Spencer Webster-Bass

CIS 563 Final Project Report

The general workflow of the program is to generate the per particle information representing the center of the discretized volume of the cube which will be our material configuration. This information includes generating per point positions, masses, velocities, and deformation gradients. Other information necessary for the Material Point Method (MPM) are passed over to the simulation driver such as the dilatational and shear terms and the resolution, size, and density of the nodes of the grid. With all of these parameters initialized, the SimulationDriver and MassSpringSystem classes can operate on this information using MPM to compute the deformed configuration of the cube frame-by-frame. The necessary information for describing the system’s behavior over time was exported as a .poly file to be visualized in Houdini.

The Grid data structure contains the node spacing, grid resolution (number of nodes per dimension), and grid origin. The data structure assumes that the boundaries of the simulation region is a cube of size grid spacing times the resolution minus one. The grid object also contains vectors for the mass, velocity, and forces of each grid node. There are also utility functions for indexing these nodes from 3D indices to 1D indices and 3D index to grid position. The MassSpringSystem class contains an instance of this Grid class.

With all of this information, the simulation can now run by computing the next state of the system based on the previous state of the system. The numerical scheme for the simulation to perform the transition from one state to the next state is called MPM Explicit Euler. The steps of this scheme are all implemented in SimulationDriver::advanceOneStepMPM().

With initialization complete, we transfer each particle’s mass and velocity over to the grid using weighted sums. These steps are performed in Grid::updateGridMass(), Grid::updateGridMomentum(), and Grid::momentumToVelocity(). These weighted sums utilize a kernel function which causes all weights at the point of information to sum to 1. This kernel function uses B-Spline Interpolation to allow for continuous 1st derivatives and smoother transitions between frames. To further optimize the performance of the program, whenever we do per grid node operations per particle, we only use the 27 grid nodes inside of a 3x3x3 cage surrounding the particle of interest.

In MPM, instead of applying forces to particles, we apply forces to the nodes of the grid. So once we have the mass and velocity of every node in the grid from the previous steps, we can easily compute the force of gravity’s effect on the grid node’s velocity. This will simply be the current velocity plus the acceleration due to gravity times how much time has passed between the current and previous time step. Before the force of gravity is applied, I check to see if the nodes are in the ground or some unmovable volume, in which case I set the velocity to zero. This is computed in Grid::applyGravity().

With the gravitational force computed and applied, the elastic force must also be computed and applied for each grid node. The elastic force on a node is the negative gradient of the elastic potential energy evaluated at the position of the target grid node. After lots of derivation, calculus, linear algebra, and applying principles such as rotational invariance and isotropy the elastic force on a node can be rewritten in the following form:

This function was evaluated in Grid::computeElasticForce(). F is the deformation gradient. is the volume of the particle in the material we are simulating.

The gradient of the weight function is the derivative of the piecewise B-Spline Interpolation function with respect to each spatial dimension. This was computed in Grid::kernelGradient() and Grid::BSplineInterpDerivative(). For this project we had to compute the derivative of the math in the Grid::BSplineInterp() for which I got:

if (0 <= ax && ax < 0.5f) {

                return -2 \* x;

            } else if (0.5 <= ax && ax < 1.5f) {

                return x \* (1.5 - ax) / ax;

            } else if (1.5f <= x) {

                return 0.f;

            }

P is the stress which is the derivative of the energy density function with respect to the deformation gradient (F). The energy density function for this simulation, uses the Corotated Model which defines the energy density function as follows:

The stress can be expressed as follows:

My simulated results were not good. My code works fine with only gravity but throws errors when I add the elastic force code. So the results only show the effects due to gravity unfortunately. The errors were out of bounds errors however, after checking right before every vector access the index that I am using to access the vector is still in bounds. I’ll have to continue debugging to see why I am receiving out of bounds errors only when I add in the elastic code. So, eliminating these errors would be my first step in resolving this issue. Hopefully after eliminating these errors the program will run smoothly. But with only gravity the simulation looks great since it is falling only using the MPM framework and Grid data structure and the object doesn’t fall through the ground which I specified to be a plane at the origin with a normal in the positive y direction. However, because the elastic force is not working correctly there is no bouncing upon impact with the ground and the object instead just slowly compresses itself.

Below you can find some visualizations from my simulator and code on GitHub:

<https://github.com/spencerwb/cis563/tree/master/Projects/mass_spring>