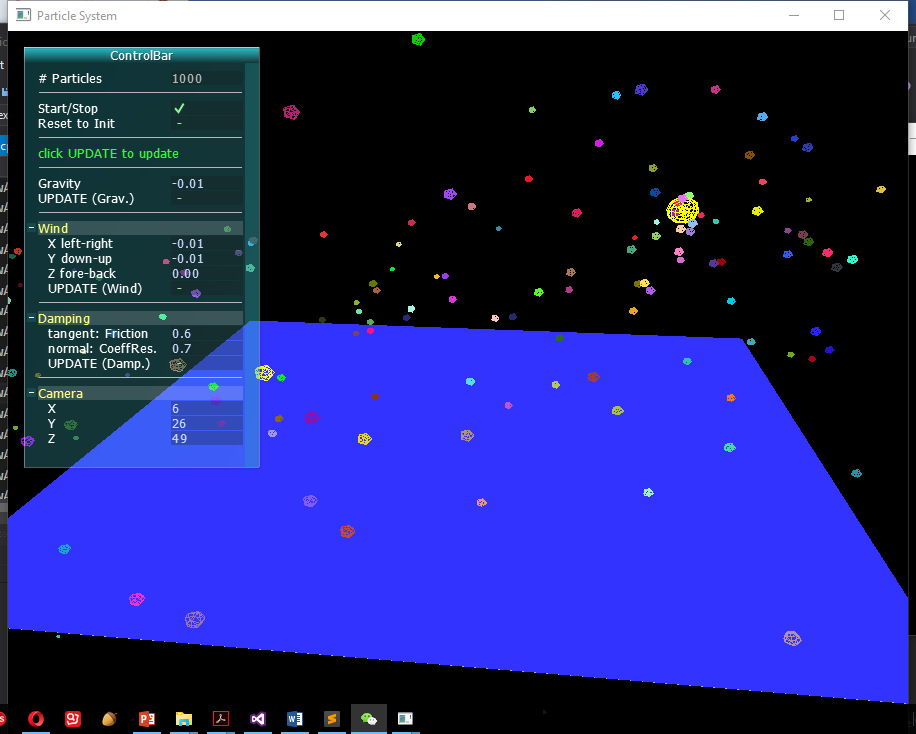
Hao WAN

**Particle System**

****

**•Problem Summary**

The problem I’m dealing with is to implement a particle system, animate about 1000 particles with simple physics (gravity and wind), and bounce off the particles against a ground plane with damping.

Particle simulation is interesting and important because it is relatively simple to create and can derive thousands of interesting visual effects from it - from very basic implementations (like effects of rain, snow, water fountain, fire, smoke, etc.) to some very complex high-level phenomena (like cloth, hair, fluids etc.) These visual effects are widely used in video games, animations, movies, art pieces, etc.

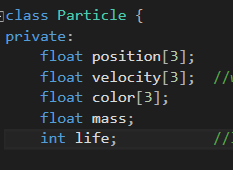
**• Description of work**

**Summary**

A particle system is a collection of point masses that obeys some physical laws. In this project, the physical laws would be the gravity, the wind; and there will be particle-plane collision with damping. We begin with creating a single particle by declare its position, velocity, etc. and how forces should apply on it. Then we construct a particle system with about 1000 particles, defining the methods for creating, updating and drawing the system. To implement the particle-plane collision, we detect the collision first, and then decompose the velocity of particle in the normal and tangent directions of the plane to obtain the new velocity after collision. We use a linked list as the data structure to store the particle system. We also implement a UI bar by using the AntTweakBar library so that the users can change some parameters and observe different visual effects of the particle system.

**Particle Class**

Every particle is defined by its position, velocity, color, mass and a life time. To simplify the problem, we assume that a particle has no rotation on itself and it doesn’t collide with other particles.



The velocity is stored in an array for its x, y and z components. The velocity contains magnitude and direction.

Next, we create a constructor of the Particle class with the above characterizations.

Then some useful functions applied on a single particle are defined, such as: get/set position, get velocity, get color, get/update life time. These functions will be used later for updating or drawing the system in the ParticleSystem class.

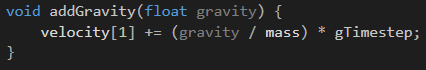
We define how to apply the forces on every particle in this class, too.

For a force taking an acceleration on a particle, we have force = mass \* acceleration. So, the acceleration = force / mass. To get the new velocity we can write velocity += (force / mass) \* timestep.

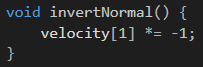
According to the basic physical laws and the integration above, we have the following functions in the Particle Class:

To apply gravity:

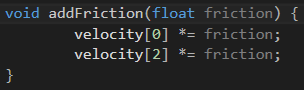
Only the velocity in Y direction affected, because gravity is downward.



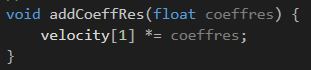
To invert the velocity in Y direction:



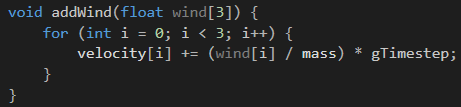
To apply friction factor in tangent directions of the plane (X and Z directions):



To apply coefficient of restitution in normal direction of the plane (Y direction):



To apply wind (in all directions):



**ParticleSystem Class**

In this class, a data structure is defined to store the particles in the system. A doubly linked list is used. The element in the node of the list is a pointer that points to an existing particle.

The constructor of the class is defined by the forces applied to the system (gravity, wind and collision damping factors), and the position of the particles.

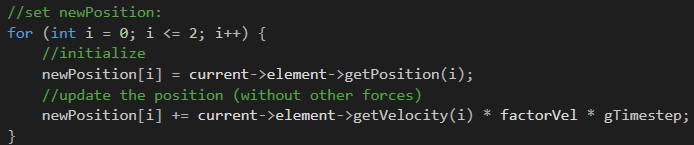
We also construct the functions of creating, drawing and updating every particle in this class:

C:\Users\WAN\AppData\Local\Temp\1512271993(1).png **-** For every update, createParticle () will create one particle, and add it to the end of the linked list.

**C:\Users\WAN\AppData\Local\Temp\1512272009(1).png -**For every update, drawParticles () will iterate through the whole list and draw all the particles in the list.

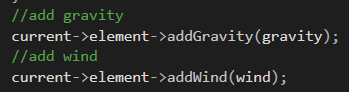
**C:\Users\WAN\AppData\Local\Temp\1512272059(1).png -**For every update, updateParticles() will iterate through the whole list and set new position for all the particles in the list using the physical laws. It will get rid of the dead particles from the list, too.

The following is how the updateParticles() method updates the new position:



We have a factor of velocity to control the magnitude of the velocity. Multiplying gTimestep to velocity will get the value of a difference in position during the timestep. And then we add this difference to the old position to obtain a new one.

With that, we add gravity and wind to the system:

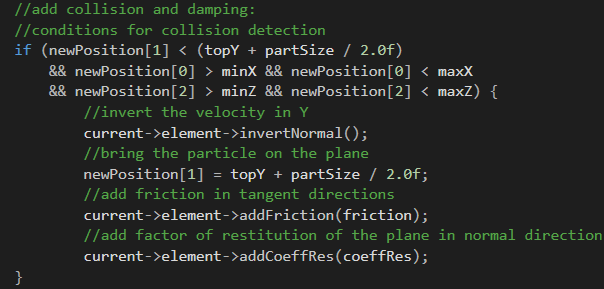


We already define how to apply gravity and wind to a single particle in the Particle class. So, here we simply call those functions and pass the arguments.

Right after the above code, a collision detection is implemented.

**Collision Handling**

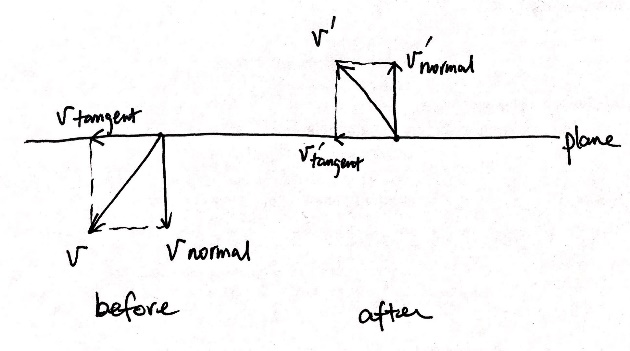
In our case, the particle vs. plane collision detection is very easy to realize. In the updateParticles() method, right after the above code, we have:



We simply check the position of the current particle. If the particle comes into the X and Z range of the plane, and in the Y direction it is below the plane, it will have a collision with the plane.

In this case, we decompose the velocity into two components: v = v(tangent) + v(normal).

After the collision, we have v’(tangent) = friction factor \* v(tangent) and v’(normal) = - coefficient of restitution \* v(normal). The velocity after collision v’ will be v’ = v’(tangent) + v’(normal):



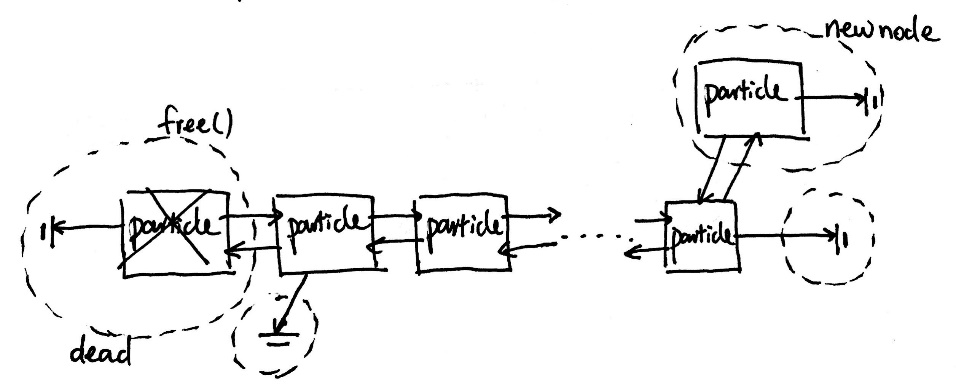
The related functions are invertNormal(), addFriction(), and addCoeffRes(). They are all defined previously in the Particle class. Here we simply call them and pass the corresponding arguments.

The friction factor “friction” and the coefficient of restitution “coeffRes” are both between 0 to 1. The friction factor being 1 means the plane has no friction, there is no decrease for the velocity in the tangent direction(s). The coeffRes factor being 1 means the plane is perfectly elastic, there is no consumption of the velocity in the normal. These two factors can be controlled as the users would like in the UI bar.

**Data Structure to store the particles**

We use a doubly linked list to store all the particles. A linked list is very convenient to use in our case.

Every update, we add one new node at the end of the list as the newly created particle. We also check if a particle’s life time becomes negative. If the life time is smaller than 0, it will be treated as dead and will be removed. The corresponding memory will be freed, too. Because we add at the end, the oldest ones will always appear at the beginning of the list.



In our case, in the create and update function, we only add one new particle, and the life time of a particle only decrements by one. Then we put the create and update methods in a same timer function, i.e. we have a same frequency of creating and removing particle. After we click “start”, for the first update, we generate the first particle; for the second update, we generate the second particle; and so on. After 1000th updates, we begin to remove a dead one because we set the initial life time to 1000. After that, for every update, we create one particle and at the same time we remove one particle, the system will always keep 1000 particles in it.

For drawing and updating the positions of the particle, we just iterate through the whole linked list during every update.

The update timer is set to 10 milliseconds, so every 10 milliseconds we’ll have an update.

**Implementation of UI**



There is a user interface bar using AntTweakBar library at the top left of the window to enable the interaction, as the figure shows above. Through this UI, users can start or stop the animation, reset the parameters to the initial state, apply new gravity or collision damping values, apply and change wind to the system, and change the camera view. It’s a simple UI but provides the basic control of the particle system.

• **Analysis of work**

The project meets the goals in the proposal very well: I successfully build a particle system with gravity and wind, and let the particles bounce off the plane with damping factors. The simulation works well. On the basis above, I add a user interface bar allowing users to manipulate the parameters.

The deficiency point would be that in this project I use the old OpenGL. When making the proposal of the project I had considered to learn and try the modern OpenGL. However, the remaining time was limited and I get used to the old OpenGL through the assignments during this semester. This could have been a good chance for me to learn the modern OpenGL. So, the future work might be to rebuild the system using modern OpenGL and add some textures to the particles.

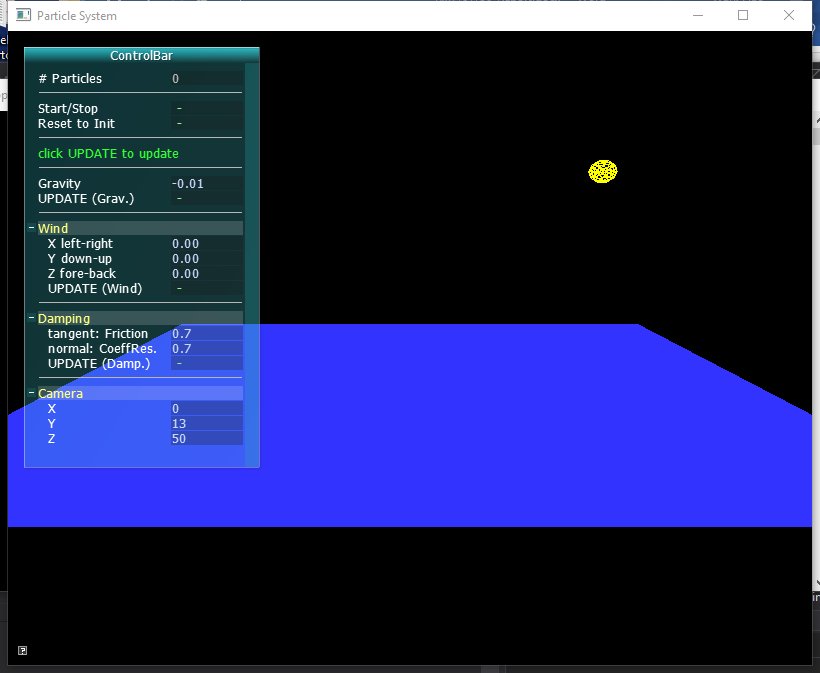
Another point is about the choice of data structure to store the particles. I simply use one linked list. For this particle system, it is simple and convenient to use. I was very happy and satisfied with it. However, when I came across several threads online talking about data structure for a particle system, I realized that there exist multiple choices. A list for a particle system may not be the best. But the choice really depends on the circumstances. In the future, I would like to continue to think about it and look up information on this data structure topic.

• **Results and Conclusion**

The following are the snapshots of the animation window:

**1)** The scene without particles:

We have a blue ground plane. Above the plane there is a yellow ball as the source of the particle.



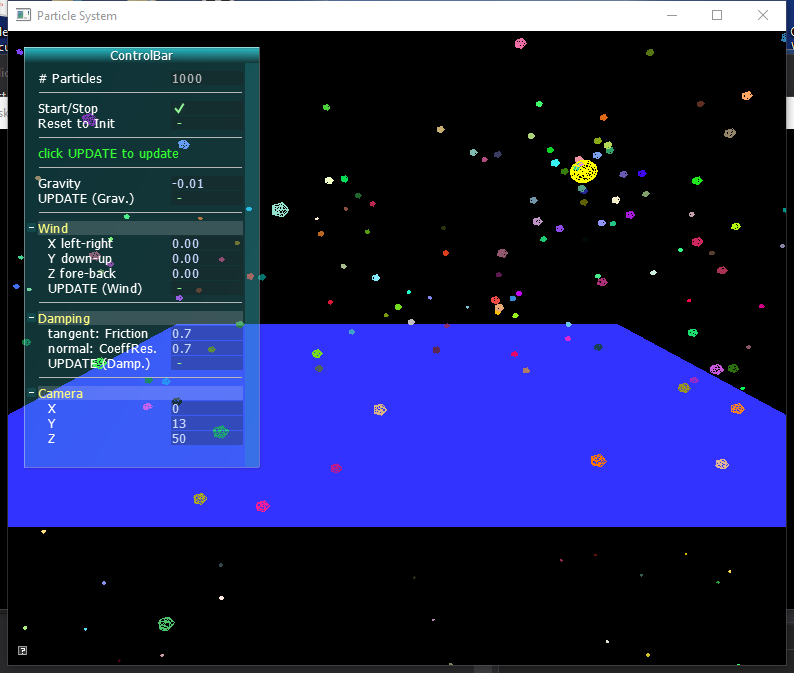
**2)** After user clicks “Start/Stop”:

The particles have random velocities and colors. The source starts to emit particles of different colors in all the directions. After several seconds, the number of particles in the system will reach 1000 and stay unchanged.

The initial scene with the particles will show an environment with a gravity of 0.01, without wind, and with collision damping factor of 0.7 in both normal and tangent directions.

We can see the particles tends to fall. Some will hit the ground, and bounce off with a damping factor. After each collision with the plane, the velocities in the normal and tangent directions will be reduced. Some particles will stop on top of the plane after multiple collisions with the plane.

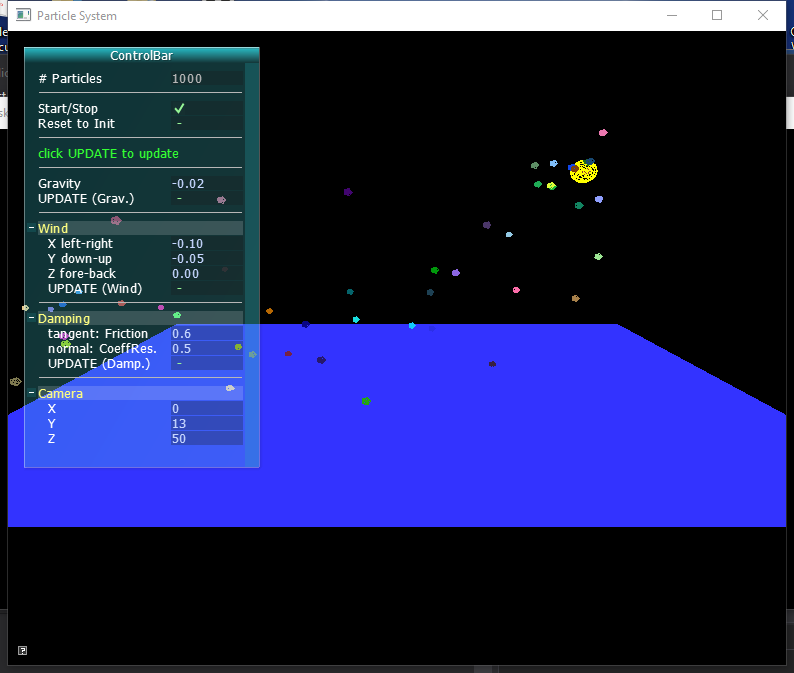
All the particle will be forced to disappear after a period of time.



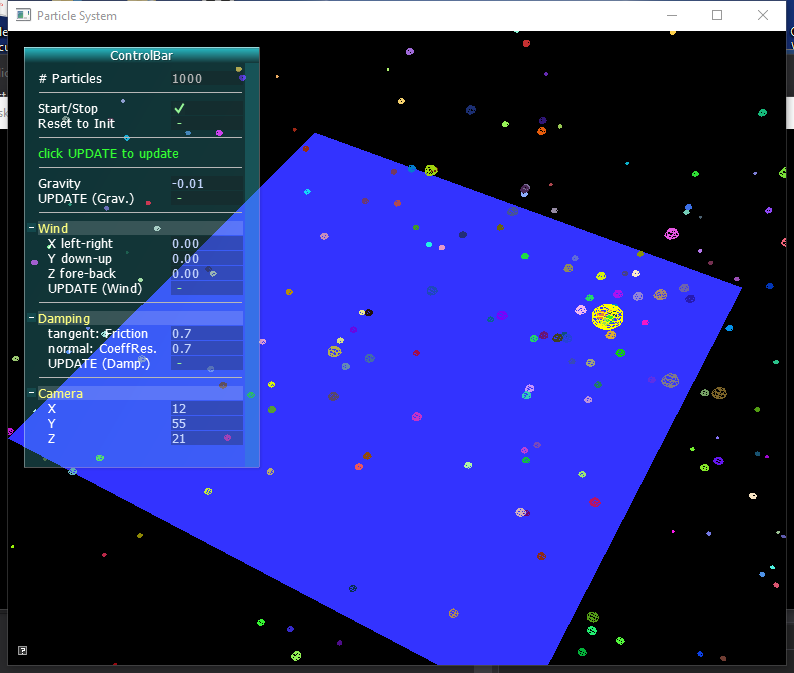
**3)** The following snapshot shows a system with a gravity of 0.02 and a wind in X and Y direction. The friction is set to 0.6 and the coefficient of restitution is set to 0.5.

Users can set new values through the control bar to observe different effects of the particles under different parameters. Users can apply inverse gravity (particles tend to fly upwards), void environment (no gravity, particles keep their initial velocities), wind in different directions and magnitudes, a plane with no friction or very huge friction, a perfectly elastic or inelastic plane, etc.

Users can also reset to the initial parameters by clicking “Reset to Init”.



**4)** Users can set the camera position to view the particle effects from different positions:



**Some observations**:

* Once a particle system is built, we can easily manipulate the physical parameters to obtain many interesting simulations.
* The particle system is built on physical laws. To successfully build a particle system we should understand the mechanisms and correctly apply the physical laws to every single particle, so that the simulation will obey the real physical world and seem reasonable.