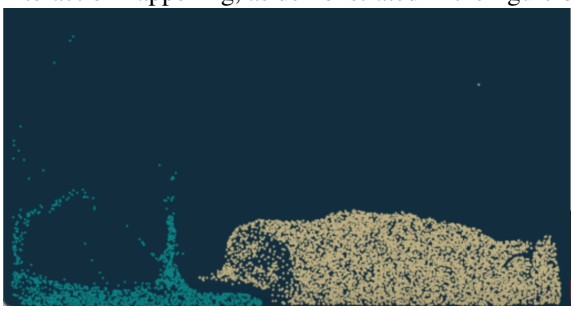
This concludes one iteration of the *substep()* function. The multi-grid MPM solver repeatedly applies this process for a provided number of frames.

IV. RESULTS

A. Single-grid MPM

The implementation described above unfortunately results in sand particles that collapse on themselves rather than deforming correctly. This also means that we can't currently see how the sand and water interact.

In our implementation that built more heavily upon the single-grid MPM starter code (rather than the latest implementation, written mostly from scratch), the sand and water still didn't interact correctly. However, the sand and water each deformed correctly, and there was genuine 2-grid interaction happening, as demonstrated in the figure below.



B. Video demos

Demo 1: two independent grids, each grid containing water and sand, respectively. Commit, Video

Demo 2: One grid, multiple blocks of water interacting with each other. Commit, Video

Demo 3: One grid, multiple blocks of sand interacting with each other. Commit, Video

Demo 4: One grid, sand particles interacting with water particles Commit, Video 1: Sand falling into water, Video

2: Water falling into sand

Demo 5: Two grids, sand particles interacting with water particles (unable to kernelize due to taichi errors, which made it difficult to debug the errors. The result as shown is both slow and non-realistic, which is likely due to the C matrix (description of C matrix) and minor discrepancies in the force computation which snowballed into larger scale effects in the simulation). Commit, Video

Demo 6: Two grids, sand particles interacting with water particles but interaction forces are incorrect. New branch created from an old commit that is less broken than our latest code. Branch, Video, Video with the interaction force zeroed out instead

V. PROJECT ASSESSMENT

We now realize that we greatly underestimated the time it would take to implement this algorithm. Given our lack of experience in the simulation space, we were unable to properly gauge the workload upon coming into this project. We found some of the papers we looked at to be difficult to understand, and even more difficult to implement! We also ran into quite a few hardware-related troubles while working on our CS284B final project — these issues set us back several weeks in our development. We tried computers in various labs on campus and ended up working in the Sutardja Dai 200 computer lab. The computers took some time to set up, given that we did not have admin permissions to install software. However, after working on these computers for a few weeks, we started getting CUDA errors that we were unable to resolve and had to pivot to a different codebase. In all of these cases, we've found Taichi Graphics very difficult to work with and debug, which has further contributed to delays in moving forward with the implementation. In retrospect, it may have been wise to switch to another codebase or implement a test project in Taichi before committing to such a large endeavor, just so we were able to understand the limitations of our hardware and the software itself. However, from this experience, we ended up learning a lot about how to troubleshoot and interface with admins to get work done!

In spite of (and because of) these struggles, we took a lot away from this experience. We both have a much stronger understanding of fluid simulation technology, which was a goal we both had coming into the course. We were fascinated by the effectiveness of transferring data between particles and grids as a process to speed up and improve the stability of fluid simulations. It was exciting to dive deeply into a new paper and project, implementing a cutting-edge concept in computer graphics. We did a lot of digging into other papers to find derivations for equations for the purpose of implementation, which helped us further understand how complex it is to implement this sort of paper with little outside framework.

Early last month, we met up with Andre Pradhana Tampubolon, who is the first author on the paper we based most of our work on. When he heard we were interested in improving how the absorbance of sand is modeled one we'd implemented his paper, he pointed us to a recent paper on porous materials, "Unified Particle System for Multiple-fluid Flow and Porous Material" [Ren et. al. 2021]. However, he was also surprised to hear that we only had a month to re-implement the paper and then build on top of it. Accordingly, he told us to limit the scope of what we were trying to do. While this discussion didn't lead us to pivot to a different topic (as it probably should have), it did convince us to work entirely in 2D. At that point we had already set several reach goals for ourselves in the proposal that went beyond the source paper, but we were just starting to realize how difficult the re-implementation of multi-grid MPM would be on its own.

Neither of us had ever taken a graduate level course before this one. The open-ended nature of the project was both daunting and exciting, as it really allowed us to explore our interests!

VI. FUTURE WORK

Due to hardware constraints, we were unable to render the simulation in 3D. In the future, we hope to extend our code into 3D space. Further, we hope to increase the resolution of the simulation without having excessive runtimes. One method of doing so would be to use a machine learning model to learn velocity and cohesion values, or to upscale the particle count. Further, we could incorporate adaptive particle size techniques, as described by the paper "Highly Adaptive Liquid Simulations on Tetrahedral Meshes" [Ando 2013]. Another avenue of exploration is to implement a more realistic surface tension effect using techniques outlined in "Real-time Animation of Sand-Water Interaction" (Rungjiratananon et al. [2008]). In this implementation, the surface tension induces a "liquid bridge" between particles, which in turn, affects the cohesion and stickiness properties of the sand. Further, every particle has a "wetness" term which also influences the tension and cohesion interactions. This is a method to extend our baseline framework!

VII. REFERENCES (Papers)

Andre Pradhana Tampubolon, Theodore Gast, Gergely Klár, Chuyuan Fu, Joseph Teran, Chenfanfu Jiang, and Ken Museth. 2017. Multi-species simulation of porous sand and water mixtures. ACM Trans. Graph. 36, 4, Article 105 (July 2017), 11 pages. DOI: http://dx.doi.org/10.1145/3072959.3073651

Bo Ren, Ben Xu, and Chenfeng Li. 2021. Unified Particle System for Multiple-fluid Flow and Porous Material. ACM Trans. Graph. 40, 4, Article 118 (August 2021), 14 pages. https://doi.org/10.1145/ 3450626.3459764

Cao, C.; Neilsen, M. Dam Breach Simulation with the Material Point Method. Computation 2021, 9, 8. https://doi.org/10.3390/computation 9020008

Drucker, D., and Prager, W. 1952. Soil mechanics and plasticity analysis or limit design. Quart App Math 10, 157–165.

Yuanming Hu† , Yu Fang† , Ziheng Ge† , Ziyin Qu, Yixin Zhu† , Andre Pradhana, and Chenfanfu Jiang. 2018. A Moving Least Squares Material Point Method with Displacement Discontinuity and Two-Way Rigid Body Coupling. ACM Trans. Graph. 37, 4, Article 150 (August 2018), 14 pages. https: //doi.org/10.1145/3197517.3201293

VIII. REFERENCES (Code)

- Our starter code: https://github.com/yuanming-hu/taichi_mpm
- Codebase we built on in the Sutardja Dai computer lab: https://github.com/Jack12xl/taichi_elements
- A C++ repository we looked over extensively to help create our own original implementation in Taichi/Python: https://github.com/YiYiXia/Flame and its accompanying paper (very similar to [Tampubolon et al 2017])
- Taichi lang: https://www.taichi-lang.org/