CIS 563 Project 3 — Material Point Method

Junyong Zhao (64562821)
junyong@seas.upenn.edu

1. Environment and Compilation Setups

- This project followed the structure of previous mass spring system projects, simply type "make" at the top level directory will compile and run the code. The compiled binary and output .bgeo sequence are located under "Projects/material_point/".
- Please notice that this project require C++ 11, Eigen, and Partio library. This project also used professor Robert Bridson's mesh query library to check if the sampled particle is within the model's geometry. They could be found at "Projects/material_point/mesh_query0.1".

2. Data Structures

A. Grid Data Structures

- Four data structures are used in each step, they are initialized as zeros every time. And their update rule basically follows the instructions of week 10 — Hybrid Method.
- **std::vector<int> active_nodes**: used to track the non-zero mass grid node.
- **std::vector<T> grid_mass**: used to track the grid node mass in each transfer from particle mass to grid mass.
- **std::vector<TV> grid_velocity**: used to track the grid node velocity in each transfer from particle velocity to grid velocity. The velocity was kept initially as grid momentum and only those grid with non-zero mass was converged to velocity eventually. If the mass is zero, the velocity is then set to zero.

B. Particle Data Structure

Four major data structures are used to track the particle parameters.

- **std::vector<TV> point_position**: used to store the particle position in each iteration. Initialized as zeros and updated by transferring grid to particle.
- **std::vector<T> point_mass**: used to store the mass of each particle, which is initialized as the point's volume (8 point in 1 grid) times the density (T rho). It is never changed and only used for updating the grid's mass.
- **std::vector<TV> point_velocity**: the velocity of the point. Initialized as zeros and updated by transferring grid to particle.
- **std::vector<TM> point_deform_grad**: the deformation gradient of the particle, which is the $\Psi(\mathbf{F})$ from week 9 — Energy and Stress. This is then used to compute the stress and update point force, etc.

3. Code Structures

A. The main.cpp

- This is the main function of the project and it basically does three things — read the .obj file, parse the object for initializing mesh query structure and generate grid & point, and pass all the above data to simulation driver and run for 240 frames.
- The function read_obj basically just does the standard reading and will output two data structures — vertices and edges, which are the point and face geometry from the .obj file. Each face are then divided into two triangles (since we are only dealing with rectangle in this project and that is required argument for the mesh_query) and they are sent to the construct_mesh_object function from mesh_query to create the data structure that is needed.
- A major goal of the main function is to create the grid and particles needed. To accomplish that, we used a grid size of 0.02 and sampling randomly 8 points in 1 grid. We first generate all grids in the range of [0, 1] (not keeping it, just abstractly) and loop through those grid and sample points. If the grid is within the geometry that we read previously, we will push those sample point coordinates into the point_position data structure. In order to sample 8 random points, we will first divide the 0.02 grid evenly into 8 sub-grids and sample a point randomly in that grid. In this way, we could avoid the points being too dense or sparse. As we could see from the main.cpp line 92, the outer 3 loops are used to loop through all the grids in [0, 1], and the inner 3 loops are used to go through the 8 sub-grid and sample a particle.
- As we read the .obj file of a cube from [0.25, 0.75] generated by Houdini, we will acquire 125,000 points in total.

B. The SimulationDriver.h

- The simulation drier is really to initialize a material_point object, which is the solver and do the step computation that is basically the same thing for the mass spring system project. It will initialize zero grid data structures, pass them to do the computation (the point - grid - collision - point iteration), and dump a .bgeo file based on the computation outcome. The dumpBgeo function is basically brought from the partio example provided by the cispba skeleton code.

C. The material_point.h

- This is the main implementation of the material point solver. The solver contains some major part of polar_svd — doing the polar SVD decomposition, kirchoff_fixed_corotate — does the stress computation through the corotate model, and inter_weight — compute the base node index and the interpolation weight parameter of quadratic spline. These functions are then used in the two major functions, trans_pnt_grd and trans_grd_pnt, which does the transformation between points and grid nodes. The force, velocity and position update are done within the two function. For the boundary condition, we simply set the lower 3 layer of grid to be zero y-direction velocity, which will give the effect of object hitting the ground and bounce back.
- Polar SVD, Kirchoff. These functions are implemented following the lecture slides and sample python script. The polar_svd will do the decomposition and polar changes accordingly. The Kirchoff stress is computed mainly following this derivative of week 11. The basic interpolation weight and the position of the base node is computed following the quadratic B-splines and how it is implemented in the python script.

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

```
stress =
    T(2) * mu * (deform_grad - rotate) + lambda * (jacobi - 1) * jacobi * deform_grad.inverse().transpose();
```

- Transformation A. The two transformation does not only the mass, position and velocity transformation between the point and the grid node, but also did the force, gravity and deformation gradient update using the functions defined above. Transforming from particle to grid node, we first iterate through all the points and compute the stress, base node and weight. Then loop through the surrounding relative grid nodes (27 for 3D case) and interpolate the mass, velocity and forces to the grid node. This function also filter the nodes with zeros mass and update the gravity of each grid. The elastic force is updated by the following rule of week 11. The stress P is computed through the above equation.

$$\mathbf{f}_i = -\sum_p V_p^0 \mathbf{P}(\mathbf{F}^i)^T \nabla w_{ip}$$

```
grid_force[idx] = grid_force[idx] + (-point_volume * stress * deform_grad.transpose()) *
                                    dw_gradient; // elastic force
```

- Transformation B. Transforming values from grid nodes back to particles is also simple. Following a similar procedure of the previous section, the point velocity gradient and the deform gradient following this rule of week 11. The point's velocity and position is then computed by the point velocity gradient (the acceleration).

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

$$\mathbf{v}_p = \mathbf{v}_p + \Delta t \frac{\mathbf{f}_p}{m_p}$$

$$\mathbf{x}_p = \mathbf{x}_p + \mathbf{v}_p \Delta t$$

```
    /* update deformation gradient */
    TM new_deform_grad;
    new_deform_grad = deform_grad * (TM::Identity() + point_velocity_grad * dt);
    point_deform_grad[p_idx] = new_deform_grad;
```

- Collision. The collision is simple setting relative grid node's velocity to zero. And if any particle enter the region, their speed will be update through the grid transformation.

4. Simulation Result

The simulation result could be shown by the video of this link (click it). Through this solver, we successfully implemented an elastic cube hitting the ground and bouncing back. The simulation seems to be working as expected as the result is smooth and close to real-life kinematics.