Susan Xie
CIS 563
Chenfanfu Jiang
31 December 2020

**Project 3: Material Point Method**

For this project, I implemented the material point method using the corotated elasticity model. To execute the project, run `./material_point` followed by the filename and path of the object to simulate (must be in stripped down `.obj` format with only vertices and faces; no texture data, normal data, group data, etc). More details on additional arguments that can be given to the executable can be found in README (also attached at the end of this document).

**Data Structures**

For this simulation to work, I had to maintain data for a grid and data associated with a set of particles. To maintain the grid, I made a new class **`Grid.h`**, and to maintain the particles, I used the **`Partio`** library. Within `Grid.h`, I maintain the following parameters:

- `origin`: the grid origin, or the minimum corner of the grid
- `dims`: grid dimensions, or the number of cells the grid is divided for x, y, and z
  *this is a vector value, so a unique number of cells can be set per dimension
- `h`: the width of each grid cell; stores a separate value for x, y, and z
- `buffer`: number of cells to "line" the grid; this is used to ensure particles stay within the boundaries of the grid
- `density`: this is the number of particles that will be sampled for each grid cell
  *should be a value between 8 and ~30
- `parts`: particle data, maintained by the `Partio` library
- `mH`: handle to access particle mass attribute
- `xH`: handle to access particle position attribute
- `vH`: handle to access particle velocity attribute
- `m`: vector maintaining mass at each grid point
- `v`: vector maintaining velocity at each grid point
- `f`: vector maintaining force at each grid point

Once a grid is initialized with an origin, maximum corner, dimensions, buffer, and particle density, there is a function `generateSamples()` which takes in a `MeshObject*`, as defined by Robert Bridson's `mesh_query` library, and an object `mass`. The mesh is generated from the user-input .obj to be simulated, and the `mass` can also be set within `main.cpp`. The function then uses a C++ standard library random number generator to generate random particle positions and initialize particles to add to the grid's `parts` particle group. Before adding a point to this group, the function also uses Bridson's `point_inside_mesh` function to check if the sampled point lies within the object to be simulated—this allows the grid to create a cloud of sampled particles in the shape of the object mesh to be simulated.

The particles themselves are maintained by the `Partio` library, specifically within a `ParticlesDataMutable*` structure. Particles each maintain their own mass, position, and velocity, and this is repeatedly accessed and updated throughout the simulation. For deformation to work, each particle also needs to maintain a deformation gradient `F`. However, I found `Partio`'s attribute handling to be very limited here; `Partio` has a vector, float, and integer datatype for particle attributes, but the deformation gradient is a matrix. So, rather than storing `F` within `Partio`'s particle data framework, I maintain a vector of `F` values within the

MPM solver itself, as well as an intermediary vector `updateF` to store intermediate values needed to compute the updated `F` for the next simulation step.

**Code Organization**

Besides `Grid.h`, which was simply used to define a data structure, the code contains three primary files: **`main.cpp`**, **`SimulationDriver.h`**, and **`MPMSystem.h`**. The main file, `main.cpp`, is used to manage user-input and output, sets up the grid appropriately, and then calls the driver to start the simulation. Based on the user-provided filename, `main.cpp` constructs a mesh object from the .obj, constructs a grid, and samples points for the mesh within the grid. There are a series of parameters at the top of the file as well that define values for Young's Modulus, Poisson's Ratio, timestep dt (for simulation driver), an offset for the position of the given .obj, grid cell dimensions, grid origin, grid maximum corner, grid buffer (number of cells lining the edges of the grid which act as the grid's boundaries), and object mass. This is also described in greater detail within the README.

Once a satisfactory number of particles are generated for the simulation, the `SimulationDriver` driver is set up to contain all the values it needs, and the driver begins the simulation. At each frame of the simulation, the driver advances one step of the MPM solver and generates a `.bgeo` file containing updated particle data. To advance one step, the driver does the following:

```
mpm.zeroValues();
mpm.interpolateMassVelToGrid();
mpm.setGridVelocities();

applyGravity();

mpm.computeForces();
mpm.addForceGridVelocities(dt);

mpm.interpolateVelToParticles();
mpm.computeNewF(dt);

// move particles
Partio::ParticlesDataMutable* parts = mpm.parts;
for (int idx = 0; idx < mpm.parts->numParticles(); idx++) {
    float* xp = parts->dataWrite<float>(mpm.grid->xH, idx);
    float* vp = parts->dataWrite<float>(mpm.grid->vH, idx);

    xp[0] += vp[0] * dt;
    xp[1] += vp[1] * dt;
    xp[2] += vp[2] * dt;
}
```

The majority of the math needed to compute each frame of the simulation is within `MPMSystem.h`. The first thing that is done at each step is `ZeroValues()`—this sets all velocities, masses, and forces within the grid to be 0, as well as clears the intermediary vector used to compute weights when updating deformation gradient F. Once data from the previous timestep has been cleared, mass and velocity is interpolated to the grid nodes from particles based on a quadratic b-spline interpolation, implemented by `nHat()` and `Ni()` within `MPMSystem.h`. After the MPM solver computes the appropriate velocities, the simulation driver applies gravity to the grid nodes. The MPM solver then needs to calculate forces, as determined by the deformation gradient and stress. Once all the forces on each grid node are calculated, they are then converted to velocity values and added to the existing velocity, so that the final velocity of each grid node includes both freefall and deformation. This final velocity is finally interpolated back to the particles, and the deformation gradient F is updated after each frame. Finally, based on the new particle velocities, the driver updates each particle's position.

Essentially, `main.cpp` handles data that the user may want to change when simulating different objects using the program, `SimulationDriver.h` handles advancing simulation steps and writing output files, and `MPMSystem.h` handles all the math needed to actually update the particles correctly.

**Simulated Results**

The demos I created include 3 separate simulations: a pink jello cube, a blue jello cube, and two blue falling jello cubes. The pink jello cube has a Young's modulus of 10000 and Poisson's ratio of 0.35 while the blue cubes all have a Young's modulus of 10000 and a Poisson's ratio of 0.4. The first two individual cubes are simulated with 120,000 particles each, and the two cubes intersecting are simulated with a total of ~220,000 particles (precisely 218,732).

**Pink cube and blue cube**:
Particle View
Rendered Geometry

**Intersecting cubes**:
Particle View
Rendered Geometry

Overall, I am pretty happy with the results, and I was happy to see that I could use Young's Modulus and Poisson's Ratio partially based on the real values that jello has—one source online said that jello usually has a Poisson's Ratio between 0.3 and 0.5, and another stated jello had a Young's Modulus of about 81kPa. I left my simulation at 10kPa however, since I found increasing Young's Modulus too much would cause the simulation to explode unless I further decreased dt, which took much longer to simulate. I thought 10kPa gave a satisfactory result, and so I left it there to get the bouncy jello-like effect. If I were to work more with MPM, I think I would like to try snow or fluids, but I am fine with just a couple bouncy jello cubes for now.

**README:**

# Material Point Method
Files referenced here are located under Projects > material_point
1) Run "make" or "make all" from project root to compile the executable
    * The C++ Eigen library and Partio library are required for this project.
2) Type "./material_point" from terminal. The executable requires 1 argument--the name of the mesh .obj to be simulated--and has 1-3 optional arguments:
    a. You can pass in a 2nd integer argument to dictate the point density

of the point sampling aka the number of points to be sampled per grid cell. By default, this is 8.
i.e. ./material_point cube.obj 10

b. You can pass in a 3rd integer argument to set the grid dimensions. If you pass in only these 3 arguments, the grid's x y and z dimension will all be set to this 3rd argument.
i.e. ./material_point cube.obj 10 100

c. You can also pass in a set of different dimensions for the grid's x, y, and z.
i.e. ./material_point cube.obj 10 100 64 90

2.1) Additional parameters are located at the top of main.cpp
- youngs_modulus
- poissons_ratio
- dt: simulation time step
- offset: offset given for mesh location, useful for repositioning within grid
- dimensions: grid dimensions (number of cells)
- origin: minimum corner of grid
- grid_max: maximum corner of grid
- grid_buffer: number of cells lining the outside of the grid
- mass: object mass

2.2) Number of frames is set at the bottom of main.cpp; replace the number in "driver.run()" with however many frames you want to simulate.

3) The executable will print out the number of particles in the simulation, based on the grid dimensions and particle density set. If satisfied with the number of particles sampled, typing "y" will prompt the simulation to start, and typing anything else will cause the simulation to abort.

4) Output simulation files will be in an output folder. These .bgeo files can be rendered in Houdini using the rendering.hipnc attached.

The current values are optimized for rendering the following:
./material_point data/cube.obj 15 100

This should give you a nice, bouncy jello cube! :)