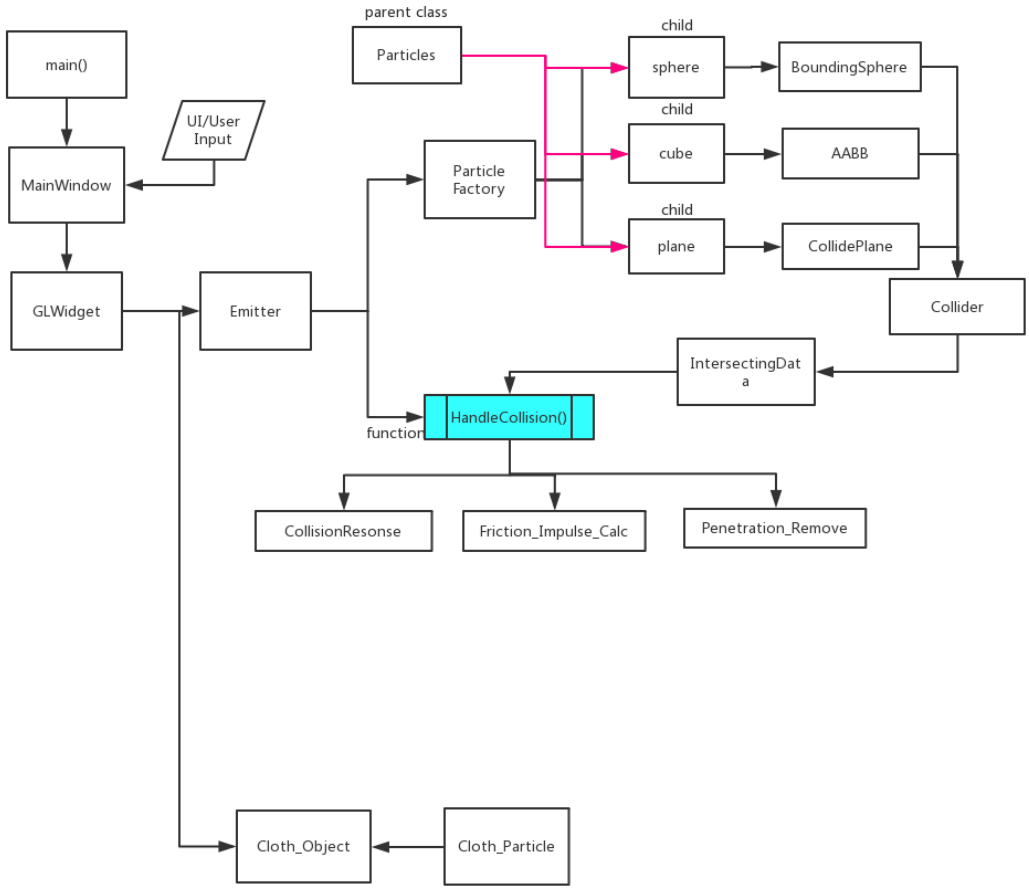
Report

1.Brief

This program implements a simple rigid-body physics engine with a demonstrating particle system. This physics engine is impulse-based, with the ability to calculate gravity, collision and friction impulses and visualize the impact to particles.

2.Structure Chart

This chart demonstrates the relationship between the classes.



3.Implementation

Particle System

Structure and inheritance

In this program, there are three types of particles: sphere, cube and plane. They are organized by a ParticleFactory object created inside emitter object, which will generate particles according to the type specified.

Cube, sphere and plane are child classes of ‘Particle’, which is a base class containing physical

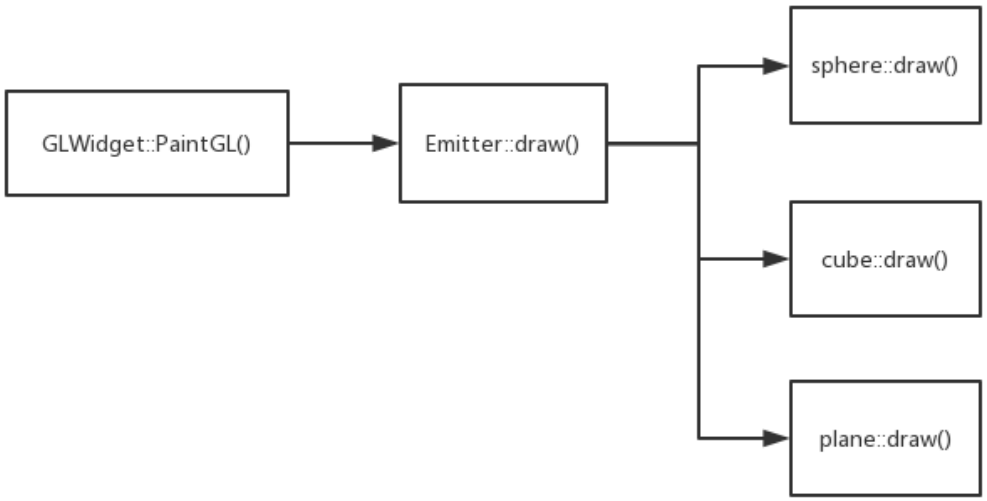
attributes, getters and functions to manipulate status of particles. It also has the virtual drawing and updating functions, which have been defined differently in child classes.

Status updating

In this case, a particle with type “plane” is considered as a ground plane which doesn’t move at all. The moving particles, cube and sphere objects are following the same explicit euler integration method which will be explained in details later.

Particle visualization

Particles are drawn individually by calling *draw()* function from emitter object.



The visualizing set up are completed in GLWidget object before calling the Emitter constructor. When emitter object is created and ready to draw, it receives global mouse transformation and camera information from GLWidget object. This data is accessible from particles created by this emitter. The emitter also passes shader information to its particles.

When draw function is called inside the particle object, firstly, shader from parent object is set to use. Then, a transformation object t takes the position and scale of current particle, sending this data to MVP matrix. Combined with camera and global transformation data from parent, MVP matrix is loaded to the shader to draw the current VAO primitive.

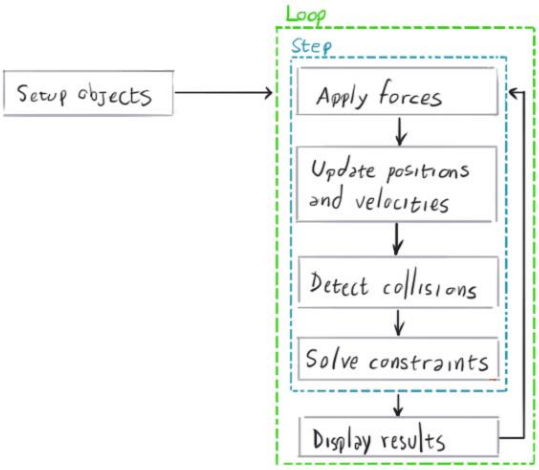
This function is similar for cube and sphere, the only difference is the primitive to draw at the end.

For plane visualization, I created an ngl::simpleIndexVAO object with four vertex and 6 indices to draw the plane as two big triangles.

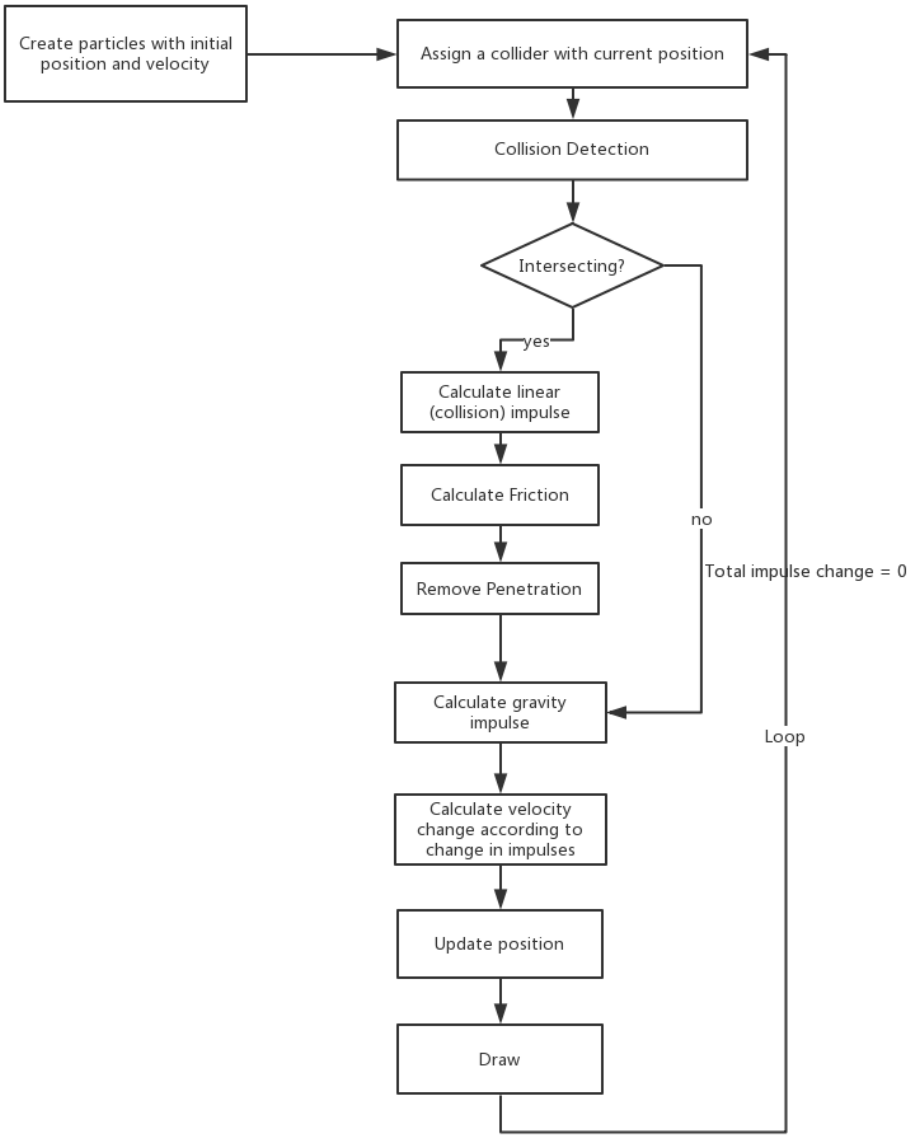
For sphere, a bounding sphere will be created at the run time, and an AABB(axis-aligned bounding box) collider will be assigned to a cube particle.

Physics Algorithm

The following graph I found during my research demonstrates the general logic of a physics engine:



However, I did some modifications to the order of the steps so collision handling has the priority:



In general, during every timer interval, the total momentum change is accumulated and updated for every particles, therefore velocity change can be calculate using:

$$I = mv$$

Position can be calculated by:

$$P' = P + v \times \Delta t$$

According to the graph, this loop follows a basic Explicit Euler Integration, as the position is updated at the end of the loop. From my research, this method is feasible if the acceleration stays constant. In my physics engine for rigid body, gravity is the only force that makes the constant acceleration, which means Explicit Euler Integration can be applied to my simulation loop.

Collision Detection

I have implemented three types of collider - for a sphere, a 'boundingsphere' collider will be used; an 'AABB'(axis-aligned bounding box) collider will be assigned to a cube particle. Ground plane particle has its collider named "CollidePlane".

After collision detection process, an IntersectingDate object is returned, containing the boolean expression indicates whether objects are intersecting, and a penetration vector which is useful in the following penetration removal step.

I made four types of collision: sphere - sphere, sphere - plane, cube - cube and cube - plane.

1. Sphere to sphere collision

The algorithm checks the distance between two sphere center, and compare it with the sum of radius of these two spheres. If distance is less than or equal to radius sum, two spheres are intersecting or perfectly touching; if distance is greater than radius sum, two spheres are not intersecting.

2. Sphere-plane collision

The distance between a sphere bottom and a plane is calculated as the sphere's position vector dot product plane normal, minus its radius. If this distance is less than zero, the sphere is intersecting with plane.

3. Cube-cube collision

This is an AABB intersecting problem. First, I compared distance between maximum and minimum extends of this AABB as well as in reversed order(in case they swap position):

```
IntersectingData AABB::IntersectAABB(const AABB& other)
{
  ngl::Vec3 distance1 = other.m_minExtends - m_maxExtends;
  ngl::Vec3 distance2 = m_minExtends - other.m_maxExtends;
  //get the distances between all three axes
  float x1 = distance1.m_x;
  float x2 = distance2.m_x;

  float y1 = distance1.m_y;
  float y2 = distance2.m_y;

  float z1 = distance1.m_z;
  float z2 = distance2.m_z;
```

Then, if any of these distances is greater than zero, there is no intersecting.

4. Cube-plane collision

This algorithm is the same as sphere-plane intersection. However, instead of subtracting radius from center-to-plane distance, length of half cube edge has been subtracted, as this represent the distance from cube center to its bottom face.

Collision Response(Linear collision impulse)

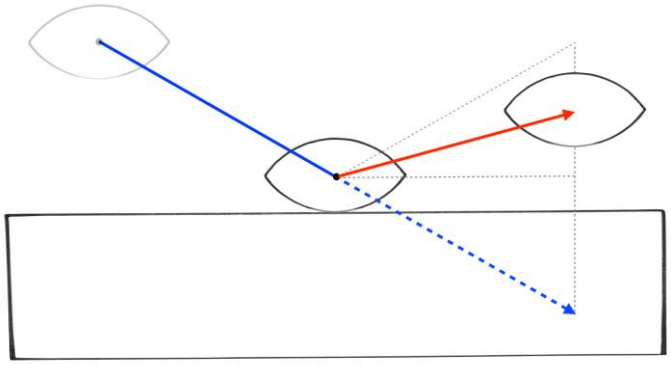
As the physics engine is impulse-based, the data required for collision response is the linear collision momentum impulse along the contact normal. The equation of calculating this momentum is defined as follows:

[latex]j = -(1 + e) \boldsymbol{p} \cdot \boldsymbol{n}[/latex]

Where:

- j is the magnitude of the collision impulse
- e is the coefficient of restitution [0,1]
- p is the linear momentum of the go stone
- n in the contact normal for the collision

By adding the collision momentum to current momentum, the resulting momentum direction lies between perfect bouncing direction and zero bouncing direction, as shown below:



This method works fine, but I found that the particles will never stop bouncing. Therefore, I added custom clamping conditions to avoid infinite bouncing with plane. First, The maximum bouncing time for each particle is 7. If the bouncing count reaches 4, then the collision momentum is forced to decrease 25% each time it bounces off the plane. Then, if the total momentum is beyond a threshold, the particle will not bounce anymore. Instead, it will begin to "slide" with a kinetic friction. This is for faking the rotating effect as this engine doesn't have angular velocity calculation.

Friction Calculation

I have used a fairly simple friction calculation equation as shown below:

$$F = \mu N$$

In terms of impulse, the magnitude of resulted friction impulse should be the magnitude of current momentum along contacting normal multiply the friction coefficient. The direction will be perpendicular to contact normal against the sliding direction.

Penetration Remove

For particles colliding with ground plane, if the penetration does not get moved over time,

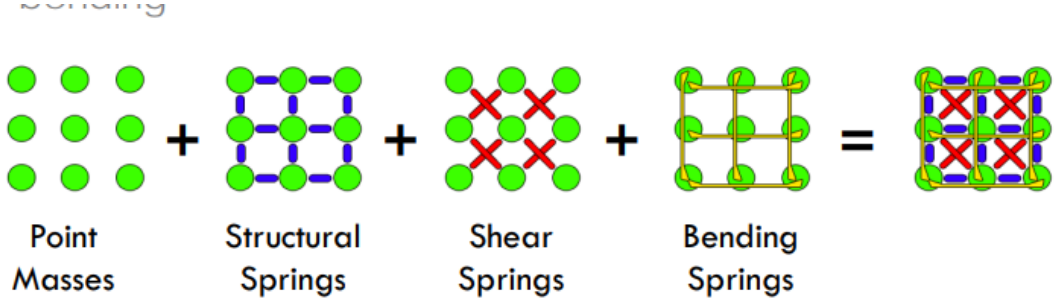
particles will fall through the plane. Therefore, it is necessary to add a penetration removal process at then end of time interval.

According to information received from IntersectingData object, the penetration distance is the length of penetrating vector. So for particle-particle collision, half of the penetration distance is subtracted from both particles; for particle-plane collision, only particles are moved away from plane.

Cloth Simulation

A piece of cloth is represented in this scene by dividing a grid to small triangles and inserting one cloth particle at each vertex. Cloth particles are constrained by invisible springs connecting them.

Below is a classic model for mass-spring cloth simulation:



I was inspired by a tutorial source code by Nick Crook. In his source code, he checked the nearby 24 particles, which gives a realistic behavior of cloth particles. I copied and modified his looping algorithm, as well as a maxim stretching capping function to prevent the cloth being stretched too much.

I build the mass-spring force calculation from scratch by myself. This is done by implementing the following equation:

$$\mathbf{f} = \frac{\mathbf{x}_j - \mathbf{x}_i}{\left|\mathbf{x}_j - \mathbf{x}_i\right|}\left[k_s(\left|\mathbf{x}_j - \mathbf{x}_i\right| - l_o) + k_d(\mathbf{v}_j - \mathbf{v}_i)\cdot\frac{\mathbf{x}_j - \mathbf{x}_i}{\left|\mathbf{x}_j - \mathbf{x}_i\right|}\right]$$

I have also added wind with turbulence to the cloth, which makes it move naturally.

4.Problems and Further Improvement

During my developing and testing I discovered a few problems:

1. Particle Emission. As I have set all particles to emit from one point, the collision response and penetration remove algorithm cannot be applied at the beginning. I managed it by set a releasing

speed, which means the collision algorithms will only work if particles have reached this speed, which means they have jittered a little and will not go crazy. This leads to a problem that the collision between particles will not function properly once particles' speed is lower than releasing speed.

A possible solution to this problem is to jitter particles before simulation starts. Like the 'scatter inside object' algorithms in some 3D software, particles could be placed and separately properly without overlapping each other.

2. Cloth visualization. I have tried to use ngl/simpleIndexVAO class to implement the cloth visualization, but the index problem couldn't be solved. A research in depth will be needed to work out the correct indexing.

3. GUI user input efficiency. In this program, sometimes the user input has to be passed through three or four objects during run-time(e.g: friction coefficient input from Mainwindow.cpp to Particle.cpp). A non- interactive solution could be used to initialize all objects with input values before simulation starts, and user cannot change it before resetting the simulation.

4. Hashing development. In collision detection part, the complexity of searching algorithm follows Big-O notation which will be significantly slow with large amount of particles.

An Octree could be built and tested for this problem.