The real-time simulation of natural phenomena --A general particle system built for Wonderland

2010-01-21

Supervisor
Yanwen guo(郭延文)   ywguo@nju.edu.cn

Team member
Qing Da (笪庆)   csdaqing@gmail.com
Jia Xu (许佳)   xujianjucs@gmail.com
Fanjiang Zeng (曾繁江) zeng@smail.nju.edu.cn

All codes and documents can be found at our home page http://wlandparticles.sourceforge.net/

daqing
daqing
daqing
daqing
daqing
daqing

Contents

1 Introduction

2 Overview

3 Implementation

4 Application

5 Tutorial

6 Acknowledgement

Introduction

- Real-time simulation of natural phenomena, such as clouds, snowing, or raining, in a virtual environment has always been a hotspot in computer graphics and virtual reality.

- It is necessary, and to some extent significant, to explore the realtime realistic simulation of these natural phenomena in the Wonderland virtual world

Introduction

- In our early research on this project, we found many natural phenomena could be simulated using particle. To make full use of the method, we focus on building a general particle system in the Wonderland, apart from implementing several phenomena described in our proposal.

Introduction

- We successfully simulate snow and rain scene using our particle system, Furthermore, we have used our particle system to build many other fantastic phenomena like fountain, fire, water vapor and halo.

- Some of them can be seen at the beginning of this slides.

Introduction

- After several attempts, we terminate the simulation of cloud for the following reasons:

- Simulation of cloud is very difficult;
- Wonderland already has a sky box in its scene
- The avatar in Wonderland cannot raise her/his head, so what s/he can see from the sky is a very limited area.

Overview

- Particle system is a technique to simulate certain fuzzy phenomena, which are otherwise very hard to reproduce with conventional rendering techniques. Examples of such phenomena which are commonly replicated using particle systems include fire, explosions, smoke, moving water, snow, dust, grass, or abstract visual effects like glowing trails, magic spells, etc.

- More about particle system can be found in our detail technical report.

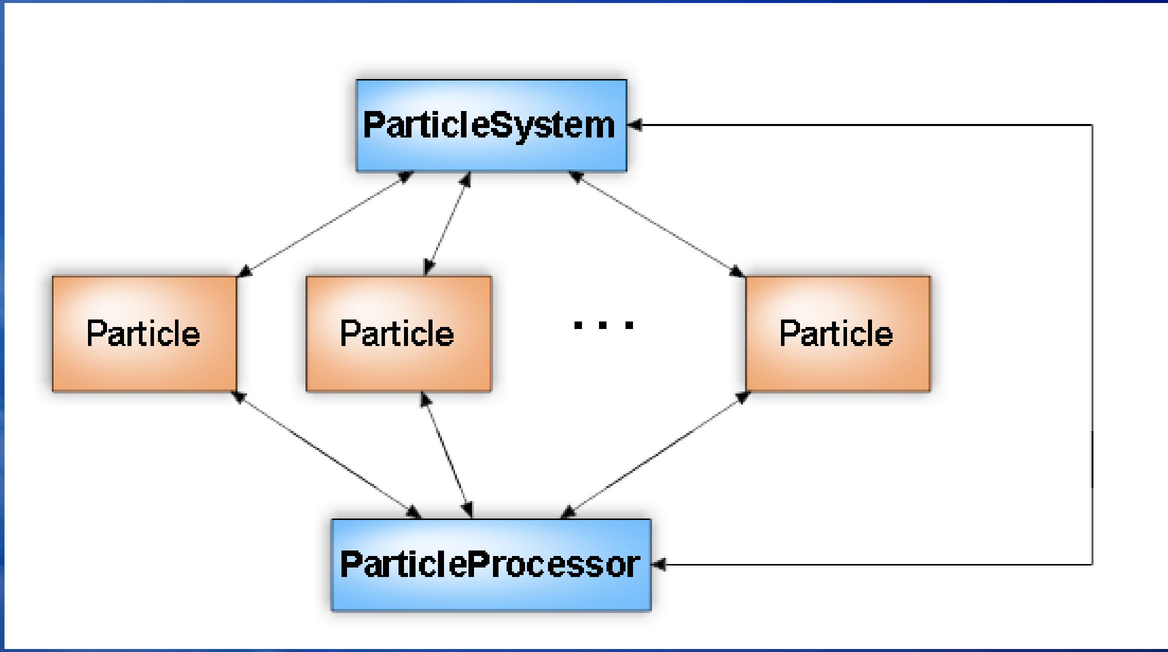
Implementation

- In this section, we will give a detailed technical description of our particle system.

- In the following parts, we will give a brief introduction to the idea of our designment and basic components of our particle system. If you want to learn more, please refer to the Tutorial section, and try some applications

Implementation

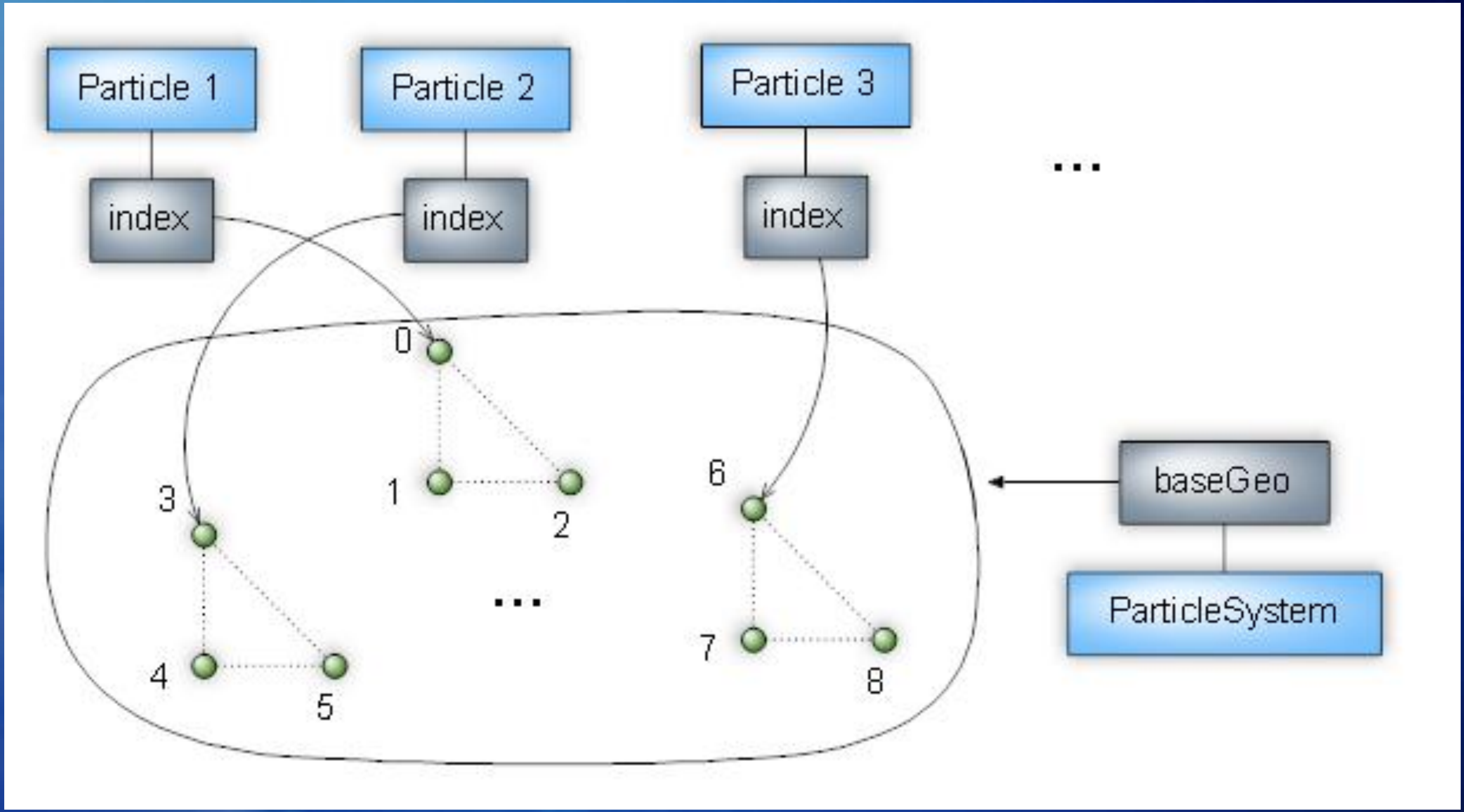
- There are three basic classes in our particle system: ***Particle***, ***ParticleSystem*** and ***ParticleProcessor***



Implementation

- In our designment, the geometric information of a particle, like the vertex buffer or texture coordinate buffer which will be directly used to display those particles in the screen, is not stored in the Particle itself, but the ParticleSystem.

Implementation



Implementation

- In the ParticleSystem, there are following basic parameters to control the whole system:
-
- isRotateWithView: the flag to determine whether to use bill-boarded technology
- numberOfParticles: the whole number of particles
- releaseRate: in every frame, ratio of new particles' number to the number of all particles
- minVelocity: the minimum velocity of a particle
- ….

Implementation

- At the beginning, the whole number of all the particles is initialized as soon as the class PaticleSystem is created. All the particles are set to the ready-state at first. In every frame, the number of new particles is calculated by

$$num_{new} = num_{all} \times releaseRate$$

Implementation

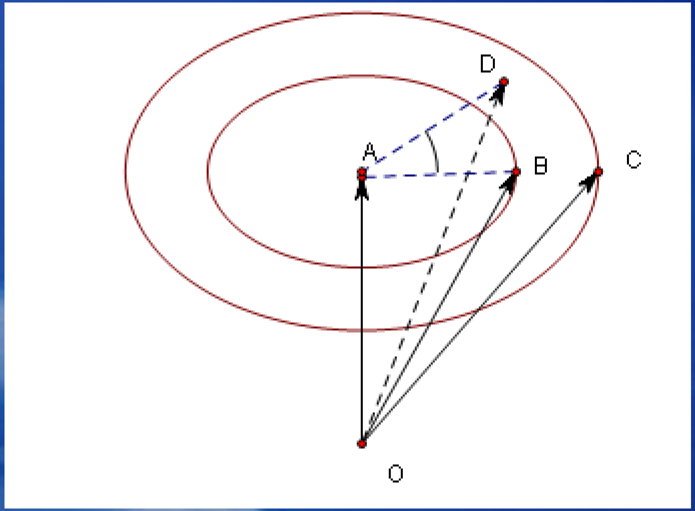
- When a particle is created by the particle system, we should assign the particle basic properties like velocity, size, and direction. Here the velocity is a scalar and the direction is a normalized vector. We update the velocity and size in the following way:

$$size_{new} = size_{\min} + rand(0,1) \times (size_{\max} - size_{\min})$$

$$velocity_{new} = velocity_{\min} + rand(0,1) \times (velocity_{\max} - velocity_{\min})$$

Implementation

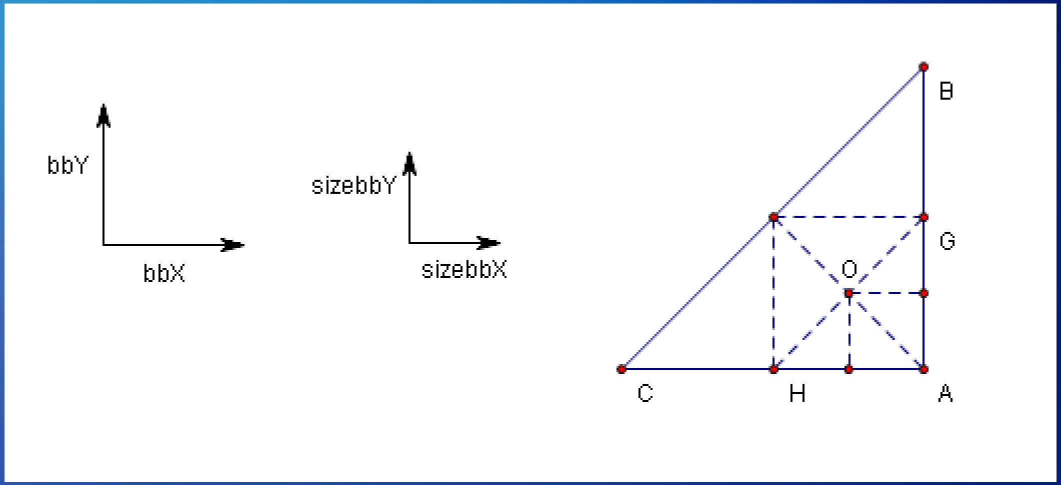
- To make the particle more realistic, we should make the particle change its direction slightly and rapidly. When generating a new direction, the key point is how to give a random direction, which cannot be far away from the main direction.



Implementation

- The next point we would like to make is how to build a particle’s geometric structure, including the location and texture coordinates of every vertex.
- Firstly we will introduce two vectors, bbX and bbY. These two vectors, which are orthogonal, determine a flat which the triangles of particles will belong to.

Implementation



- O is the position of the particle. For a particle p, assume its size is , then

$$sizebbX = size_p \times bbX$$

$$sizebbY = size_p \times bbY$$

Implementation

- Notice that the minimum index is stored in Particle, say, ,then the location of the three vertexes are defined as

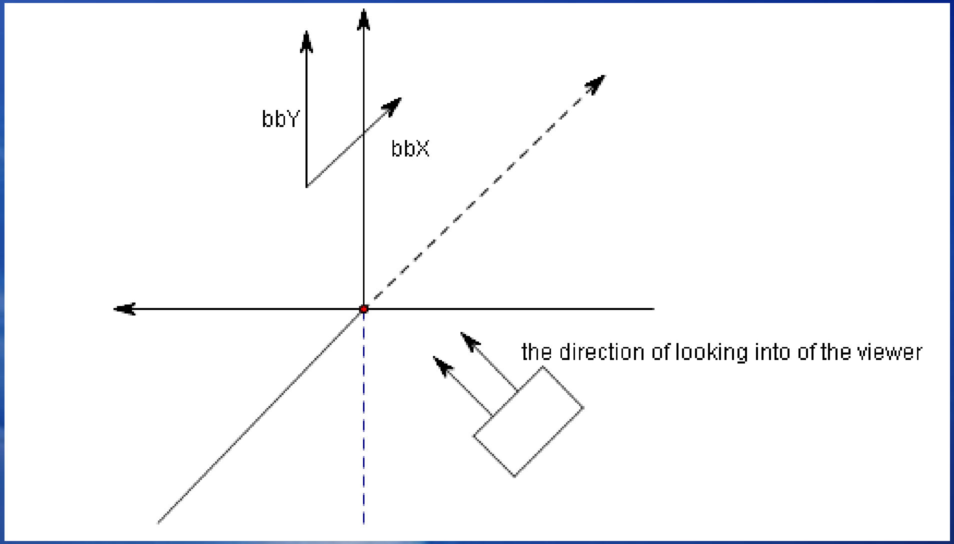
$$location_A = location_{index_p} = O + sizebbX - sizebbY$$

$$location_B = location_{index_p+1} = O + sizebbX + 3sizebbY$$

$$location_C = location_{index_p+2} = O - 3sizebbX - sizebbY$$

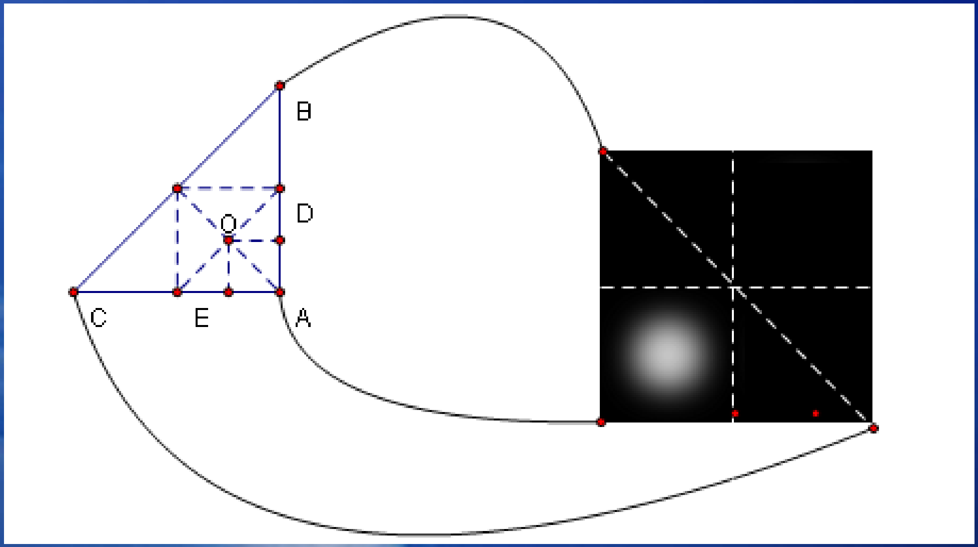
Implementation

- The two vectors, bbX and bbY play an important role in adjusting the viewer. When the avatar is walking around in the Wonderland, we should make sure that all the particles will face to the viewer, then we just update bbX



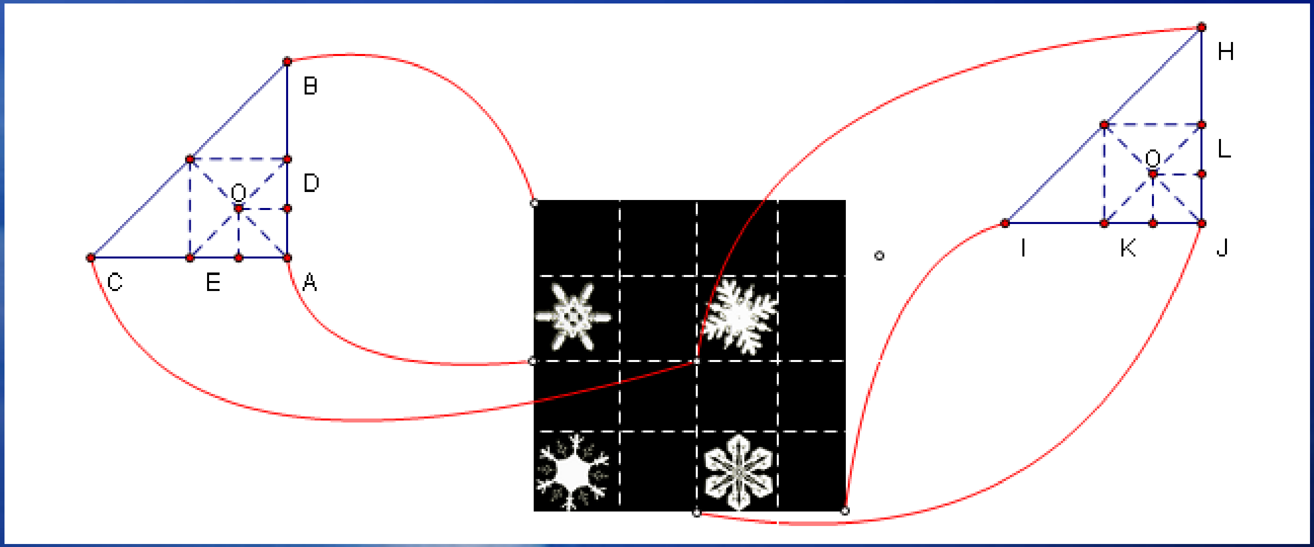
Implementation

- we map the three vertexes of the triangle, which represents the particle, to a specified picture. Then we need to compute the texture coordinates of the three vertexes and assign the texture image positions to vertices.



Implementation

- It is necessary to allow the particle to have different textures so that we could get a better expressiveness. The texture image can be a matrix of the particular pictures. Every particle can randomly choose one as its texture.



Implementation

- The last one of three main classes is ParticleProcessor. ParticleProcessor is a subclass of ProcessorComponent.
- We override two methods in the ParticleProcessor: compute() and commit().
- The compute() method is used to do all the calculation that is needed when updating particles , and the commit() method is used to update jME objects in an MT-safe manner.

Implementation

ParticleLocator, ***ParticlesInitializer***, ***ParticleConstraint*** and ***ParticleEffect***

- To extend our particle system, we design several classes to make this system more universal. One of them is ***ParticleLocator***, which works as an emitter of particle system, providing a variety of emission areas.

Implementation

ParticleLocator, ***ParticlesInitializer***, ***ParticleConstraint*** and ***ParticleEffect***

- ***ParticlesInitializer*** will give an opportunity to personalize the initialization.

Implementation

ParticleLocator, ***ParticlesInitializer***, ***ParticleConstraint*** and ***ParticleEffect***

- In most particle systems, a particle will be killed when its lifetime is over. But there are other situations in which a particle will die, when some of its properties do not satisfy some constraints any more; and lifetime is just a special case. The class ***ParticleConstraint*** is used to describe this limitation.

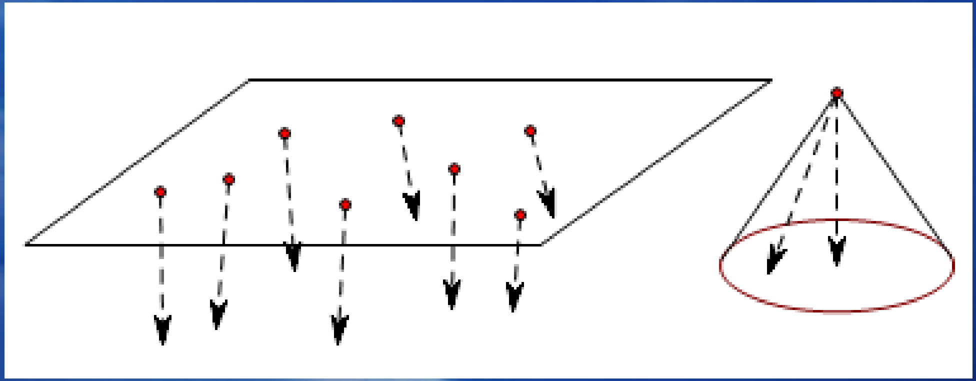
Application

- In this section, we will use our particle system to simulate several phenomena, like snow, rain, fountain, fire, water vapor and halo
- Our applications are implemented in the form of modules in Wonderland

Application

Snow

- The snow is the easiest phenomenon to simulate using particle system. We use a rectangle overhead as emission area. Particles are created from this rectangle, then falling down with a little deflection angle. The texture image is a fuzzy white point.



Application

Snow

- In the physical world, the snow particles do not move in a constant velocity, as a result of the wind and the collision of other snow particles. In our snow simulation, we do not consider the influence of particles’ collision. We would like to simulate gust, a sudden blast of wind occasionally.

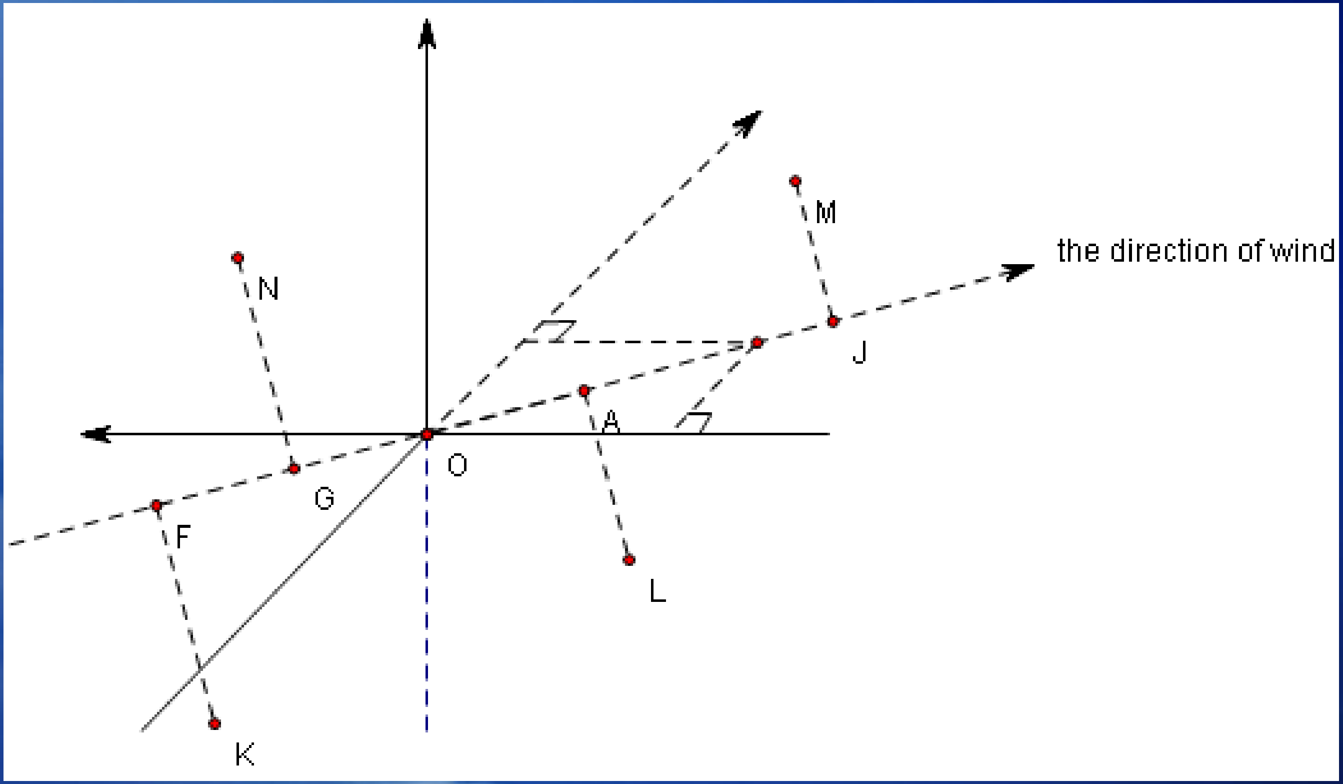
Application

Snow

- When a gust of wind comes, those particles in the position where the wind comes from will be affected firstly, and those particles in the position that the wind goes to will be the last ones to be affected. In fact, the former particles will have a higher velocity while the latter particles will have a lower one. Next figure shows how to determine which particles are in the location the wind comes from or goes to.

Application

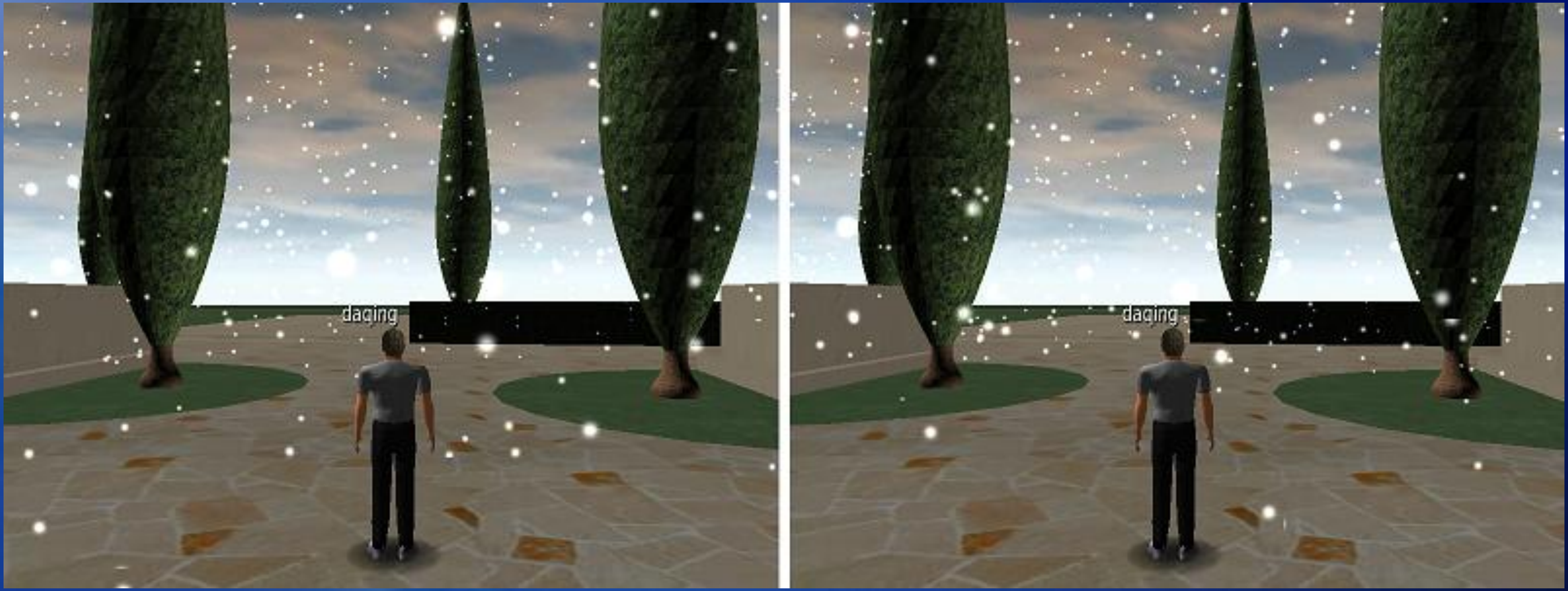
Snow



Application

Snow

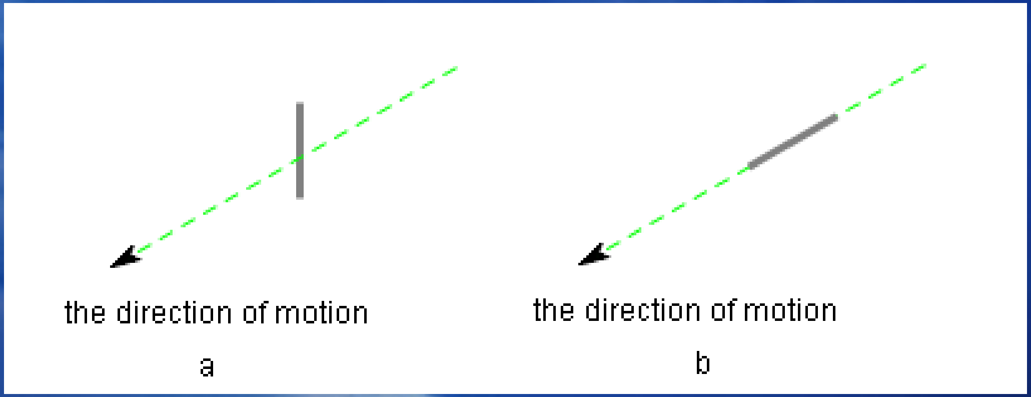
- result



Application

Rain

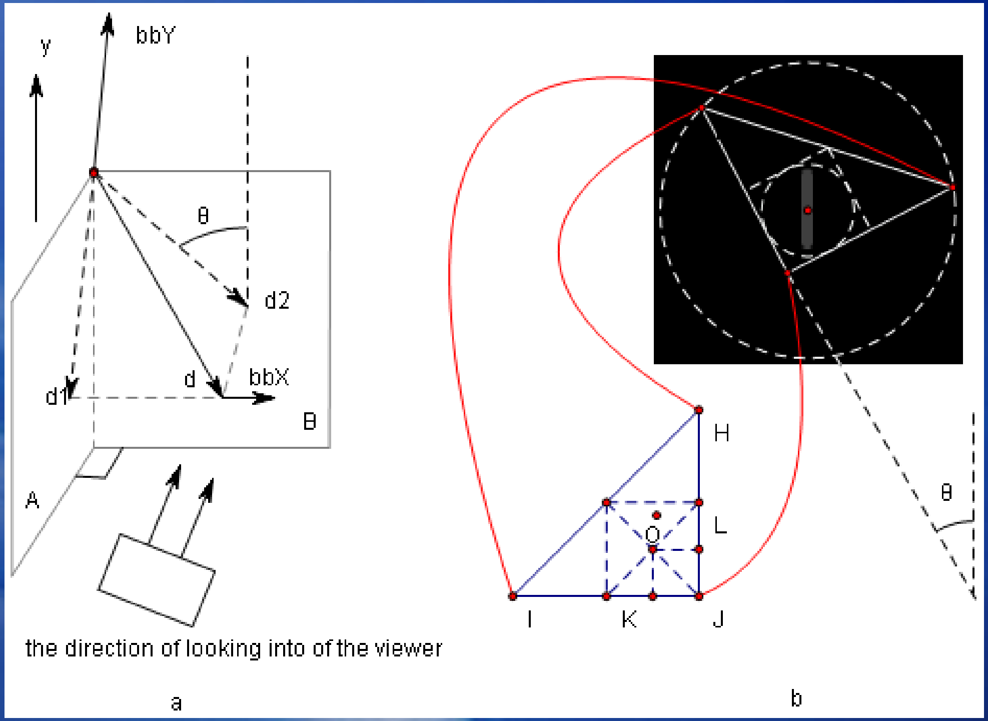
- The initialization of rain is the same as the one of snow, but with a texture image with a gray line. In the default setting of our particle system, the texture of a particle will be presented as a, but what we exactly need is as shown in b



Application

Rain

- This can be solved by updating bbY of the particle system and modifying the texture coordinates.



Application

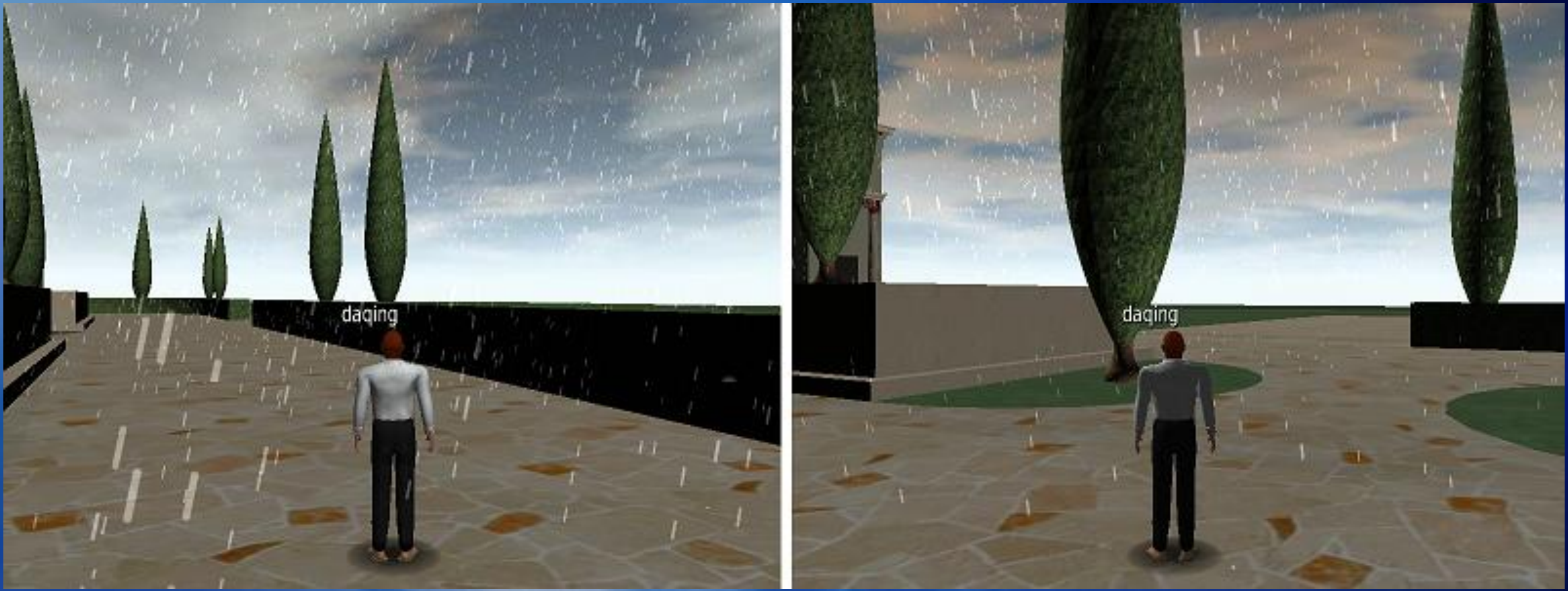
Rain

- It is a little complicated, you may get more details about it In our report. (The detail implementation can be found in *ParticleLineEffect*.)

- Here is the result.

Application

Rain



Application

Fountain

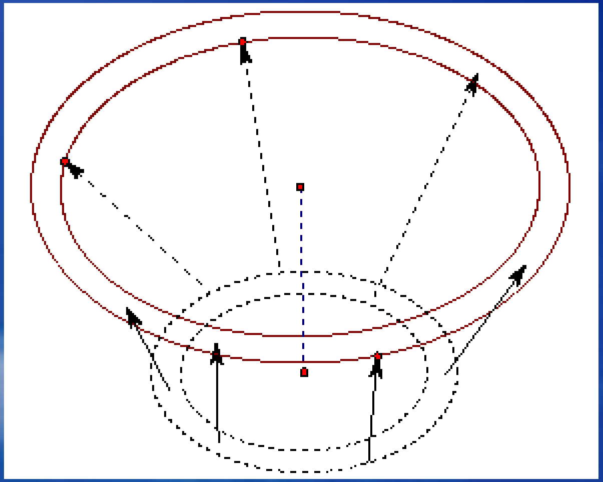
- To make a better use of *ParticleLineEffect*, we found another phenomenon made up of water particles can also be simulated using this technology.

- It is the fountain

Application

Fountain

- In the fountain simulation, all the water particles will be emitted from a ring area with a upward direction, slopping inside out



Application

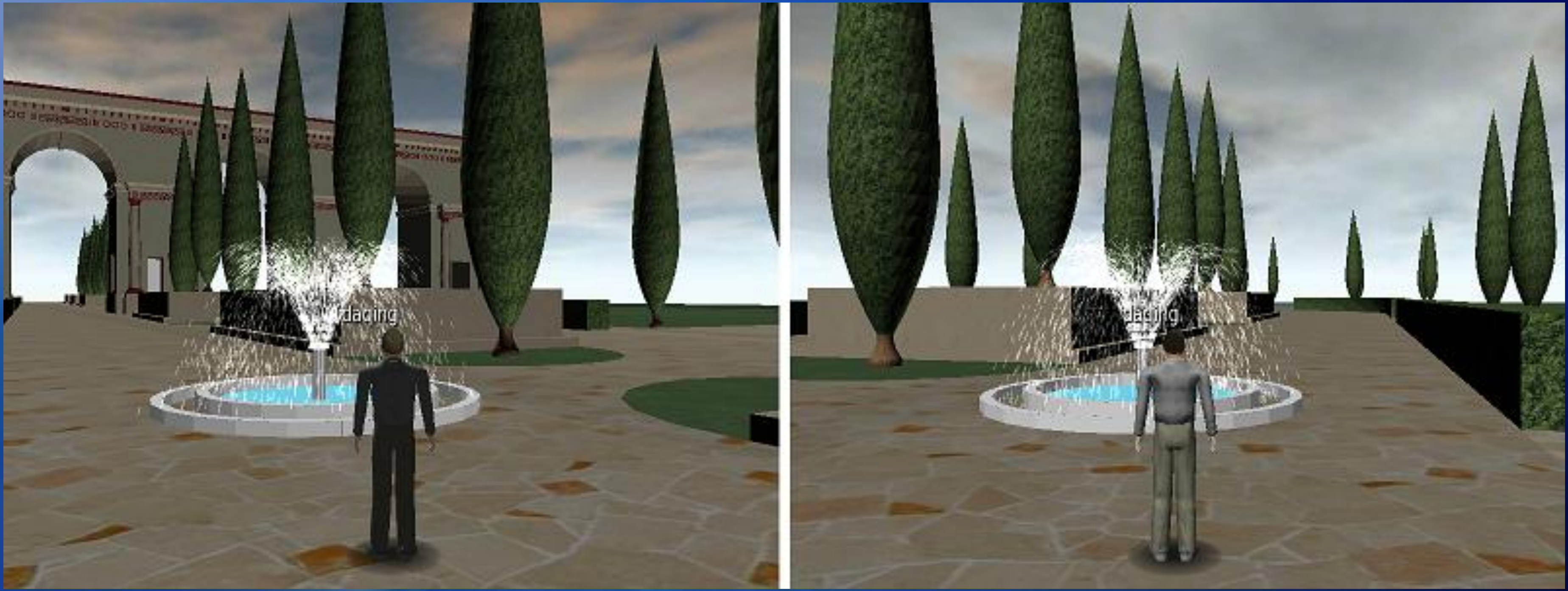
Fountain

- Hence, we have developed *ParticleAccelerationEffect*, which will provide a way to put acceleration of the particle in any direction.

Application

Fountain

- result



Application

Water vapor

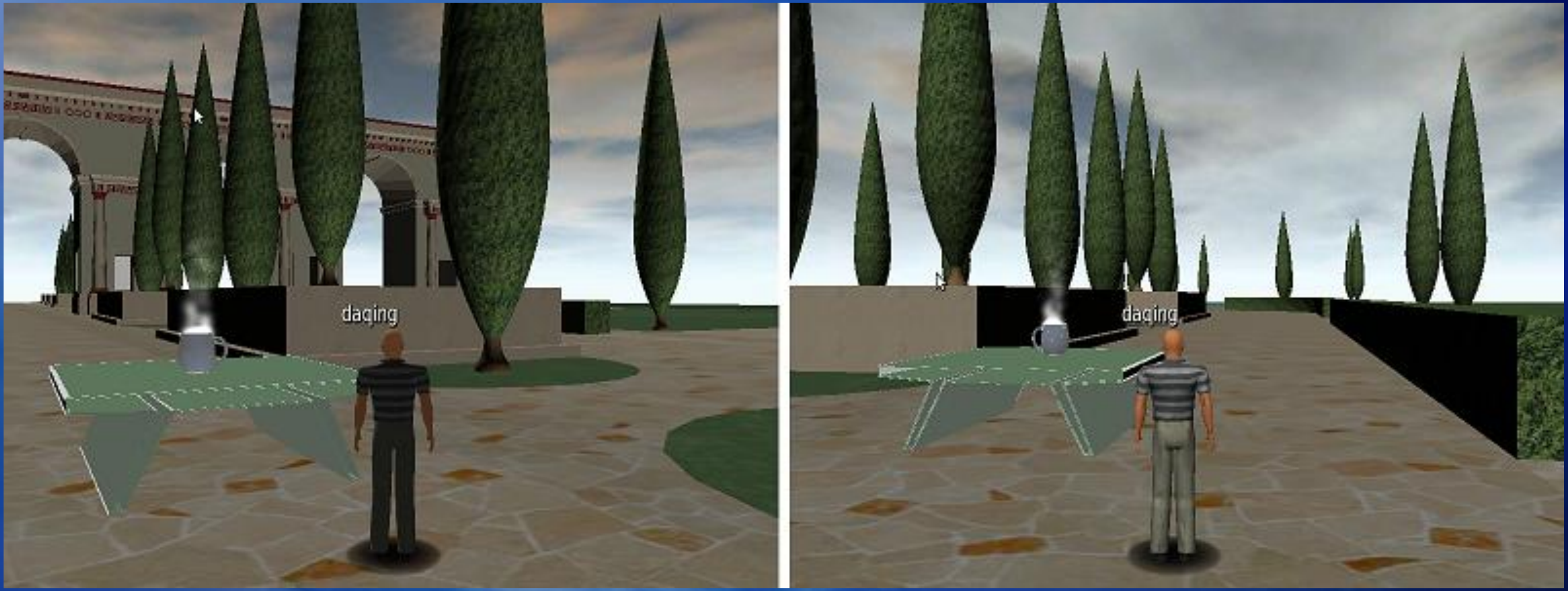
- Sometimes we want the transparency decreases to zero when its lifetime is over, like simulating water vapor of a cup of coffee. So we have developed the *ParticleFadeEffect.*

- This can be used to simulate the water vapor.

Application

Water vapor

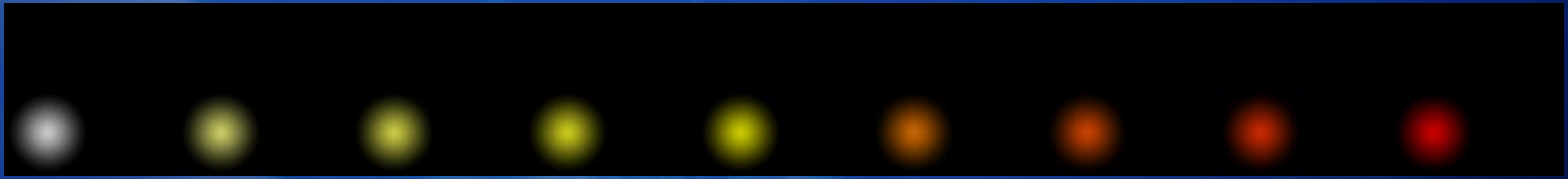
- result



Application

Fire

- Fire is another common natural phenomenon which can be simulated well using our particle system.
- As a matter of fact, the inner part of fire has a higher temperature, which makes it seem like white, while in the outer part, the temperature is relatively lower, so it will turn red



Application

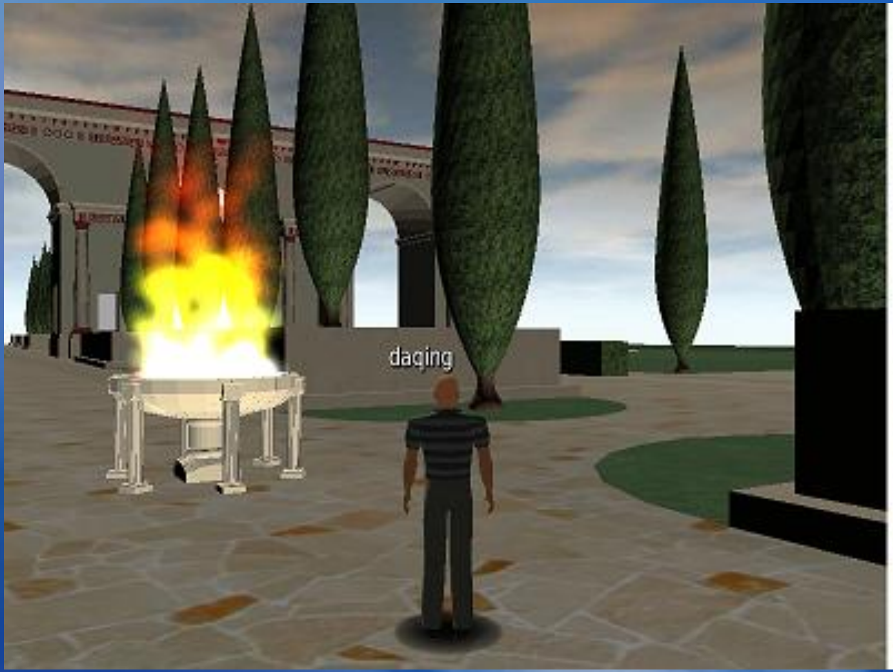
Fire

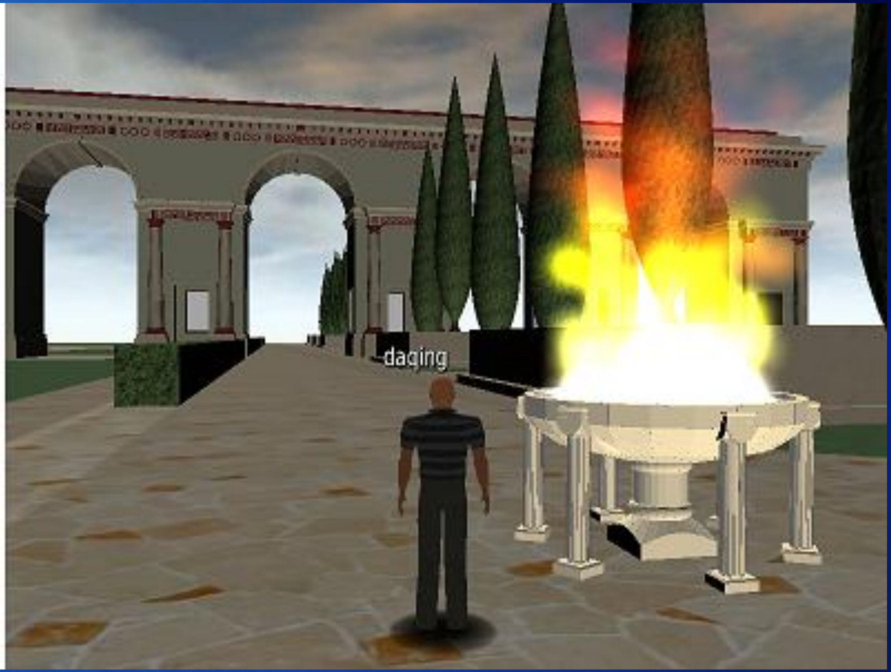
- This effect is implemented in ***ParticleFireEffect***. To make the fire more real, we have added fade effect and acceleration effect to the fire

- Here is the result

Application

Fire





Application

Halo

- A halo is an optical phenomenon produced by ice crystals creating colored or white arcs and spots in the sky. Many are near the sun or moon but others are elsewhere and even in the opposite part of the sky.

Application

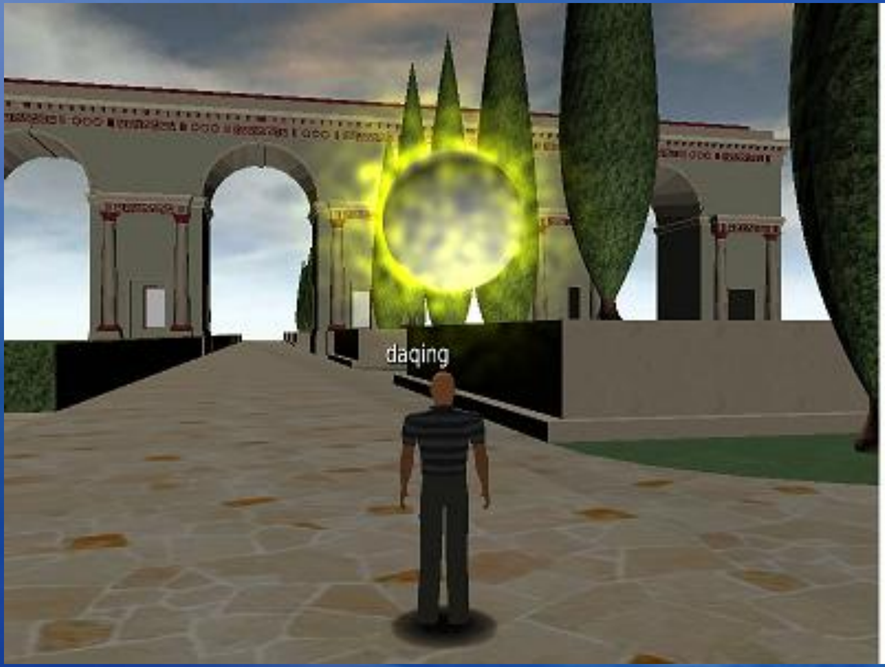
Halo

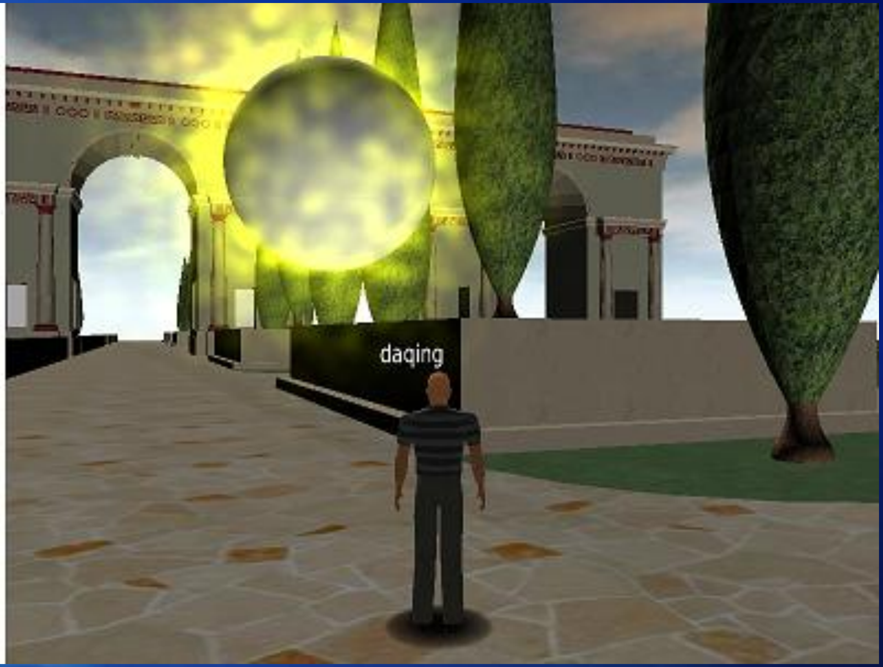
- We want to simulate the halo phenomenon by using the mesh object as emission area, and setting the velocity of every particle to be normal to the individual face of the mesh object through developing a *ParticlesNormInitializer*. In this case, a fade effect is also needed.

Application

Halo

- result





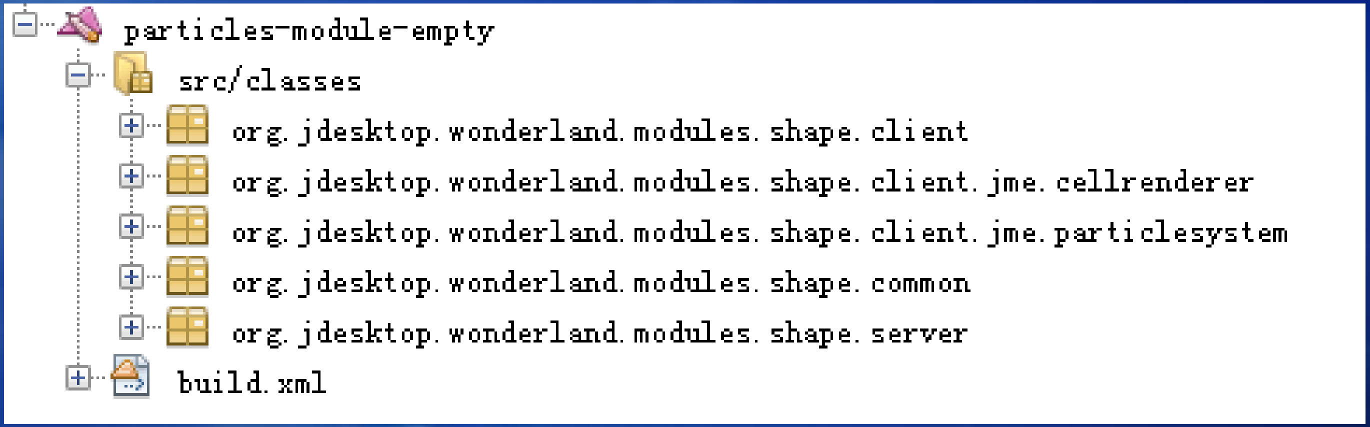
Tutorial

- In this section, you will learn how to use our particle system to build your phenomena. Since all the above applications are implemented in the form of modules of wonderland, you are supposed to be familiar with the module mechanism of wonderland. Here we will show you how to do your own application in a standard way.

Tutorial

- Firstly, you should visit our project home page and download particles-module-empty.tar.gz.

- Open this project in NetBeans, you will see the structure of this module as shown



```
particles-module-empty
  src/classes
    org.jdesktop.wonderland.modules.shape.client
    org.jdesktop.wonderland.modules.shape.client.jme.cellrenderer
    org.jdesktop.wonderland.modules.shape.client.jme.particlesystem
    org.jdesktop.wonderland.modules.shape.common
    org.jdesktop.wonderland.modules.shape.server
  build.xml
```

Tutorial

- Now we will use this to build a rotary snowflake phenomenon.

- 1.Rename the directory to *particles-module-snowflake*

- 2.Rename the name of project to *particles-module-snowflake* in NetBeans.

Tutorial

- 3. Modify the module name to *particles-module-snowflake*

- 4. Modify the getDisplayName() method in ParticlesCellFactory as

```
public String getDisplayName() {
    return "Particles Cell Snowflake";
}
```

Tutorial

- 5. In this case, we will use a texture image of snowflake which follows the rule we describe before. Add this file (snowflake.png) to the directory
- /particles-module-snowflake/art.

Tutorial

- 6. Modify the getDefaultCellServerState() method

```
public <T extends CellServerState> T getDefaultCellServerState(Properties props) {
	ParticlesCellServerState state = new ParticlesCellServerState();
	state.setTextureURI(“wla://particles-module-snowflake/snowflake.png”);
	return (T)state;
}
```

Tutorial

- 7. Write a *ParticleRotationEffect*. We hope particles can rotate when they are falling down, this can be realized by create a *ParticleRotationEffect*.

```
public class ParticleRotationEffect extends ParticleEffect{
        private Vector3f[] rotationalAxis;
        protected void initalEffect() {
            give every particle a random rotational axis
        }
        protected void updateParticl(int index) {
            rotate the particle denoted by index round its rotational axis
        }
…
}
```

Tutorial

- 8. Open ParticlesCellRenderer and insert the following code before *return node* in createSceneGraph method.

```
int num = 2500;
ParticleSystem ps = new ParticleSystem(num,2,2);
ps.attachToEntity(entity);
ps.setRenderState(bState);
ps.setRenderState(tState);
ps.setRenderState(zState);

ps.setIsRotateWithView(false);
ps.setReleaseRate(0.25f);
ps.setMinVelocity(0.05f);
```

Tutorial

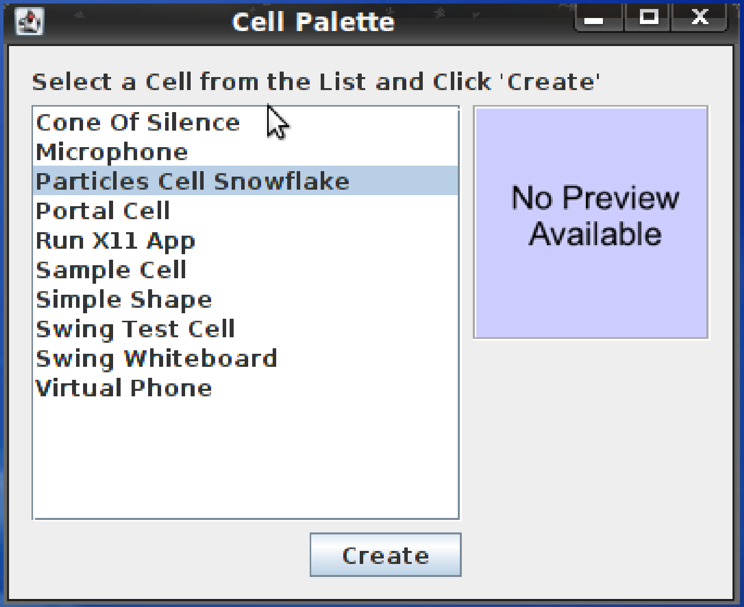
```
ps.setMaxSize(0.10f);
ps.setDirection(new Vector3f(0,-1,0));
ps.setMaxAngle(0.2f);

ps.setInitialArea(new ParticleBaseLocator(new Rectangle(
new Vector3f(20,15,20),new Vector3f (-20,15,20)
,new Vector3f (20,15,-20))));
ps.addParticleConstaint(new ParticleSimpleCuboidConstraint(
new Vector3f (-100f,0,-100f),new Vector3f (100f,100f,100f)));
 ps.addPartilceEffect(new ParticleRotationEffect());

ps.initialPartilceSystem();
ps.start();
```

Tutorial

- 8. Build and deploy this project to the wonderland afte the server successfully starts. Create *Particle Cell Snowflake* in *Cell Palette*.



Cell Palette

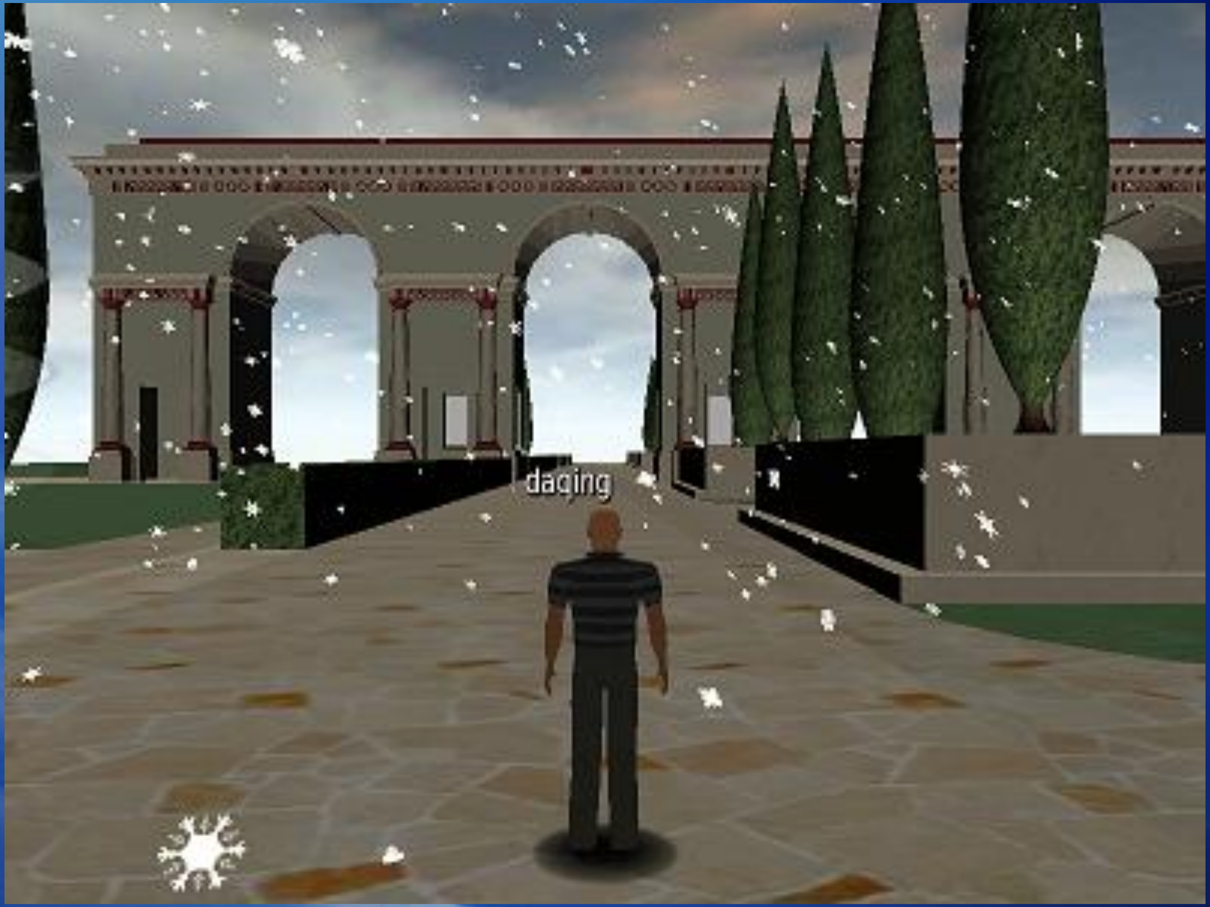
Select a Cell from the List and Click 'Create'

Cone Of Silence
Microphone
Particles Cell Snowflake
Portal Cell
Run X11 App
Sample Cell
Simple Shape
Swing Test Cell
Swing Whiteboard
Virtual Phone

No Preview Available

Tutorial

- result



Tutorial

- The detail code can be found at our home page

Acknowledgement

- This work was supported by Sun China Innovation Program-NJU01.We are especially grateful to engineers from Sun, Dr. Jordan Slott, Mr. Kevin Li, Mr. Jason Tong and Ms. Yang Wang, for their stimulating suggestions and invaluable help for our work. We owe our deepest thanks to Prof. Yangwen Guo, for his insightful comments and infinite patience.

Thank You !