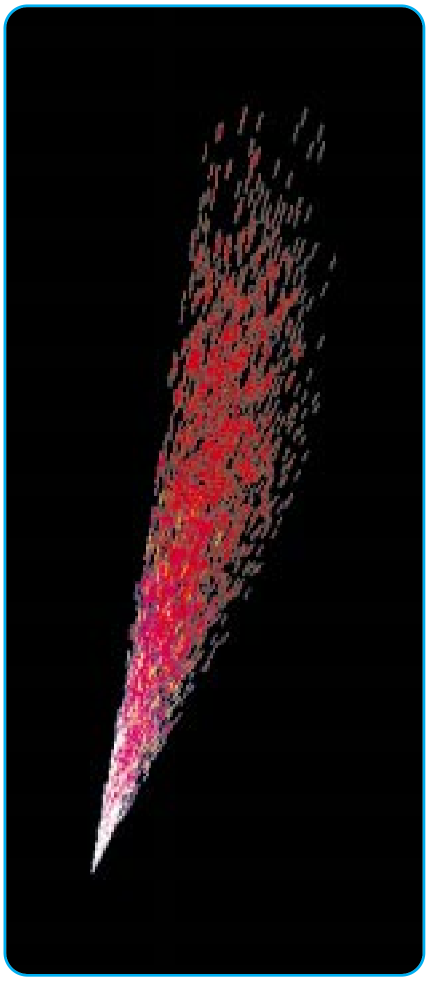
Back

The Ocean Spray in Your Face

Judging by the number of times the question comes up in public forums such as Usenet, particle systems are a pretty hot issue. This may be partially a result of the phenomenal success of QUAKE, with its use of particles for smoke, blood trails, and spark falls.

But certainly, the interest in particle systems has something to do with their ability, more so than any other computer graphics method, to create realistic natural phenomena in real time. William Reeves realized this all the way back in 1982 and 1983. When working on *Star Trek II: The Wrath of Khan*, he was in search of a method for creating realistic fire for the Genesis Demo sequence. Reeves realized that conventional modeling, which was best at creating objects that have smooth, well-defined surfaces, wouldn't do the trick. The objects that made up these effects were not made of easily definable surfaces. These objects, which he termed "fuzzy," would be better modeled as a system of particles that behaved within a set of dynamic rules. Particles had been used previously to create natural effects such as smoke and galaxies of stars, but were difficult to control. Reeves realized that by applying a system of rules to particles, he could achieve a chaotic effect while maintaining some creative control. Thus was born the particle system.



How Does It Work?

A particle system is basically just a collection of 3D points in space. Unlike standard geometry objects, particles making up the system are not static. They go through a complete life cycle. Particles are born, change over time, and then die off. By adjusting the parameters that influence this life cycle, you can create different types of effects.

Another key point regarding particle systems is that they are chaotic. That is, instead of having a completely predetermined path, each particle can have a random element that modifies its behavior. It's this random element, called a stochastic process (a good nerd party word), that makes the effect look very organic and natural. This month, I'm going to create a real-time particle system that will show off the basic techniques as well as some eye-catching effects you can create.

The Particle

Let's start by looking at what properties are needed in a particle. First, I need to know the position of the particle. I'm going to store the previous position as well, because I also want to be able to antialias the particles easily. I need to know the direction in which the particle is currently traveling. This can be stored as a direction vector. I also need to know the current speed at which this particle is traveling in that direction, but speed can simply be combined with the direction vector by multiplication. I'm going to render

LISTING 1. *The Particle Structure.*

```
struct tParticle
{
    tParticle *prev,*next;          // LINK
    tVector   pos;                  // CURRENT POSITION
    tVector   prevPos;              // PREVIOUS POSITION
    tVector   dir;                  // CURRENT DIRECTION WITH SPEED
    int life;                       // HOW LONG IT WILL LAST
    tColor color;                   // CURRENT COLOR OF PARTICLE
    tColor prevColor;               // LAST COLOR OF PARTICLE
    tColor  deltaColor;             // CHANGE OF COLOR
};
```

Jeff is a complex particle system at Darwin 3D. E-mail him at jeffl@darwin3d.com. But beware that his replies are subject to stochastic reliability.

particles as colored points, so I also need to know the current color of this particle and the previous color for antialiasing. In order to change the color over time, I'm going to store the amount of change in color per frame also. The last piece of information that I need is the life count for this particle. This is the number of frames that this particle will exist before dying.

You can see a data structure for my particles in Listing 1. If you wished to make your particle system more complex, it would be very easy to add properties here. You could animate the size of the particles by adding a size, the transparency by adding an alpha component to the color. You could furthermore add mass, other physical properties, or any number of other variables.

The Emitter

The particle emitter is the entity responsible for creating the particles in the system. This is the object that you would drop around in a real-time 3D world to create different effects. The emitter controls the number of particles and general direction in which they should be emitted as well as all the other global settings. The structure for the emitter is in Listing 2. This is also where I set up the stochastic processes that I was talking about. For example, **emitNumber** is the average number of particles that should be emitted each frame. The **emitVariance** is the random number of particles either added or subtracted from base **emitNumber**. By adjusting these two values, you can change the effect from a constant, steady stream to a more random flow. The formula for calculating how many particles to emit each frame is

particleCount = emitNumber + (emitVariance * RandomNum());

Where **RandomNum()** is a function that returns a number between -1.0 and 1.0.

These techniques are also used to vary the color, direction, speed, and life span of a particle. The color is a special case because I want the color to change over the life span of the particle. I calculate two randomly varied colors as above and then divide the difference between them by the life. This creates the color delta that is added to each particle each frame of its life.

I now need to describe the direction in which the particles should be emitted. We really only need to describe two angles of rotation about the origin because the particles are single points in space, and I'm not concerned with the spin. Those two angles are the rotation about the y axis (yaw or azimuth defined by θ) and the rotation about the x axis (pitch or inclination defined by ψ). These angles are varied by a random value and then converted to a direction vector for each particle.

The conversion process for generating this direction vector is pretty easy. It requires some general 3D rotation techniques and some basic matrix math.

A rotation about y is defined as

$x' = x*\cos(\theta) + z*\sin(\theta);$
$y' = y;$
$z' = -x*\sin(\theta) + z*\cos(\theta)$

or, in matrix form,

$$\mathbf{Roty(\theta)} = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

A rotation of about x is

$x' = x;$
$y' = y*\cos(\psi) - z*\sin(\psi);$
$z' = y*\sin(\psi) + z*\cos(\psi)$

or

LISTING 2. *The emitter structure.*

```
struct tEmitter
{
    long      id;                         // EMITTER ID
    char      name[80];                   // EMITTER NAME
    long      flags;                      // EMITTER FLAGS
    // TRANSFORMATION INFO
    tVector        pos;                   // XYZ POSITION
    float     yaw, yawVar;                // YAW AND VARIATION
    float     pitch, pitchVar;            // PITCH AND VARIATION
    float     speed,speedVar;
    // Particle
    tParticle *particle;                  // NULL TERMINATED LINKED LIST
    int       totalParticles;             // TOTAL EMITTED AT ANY TIME
    int       particleCount;              // TOTAL EMITTED RIGHT NOW
    int       emitsPerFrame, emitVar;     // EMITS PER FRAME AND VARIATION
    int       life, lifeVar;              // LIFE COUNT AND VARIATION
    tColor    startColor, startColorVar;  // CURRENT COLOR OF PARTICLE
    tColor    endColor, endColorVar;      // CURRENT COLOR OF PARTICLE
    // Physics
    tVector        force;                 // GLOBAL GRAVITY, WIND, ETC.
};
```

LISTING 3. *Converting rotations to a direction vector.*

```
///////////////////////////////////////////////////////////////////////////////
// Function:  RotationToDirection
// Purpose:      Convert a Yaw and Pitch to a direction vector
///////////////////////////////////////////////////////////////////////////////
void RotationToDirection(float pitch,float yaw,tVector *direction)
{
    direction->x = -sin(yaw) * cos(pitch);
    direction->y = sin(pitch);
    direction->z = cos(pitch) * cos(yaw);
}
/// initParticleSystem ////////////////////////////////////////////////////////
```

LISTING 4. *Adding a new particle to an emitter.*

```
///////////////////////////////////////////////////////////////////////////
// Function:  addParticle
// Purpose:   add a particle to an emitter
// Arguments: The emitter to add to
///////////////////////////////////////////////////////////////////////////
BOOL addParticle(tEmitter *emitter)
{
/// Local Variables ///////////////////////////////////////////////////////
    tParticle *particle;
    tColor start,end;
    float yaw,pitch,speed;
///////////////////////////////////////////////////////////////////////////
    // IF THERE IS AN EMITTER AND A PARTICLE IN THE POOL
    // AND I HAVEN'T EMITTED MY MAX
    if (emitter != NULL && m_ParticlePool != NULL &&
        emitter->particleCount < emitter->totalParticles)
    {
        particle = m_ParticlePool;                // THE CURRENT PARTICLE
        m_ParticlePool = m_ParticlePool->next;    // FIX THE POOL POINTERS

        if (emitter->particle != NULL)
            emitter->particle->prev = particle;   // SET BACK LINK
        particle->next = emitter->particle;       // SET ITS NEXT POINTER
        particle->prev = NULL;                    // IT HAS NO BACK POINTER
        emitter->particle = particle;             // SET IT IN THE EMITTER

        particle->pos.x = 0.0f;                   // RELATIVE TO EMITTER BASE
        particle->pos.y = 0.0f;
        particle->pos.z = 0.0f;

        particle->prevPos.x = 0.0f;               // USED FOR ANTI ALIAS
        particle->prevPos.y = 0.0f;
        particle->prevPos.z = 0.0f;

        // CALCULATE THE STARTING DIRECTION VECTOR
        yaw = emitter->yaw + (emitter->yawVar * RandomNum());
        pitch = emitter->pitch + (emitter->pitchVar * RandomNum());

        // CONVERT THE ROTATIONS TO A VECTOR
        RotationToDirection(pitch,yaw,&particle->dir);

        // MULTIPLY IN THE SPEED FACTOR
        speed = emitter->speed + (emitter->speedVar * RandomNum());
        particle->dir.x *= speed;
        particle->dir.y *= speed;
        particle->dir.z *= speed;

        // CALCULATE THE COLORS
        start.r = emitter->startColor.r + (emitter->startColorVar.r * RandomNum());
        start.g = emitter->startColor.g + (emitter->startColorVar.g * RandomNum());
        start.b = emitter->startColor.b + (emitter->startColorVar.b * RandomNum());
        end.r = emitter->endColor.r + (emitter->endColorVar.r * RandomNum());
        end.g = emitter->endColor.g + (emitter->endColorVar.g * RandomNum());
        end.b = emitter->endColor.b + (emitter->endColorVar.b * RandomNum());

        particle->color.r = start.r;
        particle->color.g = start.g;
        particle->color.b = start.b;

        // CALCULATE THE LIFE SPAN
        particle->life = emitter->life + (int)((float)emitter->lifeVar * RandomNum());

        // CREATE THE COLOR DELTA
        particle->deltaColor.r = (end.r - start.r) / particle->life;
        particle->deltaColor.g = (end.g - start.g) / particle->life;
        particle->deltaColor.b = (end.b - start.b) / particle->life;
        emitter->particleCount++;                 // A NEW PARTICLE IS BORN
        return TRUE;
    }
    return FALSE;
}
/// addParticle ///////////////////////////////////////////////////////////
```

$$\mathrm{Rotx}(\psi)=\begin{bmatrix}1 & 0 & 0\\ 0 & \cos(\psi) & \sin(\psi)\\ 0 & -\sin(\psi) & \cos(\psi)\end{bmatrix}$$

Once these two matrices are combined into a single rotation matrix, I get the following:

$$\mathrm{RotMatrix}=\begin{bmatrix}\cos(\theta) & \sin(\psi)\sin(\theta) & -\sin(\theta)\cos(\psi)\\ 0 & \cos(\psi) & \sin(\psi)\\ \sin(\theta) & -\sin(\psi)\cos(\theta) & \cos(\psi)\cos(\theta)\end{bmatrix}$$

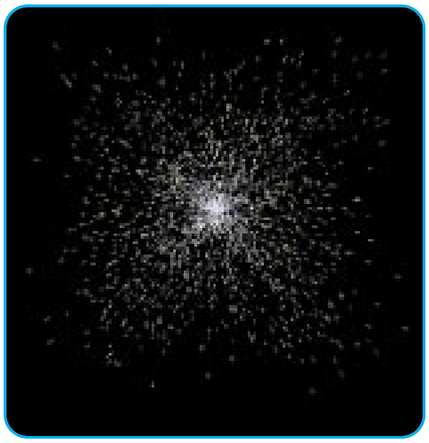
Now, since I'm calculating a direction vector, I need to multiply the vector (0,0,1) by this matrix. Once the zeros are all dropped out, I get the final piece of code in Listing 3.To finalize the particle motion vector, this final direction vector is multiplied by the speed scalar, which is also randomly modified.

Creating a New Particle

To avoid many costly memory allocations, all particles are created in a common particle pool. I chose to implement this as a linked list. When a particle is emitted, it's removed from the common pool and added to the emitter's particle list. While this limits the total number of particles I can have in the scene, it also speeds things up a bunch. By making the particle bidirectionally linked, it's easy to remove a particle when it dies.

The code that creates a new particle and adds it to the emitter is in Listing 4. It handles all the list management for the global pool and also sets up all the stochastic settings for the particle.

I chose simply to create each new particle at the origin of the emitter. In his SIGGRAPH paper, William Reeves describes generating particles in differ-



ent ways (see "References"). Along with a point source, he describes methods for creating particles on the surface of a sphere, within the volume of a sphere, on the surface of a 2D disc, and on the surface of a rectangle. These different methods will create various effects, so you should experiment to find what works best for your application.

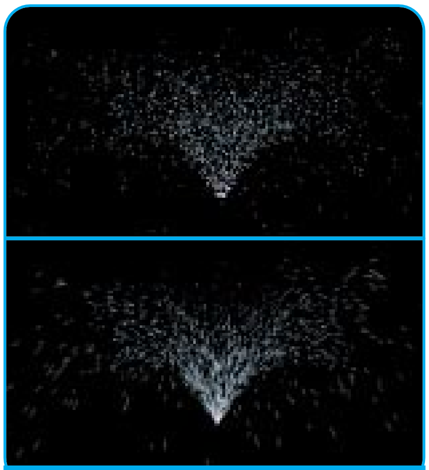
Updating the Particle

Once a particle is born, it's handled by the particle system. The update routine is in Listing 5. For each cycle of the simulation, each particle is updated. First, it's checked to see if it has died. If it has, the particle is removed from the emitter and returned to the global particle pool. At this time also, global forces are applied to the direction vector, and the color is modified.

Rendering the Particle System

A particle system is simply a collection of points, and so it can be rendered as just that, a set of colored 3D points. You can also calculate a polygon around the point so that it always faces the camera like a billboard. Then apply any texture you like to the polygon. By scaling the polygon with the distance from the camera, you can create perspective. Another option is to draw a 3D object of any type at the position of the particle.

I took the simple route. I just drew each particle as a 3D point. If you turn on antialiasing, the system draws a gouraud-shaded line from the previous position and color to the new position and color. This tends to smooth out the look at the cost of some rendering speed. You can see the difference in Figures 1a and 1b. The first image is a simple point rendering, and the second is composed of line segments.



FIGURES 1a AND 1b. *1a shows point rendering and 1b shows a composition of line segments.*

LISTING 5. *Updating a Particle.*

```
///////////////////////////////////////////////////////////////////////////////
// Function:  updateParticle
// Purpose:      updateParticle settings
// Arguments: The particle to update and the emitter it came from
///////////////////////////////////////////////////////////////////////////////
BOOL updateParticle(tParticle *particle,tEmitter *emitter)
{
   // IF THIS IS A VALID PARTICLE
   if (particle != NULL && particle->life > 0)
   {
      // SAVE ITS OLD POS FOR ANTI ALIASING
      particle->prevPos.x = particle->pos.x;
      particle->prevPos.y = particle->pos.y;
      particle->prevPos.z = particle->pos.z;

      // CALCULATE THE NEW
      particle->pos.x += particle->dir.x;
      particle->pos.y += particle->dir.y;
      particle->pos.z += particle->dir.z;

      // APPLY GLOBAL FORCE TO DIRECTION
      particle->dir.x += emitter->force.x;
      particle->dir.y += emitter->force.y;
      particle->dir.z += emitter->force.z;

      // SAVE THE OLD COLOR
      particle->prevColor.r = particle->color.r;
      particle->prevColor.g = particle->color.g;
      particle->prevColor.b = particle->color.b;

      // GET THE NEW COLOR
      particle->color.r += particle->deltaColor.r;
      particle->color.g += particle->deltaColor.g;
      particle->color.b += particle->deltaColor.b;

      particle->life--;// IT IS A CYCLE OLDER
      return TRUE;
   }
   else if (particle != NULL && particle->life == 0)
   {
      // FREE THIS SUCKER UP BACK TO THE MAIN POOL
      if (particle->prev != NULL)
         particle->prev->next = particle->next;
      else
         emitter->particle = particle->next;
      // FIX UP THE NEXT'S PREV POINTER IF THERE IS A NEXT
      if (particle->next != NULL)
         particle->next->prev = particle->prev;
      particle->next = m_ParticlePool;
      m_ParticlePool = particle;  // NEW POOL POINTER
      emitter->particleCount--;   // ADD ONE TO POOL
   }
   return FALSE;
}
/// updateParticle ////////////////////////////////////////////////////////////
```

What Can You Do With It?

Once you've designed your system, you can start building effects. You can easily build effects such as fire, water fountains, spark showers, and others simply by modifying the emitter properties. By attaching the emitter to another object and actually animating it, you can create simple smoke trails or a comet tail.

You can also create even more complex effects by creating a brand new particle system at the point at which each particle dies. The Genesis sequence in *Star Trek II* actually had up to 400 particle systems consisting of 750,000 particles. That may be a bit much for your real-time blood spray, but as hardware gets faster, who knows?

Also, my simple physics model could be greatly modified. The mass of the particles could be randomized, causing gravity to effect them differently. A friction model would force some particles to slow down while animating. The addition of local spatial effects, such as magnetic fields, wind gusts, and rotational vortexes, would vary the particles even more. Or you could vary the `emitsPerFrame` in a cycle over time to create a puffing smoke effect.

I've seen many other ideas implemented in commercial particle systems. You can animate the size of the particle over time to create a dispersing effect. Add more color key positions over the particle's lifetime to create a more complex look. Another interesting variation is the use of a particle system to create plants. By keeping track of each position over the life of a particle and then rendering a line through all those points, you get an object that resembles a clump of grass. Organic objects such as this would be difficult to hand-model convincingly with polygons. Another area for expansion is collision detection. You could create particles that bounce off of boundary objects such as cubes and spheres by simply reflecting the direction vector off of the surface.

You can see from these ideas that I've just begun to explore what can be created with particle systems. By creating a flexible particle engine, you can achieve many different effects by modifying a few simple settings. These flexible emitters can easily be dropped into an existing 3D real-time engine to add to the realism and excitement of a simulation.

The source code and application this month demonstrate the use of a particle system. The emitter settings can be manipulated via a dialog box to create custom effects. These settings can be saved to create a library of emitters. Get the source and application on the *Game Developer*'s web site at www.gdmag.com. ■

REFERENCES

Reeves, William T. "Particle Systems — A Technique for Modeling a Class of Fuzzy Objects." Computer Graphics, Vol. 17, No. 3 (1983): 359-376.

Reeves, William T. "Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particles Systems." Computer Graphics, Vol. 19, No. 3 (1985): 313-322.

Watt, Alan, *3D Computer Graphics*. Reading, Mass.: Addison Wesley, 1993.