Design and Rendering of Particle Systems
by Joey Black

Introduction

The purpose of this tutorial is to explain the methods I used to design and implement my particle system. My particle system is not intended to actuarially simulate a system of particles, but instead to give a simulation that looks good. To implement my system, I used C++ with OpenGL and GLUT. Many of the functions I use require hardware acceleration that is currently specific to NVIDIA graphics cards.

Particle System Definition

In order to make a particle system, we must first define what we are making. Wikipedia defines a particle system as:

> "...a computer graphics 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, flowing water, sparks, falling leaves, clouds, fog, snow, dust, meteor tails, hair, fur, grass, or abstract visual effects like glowing trails, magic spells, etc."
>
> <http://en.wikipedia.org/wiki/Particle_system>

For our purposes, we will be rendering a large number of individual points, or particles, that behave according to a defined pattern.

Particle System Design

Our particle system will consist of several arrays. The different arrays will be used to hold different pieces of information about each particle. Each space in an array will represent a particle. For our system we will use arrays for the <x,y,z> position of particle, the <r,g,b> color of a particle, the velocity of a particle (represented by a vector) and finally an array that tells us if a particle is alive or not. By "alive" I mean whether or not a particle exists and should be rendered. In our simulations, particles will be created, they will exist for a given length of time, and then they will die. The space left by dead particles can be used for new particles. For a more complex particle system, arrays for mass, size, temperature, etc. could be used.

Simulation Design

There are a variety of ways that you could implement particle system simulations. For this program, I created a Particle System class and added functions that could initialize and update each effect. Only one effect can be used at a time for a particular system since they all share the same particle arrays. Here is the pseudocode for my simulations. You will notice that these simulations do not follow accurate physics models, but instead attempt to simulate them using simplified equations, random numbers, and values that are chosen through trial and error.

- Fountain

Initialize Fountain: the fountain is initialized by clearing the particle system.

Update Fountain:
```
//Create new particles
For i < number of particles to be created (I created 10 per frame):
    find a free particle
    set position to the source of the fountain
    velocity = random((normal-0.3)/100.0f, (normal+0.3)/100.0f)  //this creates particles that shoot up in a cone shaped aria.
    Choose random colors for each particle in the chosen range.
end for
//Update particles
For each particle:
    if particle is free (not alive): skip to next particle
    color -= 0.001 //causes particles to fade
    if color <= 0: free the particle and skip to next particle
    position += velocity
    velocity += acceleration //acceleration = <0, -0.00005, 0>
end for
```
End Update Fountain

- Cyclone

To initialize the cyclone, I generate points along a slanted line with random variations in velocity and position. When the points are rotated around the axis, they fill out the cyclone.

Initialize Cyclone (vector base, vector top, float diameter of base, float diameter of top):
```
a = |diameter of base – diameter of top|
b = sqrt(base * top)
c = sqrt(a*a + b*b)
vector rightAngle = make a unit vector that is perpendicular to the cyclone (any will do)
vector t = rightAngle * diameter of top/2 + top
vector b = rightAngle * diameter of base/2 + base
vector change = (t – b) / number of points
For each point:
    position = b + random(-0.2, 0.2)
    b += change
    color = gray
    velocity.x = random(1, 20) //the x value is used as theta (points are being rotated)
end for
```
end initialize cyclone

Update Cyclone (vector base, vector top, float diameter of base, float diameter of top):
```
For each point:
    rotate point around axis
end for
```
end update cyclone

- Fire

```
Initialize Fire: clear particle system

Update Fire (vector center, float radius):
        For each group of particles to be created:
                vector source = random point within radius
                For each particle in new group:
                        find a free particle
                        position = source
                        velocity = up, with some random variations
                        color = random color within a certain range
                end for
        end for
        vector wind = random direction
        For each particle:
                if particle is free (not alive): skip to next particle
                if color <= 0: free the particle and skip to next particle
                //Pull to center
                if velocity.x>0 then velocity.x -= 0.0001
                else if  velocity.x<0 then velocity.x += 0.0001
                if velocity.z>0 then velocity.z -= 0.0001
                else if  velocity.z<0 then velocity.z += 0.0001
                if color > 0.1 then colors -= 0.01 //cool fire
                if color <= 0.1 then color -= 0.003 //cool smoke
                velocity += wind
                position += velocity + wind
        end for
end update fire
```

- Explode

```
Initialize: clear particle system.

Update Explode(int numNewParticles, vector source):
        for each new particle:
                find a free particle
                position = source (where the mouse was clicked)
                velocity = random (any direction)
                color = random
        end for
        for each particle:
                color -= 0.001
                if color < = 0 then free particle
                position += velocity
        end for
end update explode
```

- Bezier

Initialize Bezier(int numControlPoints, vector[] basePoints, int numPointsOnCurve):
    clear particle system
    base = new Bezier curve(int numControlPoints, vector[] basePoints, int numPointsOnCurve)
    for each point: color = random
end initialize bezier

Update Bezier:
    Each point moves along a Bezier curve generated by the 4 previous points.
    Bezier curves are updated with each frame.
end update Bezier

- Starfield

Initialize Starfield: clear particle system

Update Starfield (int numNewStars, float velocity):
    for each new star:
        find free particle
        position.x = random(-50, 50)
        position.y = random(-50, 50)
        position.z = -200
        color = 1
    end for
    for each point:
        if position.z > -5 the free the particle
        position.z += velocity (same velocity is used for all particles)
    end for
end update starfield

- Universe

Initialize Universe (vector center, vector up, float radius):
    //particles are created in groups of 10
    for each particle Step 10:
        x = random(0,r) + center.x
        ry = random(0, ((-0.5*pow(x,2))/(0.5+pow(x,2)) + 0.5)) //gives bell curve
        y = pow(-1, (int)(random(1,3))) * ry + center.y – random(0,ry/4)
        z = center.z
        v = random(0, (2 - x)) + 0.5
        for next 10 particles:
            position = <x,y,z> + random(-0.0001,0.001)
            color = 0.5
            velocity = random(v-0.4, v+0.4)
        end for
    end for
end initialize universe

Update Universe (vector center, vector up, float radius):
    Rotate all points around center axis using velocity for theta.
end update universe

Rendering

Using traditional rendering methods on particle systems is extremely limiting. Each particle must be rendered individually. Effects like anti-aliasing, blending and bitmaps must be processed for each particle. In order to keep the simulation running in real time, the number of particles used must be severely limited. However, with modern hardware rendering, many of these things can be done in parallel by the graphics card. This allows more particles to be used. The more particles you have the more realistic your simulation will look. A fire simulation will not look very good if you only have 100 particles, but with 10,000 particles the fire begins to take shape. To take advantage of hardware rendering, we will be using OpenGL functions.

To have the particles rendered by the graphics card, we need to send it the array of points that holds the positions. To do this we create an array of type GLfloat with three positions for each point.
points = new Glfloat[numberOfPoints * 3];
Next we need an array for the colors.
pcolors = new Glfloat[numberOfPoints * 4] ;
Notice here we are creating four spaces for each point color. This allows for an alpha value. We can access each element using the following method:
points[p*3] = x
points[p*3+1] = y
points[p*3+2] = z
pcolors[p*4] = r
pcolors[p*4+1] = g
pcolors[p*4+2] = b
pcolors[p*4+3] = a
Now all we have to do is pass these arrays to the graphics card for rendering.
glEnableClientState(GL_COLOR_ARRAY);
glColorPointer(4, GL_FLOAT, 0, pcolors); //4 means we are using r,g,b,a for each color
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, points); //3 means we are drawing 3d points
glDrawArrays(GL_POINTS, 0, numberOfPoints); //one statement draws all the points for us
These statements will draw basic points in 3d.

There are several other effects that can be added with some simple statements:
Anti-aliasing: glEnable(GL_POINT_SMOOTH);
Blending: glBlendFunc (GL_ONE, GL_ONE); //with sprites
    glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); //no sprites
Modify point size by distance from viewer: float quadratic[] = { 1.0f, 0.0f, 0.01f };
  glPointParameterfvARB(GL_POINT_DISTANCE_ATTENUATION_ARB, quadratic);
Set size of points: glPointSize(size);
Fade threshold for distant points:
    glPointParameterfARB(GL_POINT_FADE_THRESHOLD_SIZE_ARB, 1.0f);

Set maximum size of points: glPointParameterfARB(GL_POINT_SIZE_MIN_ARB, minSize);
Set minimum size of points: glPointParameterfARB(GL_POINT_SIZE_MAX_ARB, maxSize);
Sprites: glTexEnvf(GL_POINT_SPRITE_ARB, GL_COORD_REPLACE_ARB, GL_TRUE);
glEnable(GL_TEXTURE_2D);
glEnable(GL_POINT_SPRITE_ARB);
(Texture must be already loaded. Code to load texture can be found at <http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=06>)
All these functions should be put in your draw function for your particle system. The code to draw the points should be last. It is also a good idea to disable (glDisable()) everything that was enabled (glEnable()) at the end of your draw function.

Conclusion

Using these methods I was able to implement several simulations that rendered up to 10,000 particles in real time. Without hardware acceleration, I would not be able to render more than a couple hundred particles, and they would not have any special effects like sprites or blending. Allowing the graphics card to handle all the rendering allows for a significant improvement in speed. If the program ran entirely on the graphics card, the program could run in real time with even more particles.