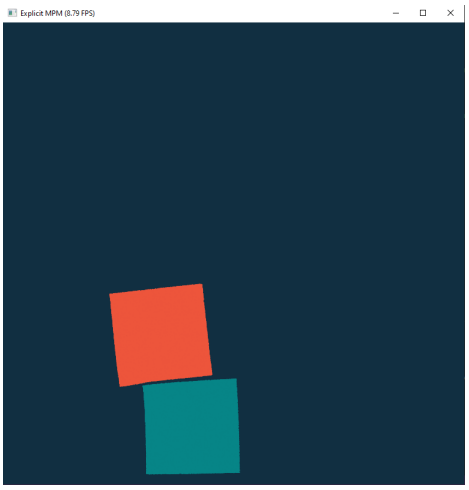
1 Project Description

This is the final project for Physically Based Animation (CIS-563), where I used the 2D Taichi MPM sample code provided by Dr. Jiang as a basis to implement a 3D MPM simulation of elastic material.

2 Setup Instructions

The MPM simulation for this assignment is coded in Taichi/Python3.8 on the Windows 10 platform. In order to install Taichi I followed the instructions here. When I was setting it up on my environment, I had to additionally install Python3.8 and the pip installer that was recommended on the console to make the Taichi code compile and run. If you don't have it already, you must also have the **numpy** library installed before you can run this program. I additionally installed Houdini Apprentice on Windows to import the OBJ sequences of MPM particles and create a rendered demo, although this is not necessary to run the Taichi program itself. Without this 3D render, you should still be able to see a 2D visualization of the 3D simulation on the popup GUI window once you run the program. When you're ready to run the code, simply cd into the same directory where the **two-cubes.py** file exists and then run the **python two-cubes.py** command from Visual Studio Code (or the Git Bash terminal if you do not have VSC).



Figur 1: Screenshot of Taichi GUI window from two-cubes.py execution

3 Code Overview

This section serves as a summary of how I used the 2D Taichi MPM code sample to achieve a 3D simulation. Rather than reimplementing everything from stratch, I enhanced the 2D setup to support 3D position/force/velocity/etc. data rather than 2D. In order to use the benefit of parallelization, we must include the following statement at the top of our code file:

```
ti.init(arch=ti.gpu) # Try to run on GPU
```

Figur 2: In the example code, we assign the ti alias to taichi library

Data Structures

Before we begin any computation, we must first set our data structures where we will store the particle information as well as the simulation parameters. We first want to set the total number of particles our MPM simulation should have (in our case, it is equal to 144054) and the number of grids (in our case, it is equal to 128). From the number of grids we can get the length of each grid cell, $dx = h$ as well as $inv\_dx = \frac{1}{h}$. Our timestep is set to $1e-4$ and the mass of each particle is set to $p_{mass} = p_{vol} \times p_{rho}$ where $p_{vol} = (dx \times .5)^2$ and $p_{rho} = 1$. For the elastic material, Young's modulus is set to $1e-3$ and Poisson's ratio is set to .3. We also have two Lame parameters mu_0 and $lambda_0$ initialized, however these parameter values can be adjusted later if we want to simulate snow or sand (you can check the taichi_mpm repo to get more information on these parameters). All the objects in both two-cubes.py and bunny.py are made of the same elastic material with parameters discussed here.
In order to store particle information throughout the simulation, we create multiple Taichi vector fields as in Figure 3; you can check it out to see what each field corresponds to. These structures are taken from the 2D example, then their dimensions are changed to support 3 dimensional data. In order to avoid repetition, I will only go through the additional vector fields created for the 3D code are as follows:

- $host_x$: This vector field is used for storing particle positions read from an OBJ file for arbitrary meshes in the Python scope and then for copying the Python scope data into Taichi scope.
- x_{2d} : This vector field is used for storing the 2D coordinates of particles to represent them as circles in the Taichi GUI.
- C : This matrix field corresponds to affine velocity field, required for APIC (affine particle in cell) method that prevents the rotational velocity field in a grid to become zero. Although it existed in the 2D code as well, it's necessary to mention its importance.
- $material$: This is used for differentiating the two elastic cubes from each other in the Taichi visualization. In the current setup, both cubes are made from the same material, however using material ids can be useful to assign different material parameters or even different material types for different scene objects/particles.

Using Taichi vector/matrix fields is similar to using the **eigen3** library in C++ to store the simulation data, which provides efficient access to each particle data by index query.

Functions & Kernels

I have some familiarity with kernel functions from GPU programming where I often wrote CUDA kernels. As the Taichi guide describes, kernels are in the Taichi-scope compiled and executed on CPU or GPU devices with performance gain from parallelization. Code outside of kernels are in Python-scope run with slower performance but more program flexibility. Here is a quick overview of the kernels in this program:

```
x = ti.Vector.field(3, dtype=float, shape=n_particles) # position
host_x = ti.Vector.field(3, dtype=float, shape=n_particles)
x_2d = ti.Vector.field(2, dtype=float, shape=n_particles) # 2d positions - this is necessary for circle visualization
v = ti.Vector.field(3, dtype=float, shape=n_particles) # velocity
C = ti.Matrix.field(3, 3, dtype=float, shape=n_particles) # affine velocity field
F = ti.Matrix.field(3, 3, dtype=float, shape=n_particles) # deformation gradient
material = ti.field(dtype=int, shape=n_particles) # material id
grid_v = ti.Vector.field(3, dtype=float, shape=(n_grid, n_grid, n_grid)) # grid node momentum/velocity
grid_m = ti.field(dtype=float, shape=(n_grid, n_grid, n_grid)) # grid node mass
gravity = ti.Vector.field(3, dtype=float, shape=()) # gravity
```

Figur 3: List of simulation data vector fields initialized before the simulation

- $kirchoff\_FCR$: Computes the Kirchoff elastic stress
- $substep$: Executes the simulation and updates all the particles for one substep
- $reset$: Sets the initial simulation configuration

We will go over the substep execution in more detail.

Simulation Substep

All the steps of MPM computation are noted in the py file, but we will go over each code snippet that corresponds to MPM steps that we covered in Week 14.

1. **Clean grid data by zeroing out everything:** We iterate over every grid and set the grid velocity and mass to zero.
2. **Transfer mass & momentum from particles to grid:** In order to transfer particle data to grid, we need to know which grid nodes we need to consider. We first compute the base index for the particle. In the same loop where we iterate over neighboring

```
# First for particle p, compute base index
base = (x[p] * inv_dx - 0.5).cast(int)
```

Figur 4: Base index computation

grids and add the force to momentum (explained later), we transfer the mass and momentum from particle to grid by following these equations.

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

Figur 5: Mass transform

The $dpos$ in the code corresponds to $(x_i - x_p)$ in the momentum transfer equation, which is the difference between the positions of neighbor grid node and the current grid node p at our outer loop iteration. As we're moving from grid to particle, we will update the C_p matrix.

$$(m\boldsymbol{v})_i = \sum_p m_p w_{ip}(\boldsymbol{v}_p + \mathbf{C}_p(\boldsymbol{x}_i - \boldsymbol{x}_p))$$

Figur 6: Momentum transform

$$\mathbf{C}_p = \frac{4}{\Delta x^2} \sum_i w_{ip} \boldsymbol{v}_i (\boldsymbol{x}_i - \boldsymbol{x}_p)^T$$

Figur 7: C_p matrix update

3. **Go over all grid nodes and set velocity:** If the grid mass is non-zero, we update the grid velocity by extracting it from momentum. We were storing the momentum instead of grid velocity in $grid_v$ before this step, so we can basically divide $grid_v$ by $grid_m$ to get the velocity. We also check the current grid and velocity against the boundary conditions and adjust the velocity so that particles would bounce off the walls as expected.

```
# Gravity and Boundary Collision
for i, j, k in grid_m:
  if grid_m[i, j, k] > 0: # No need for epsilon here
    # Step 4: Set velocity from momentum if mass != 0
    grid_v[i, j, k] = (1 / grid_m[i, j, k]) * grid_v[i, j, k] # Momentum to velocity
    # Step 5: Apply gravity on grid
    grid_v[i, j, k] += dt * gravity[None] * 9.8 # gravity

    #wall collisions - handle all 3 dimensions
    if i < 3 and grid_v[i, j, k][0] < 0:          grid_v[i, j, k][0] = 0 # Boundary conditions
    if i > n_grid - 3 and grid_v[i, j, k][0] > 0: grid_v[i, j, k][0] = 0
    if j < 3 and grid_v[i, j, k][1] < 0:          grid_v[i, j, k][1] = 0
    if j > n_grid - 3 and grid_v[i, j, k][1] > 0: grid_v[i, j, k][1] = 0
    if k < 3 and grid_v[i, j, k][2] < 0:          grid_v[i, j, k][2] = 0
    if k > n_grid - 3 and grid_v[i, j, k][2] > 0: grid_v[i, j, k][2] = 0
```

Figur 8: Set velocity

4. **Apply gravity on grid:** We update the velocity by gravitational acceleration for each grid.

```
# Step 5: Apply gravity on grid
grid_v[i, j, k] += dt * gravity[None] * 9.8 # gravity
```

Figur 9: Update velocity with gravitational acceleration

5. **Add elastic force:** We compute the elastic force from the Kirchoff stress and weight gradient. The Kirchoff stress computation requires the deformation gradient matrix F for the particle as well as the U, Σ and V matrices from the SVD of F. The diagonal entries of Σ represent the singular values of F, from which we can get $det(F)$ by

multiplying all singular values, thus calculate the volume change J. Since Taichi's SVD computation is already polar (as mentioned in week 9 lecture) we don't need to do extra work to handle negative determinants. The U and V^T are used as rotations that we pass to the Kirchoff computation kernel as a single rotation matrix $R = U \times V^T$. The Kirchoff stress is computed from the following equation:

$$2\mu(\boldsymbol{F} - \boldsymbol{R})\boldsymbol{F}^T + \lambda(J-1)J$$

Figur 10: The Kirchoff stress equation

The code computes the Kirchoff stress for FCR model by using the volume change J (which is equal to $det(F)$), deformation gradient F, closest rotation matrix to deformation gradient R, μ and λ values. This is plugged back into the following elastic force equation; keep in mind that the Kirchoff stress corresponds to PF^T, not only P (and V_p^0 corresponds to p_{vol}).
For the force computation, we also need the weight gradient. We compute it from the provided quadratic kernels that correspond to $N(x) = w$. These quadratic kernels account for a 3x3x3 grid block span, thus classifies particles by their nearest grids. We also compute $N'(x) = dw$ by hand.

```
# Compute for 3D
dweight = ti.Vector.zero(float,3)
dweight[0] = inv_dx * dw[i][0] * w[j][1] * w[k][2]
dweight[1] = inv_dx * w[i][0] * dw[j][1] * w[k][2]
dweight[2] = inv_dx * w[i][0] * w[j][1] * dw[k][2]
```

Figur 11: The weight gradient computation

Once the force is computed, we add it to corresponding particle $grid_v$. This will be done as we iterate through all neighboring grids. However since $grid_v$ is actually storing the momentum rather than the velocity at this stage we will not divide it by mass yet.

$$\boldsymbol{f}_i = -\sum_p V_p^0 \boldsymbol{P}\boldsymbol{F}^T \nabla w_{ip}$$

Figur 12: The elastic force computation

6. **Interpolate velocity back to particles:** We go back from grid to particle. We iterate over every particle once again and for each particle we iterate over its neighboring grid nodes to interpolate velocity back to particles. We also need to interpolate the deformation gradient back, which can be done either at this step or next step since updating the particle positions does not impact the F matrix. The Taichi code is currently handling it after it moves the particles.

7. **Update particle positions:** Finally, we update the current particle's position and the deformation gradient.

```
for i, j, k in ti.static(ti.ndrange(3, 3, 3)): # loop over 3x3x3 grid node neighborhood
  dpos = ti.Vector([i, j, k]).cast(float) - fx
  g_v = grid_v[base + ti.Vector([i, j, k])]
  weight = w[i][0] * w[j][1] * w[k][2]

  # Compute for 3D
  dweight = ti.Vector.zero(float,3)
  dweight[0] = inv_dx * dw[i][0] * w[j][1] * w[k][2]
  dweight[1] = inv_dx * w[i][0] * dw[j][1] * w[k][2]
  dweight[2] = inv_dx * w[i][0] * w[j][1] * dw[k][2]

  new_v += weight * g_v
  new_C += 4 * inv_dx * weight * g_v.outer_product(dpos)
  new_F += g_v.outer_product(dweight)
# Step 7: Interpolate new velocity back to particles
v[p], C[p] = new_v, new_C
```

Figur 13: Interpolate velocity back to particles

```
# Step 8: Move the particles
x[p] += dt * v[p] # advection
x_2d[p] = [x[p][0], x[p][1]] # update 2d positions
F[p] = (ti.Matrix.identity(float, 3) + (dt * new_F)) @ F[p] #updateF (explicitMPM way)
```

Figur 14: Move the particles and update F matrix

Simulation Initialization

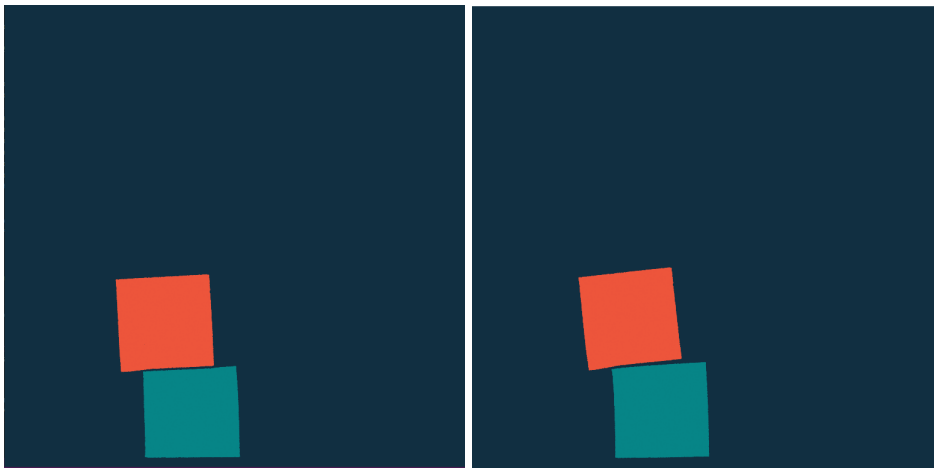
We setup the initial state of our particles inside the *reset*() kernel. We basically fill out the vector and matrix fields in Taichi scope with initial position, velocity, deformation, etc. values.
For scenes with arbitrary meshes (i.e. bunny.py) we read particle positions from an OBJ file rather than do the particle sampling ourselves. However, we handle the OBJ parsing on the Python scope, thus need an extra vector field to store the initial particle positions on the host side. Once we're done reading the particle positions from the file and apply the necessary transformations on the position, we call the reset kernel and we first copy the position data from the host vector field to Taichi scope.
In two-cubes.py, we color the cubes differently so that we can recognize them and we also split the total number of particles into 2 sub groups so that both cubes are made of the same number of particles (although this would only be possible with even total particle numbers).

4 Results

I have created two scenes with elastic materials where one scene is made of two cubes (two-cubes.py) and the other scene reads particle positions from an OBJ file to create two elastic bunnies (bunny.py). I used Houdini to sample a point cloud within the bunny geometry volume and exported it as an OBJ file (bunny_point.obj). The Houdini file for this is included in the Google drive; if you want to run it make sure that you have bunny.obj in the same directory. You can see the demo for two-cubes.py both in 2D Taichi visualization and 3D Houdini render formats. The demo for bunny.py is recorded in both formats as well.

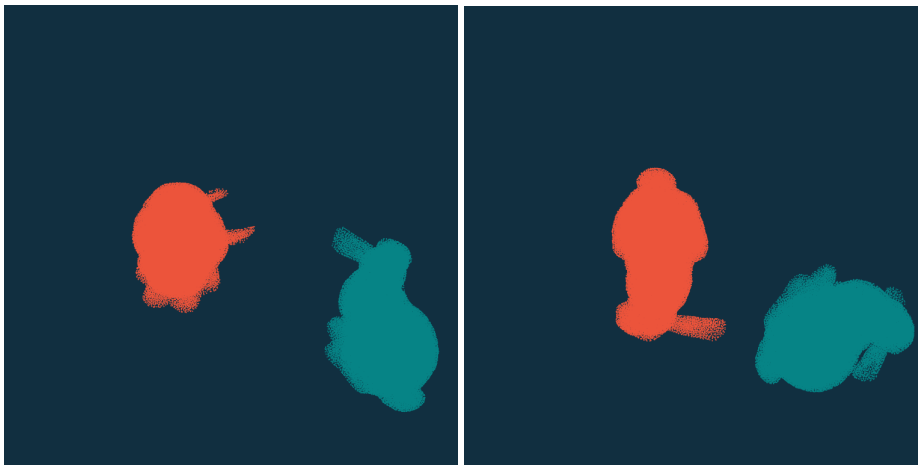
4.1 Thoughts on Results

I'm able to observe elastic behavior in the two-cubes.py scene where one cube will fall onto the other cube in the beginning and will bounce off. I ran the simulation while keeping Young's modulus constant ($1e3$), but decreasing Poisson's ratio to 0.15 from 0.3. When the ratio is lower, I noticed that the bottom cube deforms less at contact compared to the result with higher ratio, which is expected.



Figur 15: $nu = 0.15$ (left), $nu = 0.3$ (right)

I did a similar test with bunny.py except this time I tested with different Young's modulus values while keeping the Poisson's ratio constant (0.3). In the very similar amount of iterations, we observe that the jellos with higher Young's modulus value rotate more than those with lower YM. I believe this is happening because the amount of a jello entity rotating is related to angular velocity, which is related to the force applied. From our formulas, the force on a particle is proportional to the Kirchoff stress, which is proportional to both μ and λ. Both μ and λ Lame parameters are proportional to Young's modulus. Thus, higher YM results in more rotation.



Figur 16: $E = 1e3$ (left), $E = 2e3$ (right)