# CIS 563 Project 3 — Material Point Method

Junyong Zhao

December 22, 2020

## 1 Environment and Compilation Setups

This project followed the structure of previous mass spring system projects, simply typing `"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/"`. 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"` Based on your operating system, there might be some minor changes needed for correct compilation. For more information, please refer to the `README.md` included in the folder.

## 2 Data Structures

### 2.1 Grid Data Structures

Four data structures are used in each step, they are initialized as zeros as instructed.

- `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 are converted to velocity eventually. If the mass is zero, the velocity is then set to zero.
- `std::vector<TV> grid_force`: used to track the force applied on each grid node.

### 2.2 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 from week 9 — Energy and Stress. This is then used to compute the stress and update point force, etc.

# 3 Code Structures

## 3.1 `main.cpp`

- This is the main function of the project and it basically does three things — read the provided `.obj` file (generated by Houdini, a simple cube within `[0, 1]`), parse the object for initializing mesh query structure, generate grids and points, and pass all the above data to the simulation driver and run for 180 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 too 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 in the range of `[0.25, 0.75]` generated by Houdini, we will acquire 125,000 points in total, which will be complex enough for the sake of this project.

## 3.2 `SimulationDriver.h`

The simulation drier is really to initialize a `material_point` object, which is the solver and do the step computation. This is essentially the same structure of 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 in each step. The `dumpBgeo` function is basically brought from the `partio` example provided by the `cispba` skeleton code.

- Helpers: Polar SVD, Kirchoff Corotation and Interpolation. The polar singular value decomposition and kirchoff corotation computation is embedded in the function which transfers point data to grid data (`trans_pnt_grd`). The SVD is achieved through the library function

of `Eigen::JacobiSVD`. The Kirchoff corotation is computed mainly following this derivative work of week 11. The interpolation weight and the position of the base node is computed following the quadratic B-splines in function `inter_weight`, and it is implemented similar in the provided python script. A detailed code design of the major process is listed below. The stress `P` is computed through the following equation.

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

```
/* The polar svd implementation */
TM u, v;
Eigen::JacobiSVD<TM> svd_decomp(deform_grad,
    Eigen::ComputeThinU | Eigen::ComputeThinV);
u = svd_decomp.matrixU();
v = svd_decomp.matrixV();

/* polar changes */
if (u.determinant() < 0)
    u.col(2) = -u.col(2);

if (v.determinant() < 0)
    v.col(2) = -v.col(2);

/* compute stress and weight */
TM rotate = u * v.transpose();
T jacobi = deform_grad.determinant();

/* Computing the kirchoff stress by differentiating Psi */
stress = T(2) * mu * (deform_grad - rotate) + lambda * (jacobi - 1) *
    jacobi * deform_grad.inverse().transpose();

inter_weight(w, dw, base, x);
```

- `trans_pnt_grd`: The two transformation does not only the mass, position and velocity transformation between the point and the grid node using the functions defined above, but also did the force, gravity and deformation gradient update by looping through all the grids. 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 in `active_nodes`. The elastic force is updated by the following rule of week 11, the main equation and code block listed below.

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

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

- `trans_grd_pnt`. Transforming values from grid nodes back to particles is similar reversely to the function above. Following a similar procedure of the previous section (iterating through all points, and the surrounding grids), the point velocity gradient and the deform gradient could be given by the following formulas of the week 11 lecture notes. The point's velocity and position is then computed by the point velocity gradient (the acceleration). The main equations and code block are listed here.

$$\boldsymbol{F}_p = (\boldsymbol{I} + \Delta t \nabla \boldsymbol{v})\boldsymbol{F}_p$$
$$\boldsymbol{v}_p = \boldsymbol{v}_p + \Delta t \frac{\boldsymbol{f}_p}{m_p}$$
$$\boldsymbol{x}_p = \boldsymbol{x}_p + \boldsymbol{v}_p \Delta t$$

```
/* 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 and boundary conditions. The collision is simply setting relative grid node's velocity to zero. And if any particle enter the region, their speed will be update through the grid transformation. This simulation only achieved the effect of "bouncing back", which means there's only bottom boundary on y-direction set.

## 4 Simulation Result

The simulation result could be shown by the video of this link (click to follow). Through this solver, an elastic cube hitting the ground and bouncing back is successfully implemented, satisfying all basic requirements of the project. The simulation seems to be working as expected as the result is smooth and close to real-life kinematics.
There are several improvements that could be make none the less, for example there could be a better `.obj` file adapted reading/parsing function so that we could scale some more complicated (and maybe more interesting geometries) into triangles in a suitable range to for the `mesh_query` libraries. Another observation is that there's only one type of boundary condition implemented in this project and it must be admitted that two object colliding in mid-air or bouncing on a side wall is also worth investigating, not to mention other objects such as snow, smoke or fluid. Last but not least, the result is presented raw in Houdini real-time playback. A better build render project with materials and textures would give a better effect in the final demo output.