Final Project Report - MPM Fluids

Eli Bogomolny, Josh Nadel

December 29, 2018

1 High-level Overview

The explanation of the method we used can be explained with the following steps:

1. Initialization of particle positions using Poisson sampling (pre-process)
2. Initialization of background grid (pre-process)
3. Compute particle kernel weights against the background grid using quadratic interpolation
4. Transfer particle momentum (mass and velocity) to background grid
5. Compute forces due to gravity and stress (using the Neo-Hookean model) for each background grid cell
6. Update background grid velocities proportionally to the force and inverse of mass at each cell
7. Transfer grid velocities to the particles
8. Update particle member variables (APIC matrices, deformation gradients, and stress)

Steps 3. through 8. are the steps that constitute a single iteration of the algorithm, and we ran several thousand iterations in order to achieve a reasonably successful result. The initialization of the background grid will not be explained in detail because there is no mathematical functionality implemented in this step.

2 Poisson Sampling

Poisson sampling is used to create the simulated particle volume from either a user-specified cubic or spherical region. Particle placement is determined according to 3-dimensional Poisson disc sampling [**?**]. Poisson sampling offers the useful property that all points are at least r distance apart, where r is a user-input constant. This allows the initialization to easily produce a particle system of desired density. In addition to the input mesh and the distance r between particles, Poisson sampling takes in a constant k representing the maximum number

of samples the algorithm will choose before rejection. An axis-aligned bounding box with dimensions in R^n is created around the region and used as the domain for Poisson sampling. The algorithm will begin by generating a single random sample point, then continuously add nearby samples at least r distance away to grow the collection of points. Each sample is sequentially indexed with an integer as it is generated to identify each one, and each sample is tracked as "active" when it is generated. If no new samples can be generated around a given active sample within the space constraint, that sample becomes inactive and will no longer be considered for generating neighboring samples. When all samples are inactive, then there is no more space in the domain for new samples and the algorithm is finished.

The algorithm begins by initializing a 3-dimensional background grid with cells of size bounded by $r/\sqrt{n}$, where n is the dimension of the background grid, stored as a n-dimensional integer array. Each grid cell contains one or no samples; -1 means there is no sample and any other value is the index of the sample in that cell. The algorithm then randomly chooses an initial sample within the domain, places it into the background grid, and adds it to a list of indices of active samples. From here, the algorithm can be split into the following steps:

1. While the active list is not empty, repeat steps 2. through 4.
2. Randomly choose an index from the active list i. Randomly generate up to k points between r and $2r$ distance from i.
3. Check if each point is a distance of at least r from all other samples in the background grid.
4. If so, add it to the background grid and active list as a successful sample and discard the other random points. Otherwise, if none of the k points are successful, remove index i from the active list.

3 Kernel Weights

In order to compute the kernel weight for a particle p against a grid cell i, we first need to define a mapping from particle positions to grid cell "neighbors," so that we know which cells should receive momentum information from certain particles. In 3D space, we located 27 neighbors for each particle p by transforming the position of p into grid-space and flooring it to the nearest cell. We considered the nearest cell as the center of the kernel, and used the 26 adjacent cells as the remaining neighbors to the particle. The edge case in which a particle would be floored to a corner cell of the domain (which would result in fewer than 27 neighbors) was handled by creating an additional "frame" of cells surrounding the entire domain. Though this frame was not considered part of the simulated space, we stored the unused cells as indices in our background grid array, such that we did not have to deal with out of bounds exceptions.

Given a particle's 27 neighbors, we used the following interpolation equation (denoted

N), which depends on a particle's distance from a neighbor (denoted x):

$$N(x) = \begin{cases} 0.75 - |x|^2, & |x| < 0.5, \\ 0.5(1.5 - |x|)^2, & |x| < 1.5 \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

As a last note for this step, it is worth mentioning that the different cases for $|x|$ are 0.5 and 1.5, instead of the expected 0.0 and 1.0 (which would seem more natural in grid-space). This is because for more predictable interpolation, we considered the locations of each grid cell at the center of the cell rather than the corner, which we achieved by shifting positions by +0.5 in each direction before performing the interpolation.

4 P2G Momentum Transfer

Due to the implementation of the APIC method, the P2G momentum transfer has two parts. First, we assign for each particle p, we add the following quantity to each of its 27 neighbors' masses (initialized to 0): $w_{ip} * m_p$, where w_{ip} is the kernel weight computed in the previous step and m_p is the mass of the particle. Next, we use the velocity transfer implementation detailed in the APIC paper [**?**]. The intuition behind this method is the formula $p = mv$. At each grid cell, the known quantities at this stage of an iteration are $m_i, m_p, w_{ip}, and v_p$. Since each physical quantity is a summation of interpolated particle values, we can expect that the momentum at a grid cell i is given by the equation -

$$m_i v_i = \sum_p w_{ip} m_p v_p$$

Therefore, in order to solve for the desired velocity v_i, we can divide the right side of the momentum equation by m_i. The velocity v_p is computed using the affine velocity matrix C, stored per particle, according to the following equality -

$$v_p = p.v + C * (x_i - p.x)$$

where $p.v$ and $p.x$ are the velocity and position vectors stored at the current particle, and x_i is the world-space position of the current grid cell [**?**]. In order to avoid divide-by-zero errors, we perform an additional check (as per the APIC paper) to ensure that grid cells only receive a non-zero velocity when they have non-zero mass.

5 Background Grid Force and Velocity Updates

For the purposes of our simulation, we only considered forces due to particle deformation gradients (Neo-Hookian model) and gravity. The force at a grid cell i, according to the Neo-Hookean model presented in [**?**] is given by the following expression -

$$f_i = -\sum_p V_p * \sigma_p * \nabla w_{ip}$$

in which V_p is the volume of a particle p and σ_p is the Cauchy stress at a particle p. In our simulation, we assumed that the volume of each particle is constant, and computations for the Cauchy stress term are outlined in a later section. We calculated the effects of gravity by adding the vector $(0, -m_i g, 0)$ to each cell's total force summation, where g is an acceleration constant that we decided on through trial and error. Finally, we use the equation from [**?**] to update grid cell velocities based on the newly computed forces at each cell -

$$v_i = v_i + \Delta t f_i / m_i$$

where Δt is the floating point duration of a single timestep. The only exception to this equality occurs with cells that are predetermined to be "boundary cells," and are expected to behave like walls. When computing the velocity as a non-zero mass boundary cell, there are two options described in [**?**]. The first option is to simply set the velocity to $\mathbf{0}$, which we used for surfaces that we wanted our particles to stick to (i.e. the ceiling, from which our particles slowly droop down). The second option is to compute a friction force dependent on the direction of the grid cell velocity, using the collision handling from [**?**]. For our purposes, since our only collisions were along the faces of a static cube that surrounded our simulation domain, the final velocity at a boundary cell in this case is given by the following:

$$v_i = v_t + \mu * (v_i \cdot n) * v_t / ||v_t||$$

where v_t represents the tangential velocity relative to the surface, given by $v_t = v_i - n*(v_i \cdot n)$, and n represents the surface normal at the collision.

6 G2P Transfer

Since we assume that particle masses remain constant throughout the simulation, the only component that we transfer from grid cells to particles is velocity. The transfer is done according to the following summation for each particle over its neighboring cells:

$$v_p = \sum_i w_{ip} v_i$$

7 Update Particle Member Variables

7.1 APIC

The first variable updated after the G2P transfer is the particle's affine velocity matrix C. Following the method in [**?**], we can compute the affine velocity C_p^n as the product of an affine state matrix B and the inverse of an inertia-like tensor D: $\mathrm{C}_p^n = \mathrm{B}_p^n (\mathrm{D}_p^n)^{-1}$. The affine state matrix B is given by the following sum over the grid cells:

$$\mathrm{B}_p^{n+1} = \sum_i w_{ip}^n \tilde{\mathrm{v}}_i^{n+1} (\mathrm{x}_i - \mathrm{x}_p^n)^T \tag{2}$$

The inertia-like tensor D (analogous to the tensor K in the RPIC method, which we outlined in our initial paper survey) is given by the following sum over the grid cells:

$$\mathrm{D}_p^n = \sum_i w_{ip}^n (\mathrm{x}_i - \mathrm{x}_p^n)(\mathrm{x}_i - \mathrm{x}_p^n)^T \tag{3}$$

In both equations, the exponent n refers to the number of the iteration being considered (i.e. v_p^n refers to the velocity of particle p during iteration n). Conveniently, in the case of quadratic interpolation weighting of the particles, D takes the form $\mathrm{D}_n^p = \frac{1}{4}\Delta x^2 \mathrm{I}$. where Δx represents the world-space dimension of a single grid cell.

7.2 Deformation Gradient

Each particle's deformation gradient is initialized to an identity matrix and updated according to the following equation:

$$F_p^{n+1} = \left(\mathrm{I} + \Delta t \sum_i \mathrm{v}_i^{n+1} (\nabla w_{ip}^n)^T\right) F_p^n \tag{4}$$

7.3 Stress

Particles are created with initial μ and λ values, defined from user-specified constants as:

$$\mu_0 = \frac{k}{2.0(1.0 + \nu)} \tag{5}$$

and

$$\lambda_0 = \frac{k\nu}{(1.0 + \nu)(1.0 - 2.0\nu)} \tag{6}$$

where k is Young's Modulus, determining the strength of the elastic forces, and ν is Poisson's Ratio, determining the shearing qualities of the material. Each tick, particles' μ and λ values are updated from initial μ and λ values according to:

$$\mu = \mu_0 e^{\xi(1.0 - J)} \tag{7}$$

and

$$\lambda = \lambda_0 e^{\xi(1.0 - J)} \tag{8}$$

Stress for each particle is computed from the following equation:

$$\sigma = 2.0\mu(F_E - (UV^T)) + \lambda(J_E - 1.0)J_E F^{-1T} \tag{9}$$

7.4 Position

At the very end of our particle member variable updates, we update particle positions based on our previously computed particle velocities for the next iteration. This is done according to the simple equation $x_p = \Delta t * v_p$.

8 Elasticity and Plasticity

Elasticity and plasticity is modeled by separating particle deformations into elastic and plastic components F_E and F_P respectively. These components have a key qualitative difference [**?**]. The plastic deformations are those that become "baked" into the volume's resting state. The volume does not exhibit countering forces to undo the plastic deformation. The elastic deformation, on the other hand, is impermanent and causes a characteristic, energy-conserving bounce or wobble to return to the volume's resting state. These deformations are described in section 3.2.1 of [**?**] as

$$F_E^{n+1} = \hat{U}\Sigma\hat{V} \tag{10}$$

$$F_P^{n+1} = (F_E^{n+1})^{-1}F^{n+1} \tag{11}$$

After the U matrix, V matrix, and Σ vectors are computed from F_E, the elements of $Sigma$ are clamped to be between $1 - \theta_C$ and $1 - \theta_S$. This clamping separates plastic and elastic components of the deformation gradient, controlled with user-specified θ_C and θ_S. The clamped $Sigma$ is used to compute the new elastic deformation, which is inverted and multiplied with F to get F_P.

9 Structure of Code

The code begins inside the main class, which initializes the particle grid and runs the simulation loop. Initialization statically runs an initialize function in the Poisson class, which generates uniformly distributed random particle positions inside a user-specified region according to a user-specified density. Each iteration, main calls the method "runMPM" on the particle grid, which populates the grid with particle values, updates the grid according to physical forces, and populates the particles with new grid values. Material properties are controlled through the values θ_C, θ_S, k, ν, ξ, and ρ. An additional step takes place once every specified number of frames, in which the grid writes out the current particles as an obj file to be rendered as the next simulation frame.

10 Results

Our initial goal was to simulate melted cheese, which we more or less achieved. Due to time constraints, we were not quite able to get our material constants to be exactly what we would like, but we were able to get both elasticity and plasticity components working in our simulation. With a couple days of experimentation, we were able to assign constants that came pretty close to our initial goal of melted cheese, and with more time we are confident than we can continue to refine our parameters further.