Building an Advanced Particle System

by John van der Burg

Imagine a scene in a game in which a rocket flies through the air, leaving a smoke trail behind it. Suddenly the rocket explodes, sparks flying everywhere. Out of the disintegrating rocket a creature is jettisoned towards you, its body parts exploding and blood flying through the air, leaving messy blood splatters on the camera lens. What do most of the elements in this scene have in common?

Yes, most of these elements are violent. But in terms of technology, most of the effects in a scene like this would benefit from a good particle system. Smoke, sparks, and blood are routinely created in today's games using particle systems.

To realize these effects, you need to build a particle system, and not just a simple one. You need an advanced particle system, one that's fast, flexible, and extensible. If you are new to particle systems, I recommend you begin by reading Jeff Lander's article on particle systems ("The Ocean Spray in Your Face," Graphic Content, July 1998). The difference between Lander's column and this article is that the former describes the basics of particles, whereas I will demonstrate how to build a more advanced system. With this article I will include the full source code for an advanced particle system, and you can download an application that demonstrates the system.

Performance and Requirements

Advanced particle systems can result in pretty large amounts of code, so it's important to design your data structures well. Also be aware of the fact that particle systems can decrease the frame rate significantly if not constructed properly, and most performance hits are due to memory management problems caused by the particle system.

When designing a particle system, one of the first things to keep in mind is that particle systems greatly increase the number of visible polygons per frame. Each particle probably needs four vertices and two triangles. Thus, with 2,000 visible snowflake particles in a scene, we're adding 4,000 visible triangles for the snow alone. And since most particles move, we can't precalculate the vertex buffer, so the vertex buffers will probably need to be changed every frame.

The trick is to perform as few memory operations (allocations and releases)

John van der Burg is the lead programmer for Mystic Game Development, located in the Netherlands. Currently he is working on Oxygen3D, which is his third hardware-only engine and his eighth engine overall. Currently he is doing freelance work on LOOSE CANNON for Digital Anvil and for OMG Games on THE CREST OF DHARIM. He previously freelanced for Lionhead Studios on BLACK AND WHITE, and for Orange Games on CORE. You can find screenshots of his previous work at http://www.mysticgd.com. Feel free to drop him a line at john@mysticgd.com.

as possible. Thus, if a particle dies after some period of time, don't release it from memory. Instead, set a flag that marks it as dead or respawn (reinitialize) it. Then when all particles are tagged as "dead," release the entire particle system (including all particles within this system), or if it's a constant system, keep the particle system alive. If you want to respawn the system or just add a new particle to a system, you should automatically initialize the particle with its default settings/properties set up according to the system to which it belongs.

For example, let's say you have a smoke system. When you create or respawn a particle, you might have to set its values as described in Table 1. (Of course, the start color, energy, size, and velocity will be different for blood than, say, smoke.) Note that the values also depend on the settings of the system itself. If you set up wind for a smoke system so the smoke blows to the left, the velocity for a new particle will differ from a smoke system in which the smoke just rises unaffected by wind. If you have a constant smoke system, and a smoke particle's energy becomes 0 (so you can't see it anymore), you'll want to respawn its settings so it will be replaced at the bottom of the smoke system at full energy.

Some particle systems may need to have their particles rendered in different ways. For example, you may want to have multiple blood systems, such "blood squirt," "blood splat," "blood pool," and "blood splat on camera lens," each containing the appropriate particles. "Blood squirt" would render blood squirts flying through the air, and when these squirts collided with a wall, the "blood splat" system would be called, creating messy blood splats on walls and floors. A blood pool system would create pools of blood on the floor after someone had been shot dead on the ground.

Each particle system behaves in a unique manner. Blood splats are rendered differently than smoke is displayed. Smoke particles always face the active camera, whereas blood splats are mapped (and maybe clipped) onto the plane of the polygon that the splat collides with.

When creating a particle system, it is important to consider all of the possible parameters that you may want to affect in the system at any time in the game, and build that flexibility into your system. Consider a smoke system again. We might want to change the wind direction vector so that a car moving closely past a smoke system makes the smoke particles respond to the wind generated by the passing car.

At this point you may have realized that each of these systems (blood splat, smoke, sparks, and so on) is very specific to certain tasks. But what if we want to control the particles within a system in a way not supported by the formulae in the system? To support that kind of flexibility, we need to create a "manual" particle system as well, one that allows us to update all particle attributes every frame.

The last feature we might consider is the ability to link particle systems within the hierarchy of our engine. Perhaps at some point we'll want to link a smoke or glow particle system to a cigarette, which in turn is linked to the head of a smoking character. If the character moves its head or starts to walk, the position of the particle systems which are linked to the cigarette should also be updated correctly.

So there you have some basic requirements for an advanced particle system. In the next section, I'll show how to design a good data structure that is capable of doing all the above-mentioned features.

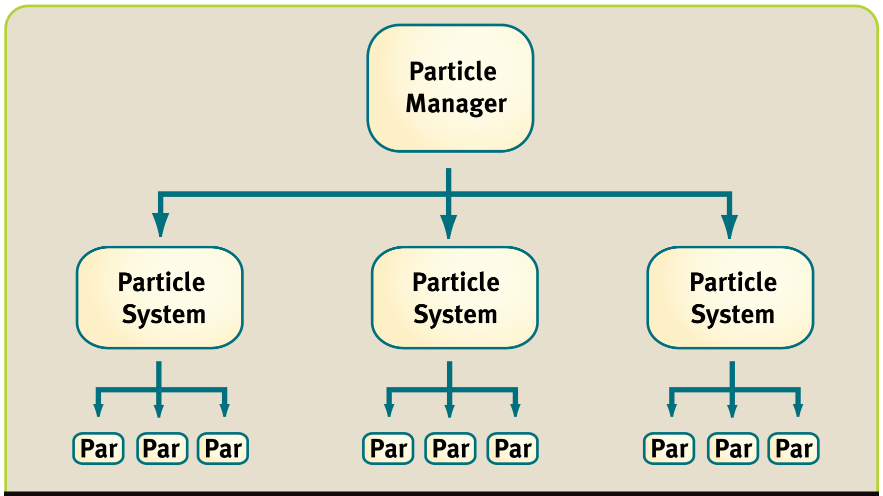


FIGURE 1. *A global overview of the system we are going to build.*

TABLE 1. *Particle attributes.*

Data type	Name	Description
Vector3	position	The position of the particle in world-space
Vector3	oldPos	The previous position of the particle, useful in some systems
Vector3	velocity	The velocity vector (position += velocity)
dword	color	The color of the particle (its vertex colors)
int	energy	The energy of the particle
float	size	The size of the particle

Creating the Data Structure

Now that we know what features we need, it's time to design our classes. Figure 1 shows an overview of the system we're going to build. Notice that there is a particle manager, which I will explain more about in a moment.

Let's use a bottom-up approach to design our classes, beginning with the particle class.

THE PARTICLE CLASS. If you have built a particle system before, you probably know the types of attributes a particle must have. Table 1 lists of some common attributes.

Note that the previous position of a particle can also be useful in some systems. For example, you might want to stretch a particle between its previous and current positions. Sparks are a good example of particles that benefit

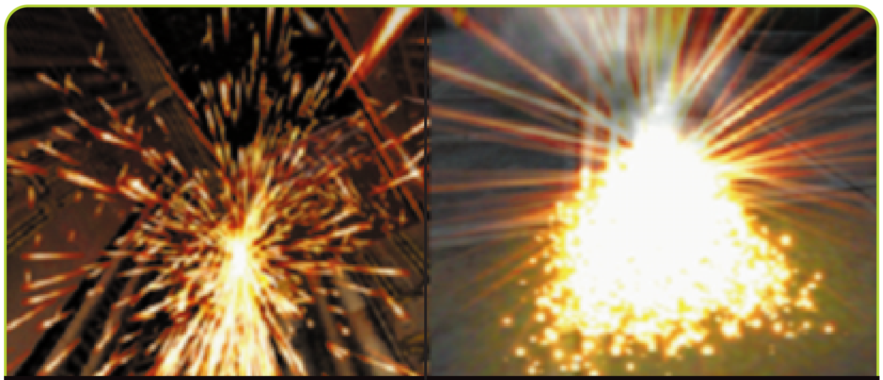


FIGURE 2. *Some spark effects you can create using a particle system such as the one discussed in this article. Note that all particles perform accurate collision detection and response in these two screenshots.*

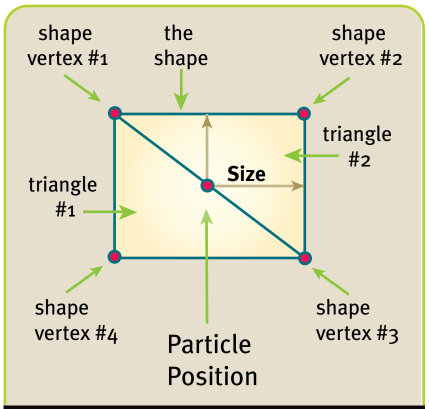


FIGURE 3. *Setting up a shape to render a particle.*

LISTING 1. *Calculating the shape of a particle facing the camera.*

```
void ParticleSystem::SetupShape(int nr)
{
    assert(nr < shapes.Length());     // make sure we don't try to shape anything we
                                      // don't have

    // calculate cameraspace position
    Vector3 csPos = gCamera->GetViewMatrix() * particles[nr].position;

    // set up shape vertex positions
    shapes[nr].vertex[0] = csPos + Vector3(-particles[nr].size, particles[nr].size, 0);
    shapes[nr].vertex[1] = csPos + Vector3( particles[nr].size, particles[nr].size, 0);
    shapes[nr].vertex[2] = csPos + Vector3( particles[nr].size, -particles[nr].size, 0);
    shapes[nr].vertex[3] = csPos + Vector3(-particles[nr].size, -particles[nr].size, 0);
}
```

from this feature. You can see some spark effects I've created in Figure 2.

The color and energy attributes can be used to create some interesting effects as well. In a previous particle system I created, I used color within the smoke system, which let me dynamically light the smoke particles using lights within the scene.

Energy value is very important as well. Energy is analogous to the age of the particle — you can use this to determine whether a particle has died. And because the color or intensity of some particles (such as sparks) changes over time, you may want to link it to the alpha channel of the vertex colors.

I strongly recommend that you leave the constructor of your particle class empty, because you don't want to use default values at construction time, simply because these values will be different for most particle systems.

THE PARTICLE SYSTEM CLASS. This class is the heart of the system. Updating the particle attributes and setting up the shape of the particles takes place inside this class. My current particle system class uses the node base class of my 3D engine, which contains data such as a position vector, a rotation quaternion, and scale values. Because I inherit all members of this node class, I can link my particle systems within the hierarchy of the engine, allowing the engine to affect the position of the particle system as discussed in the above cigarette example. If your engine does not have hierarchy support, or if you are building a stand-alone particle system, this is not needed. Table 2 lists the attributes which you need to have in the particle system base class.

Here's how to calculate the four positions of a normal (not stretched) particle that always faces the active camera. First, transform your particle world-space position into camera-space (multiply the world-space position and your active camera matrix) using the size attribute of the particle to calculate the four vertices.

The four vertices, which form the shape, are what we use to render the particle, though a particle has only one position, *xyz*. In order to render a particle (such as a spark), we need to set up a shape (created from four vertices). Two triangles are then rendered between these four points. Imagine a non-stretched particle always facing the camera in front of you, as seen in Figure 3. In our case, the particle is always facing the active camera, so this means we can simply add and subtract values from the *x* and *y* values of the particle position in camera-space. In other words, leave the *z* value as it is and pretend you are working only in 2D. You can see an example of this calculation in Listing 1.

THE FUNCTIONS. Now that we know what attributes are needed in the particle system base class, we can start thinking about what functions are needed. Since this is the base class, most functions are declared as virtual functions. Each type of particle system updates particle attributes in a different way, so we need to have a virtual update function. This update function performs the following tasks:

- Updates all particle positions and other attributes.
- Updates the bounding box if we can't precalculate it.
- Counts the number of alive particles. It returns FALSE if there are no alive particles, and returns TRUE if particles are still alive. The return value can be used to determine whether a system is ready to be deleted or not.

TABLE 2. *Particle system base class attributes.*

Data type	Name	Description
`Texture`	`*texture`	A pointer to a texture, which all particles will use. For performance reasons, we only use one texture for each individual particle system; all particles within the specific system will have the same texture assigned.
`BlendMode`	`blendMode`	The blend mode you want to use for the particles. Smoke will probably have a different blend mode from blood — that's the reason you also store the blend mode for each particle system.
`int`	`systemType`	A unique ID, which represents the type of system (smoke or sparks, for example). The `systemType` identifier is also required, since you may want to check for a specific type of particle system within the collection of all systems. For example, to remove all smoke systems, you need to know whether a given system is a smoke system or not.
`Array Particle`	`particles`	The collection of particles within this system. This may also be a linked list instead of an array.
`Array PShape`	`shapes`	A collection of shapes, describing the shapes of the particles. The shape descriptions of the particles usually consist of four positions in 3D camera-space. These four positions are used to draw the two triangles for our particle. As you can see in Table 1, a particle is only stored as a single position, but it requires four positions (vertices) to draw the texture-mapped shape of the particle.
`int`	`nrAlive`	Number of particles in the system which are still alive. If this value is zero, it means all particles are dead and the system can be removed.
`BoundingBox3`	`boundingBox`	The 3D axis-aligned bounding box (AABB), used for visibility determination. We can use this for frustum, portal, and anti-portal checks.

Now our base class has the ability to update the particles, and we are ready to set up the shapes which can be constructed using the new (and perhaps previous) position. This function, `SetupShape`, needs to be virtual, because some particle system types will need to have their particles stretched and some won't. You can see an example of this function in Listing 1.

To add a particle to a given system, or to respawn it, it's useful to have a function that takes care of this. Again, it should be another virtual function, which is declared like this:

```
virtual void SetParticleDefaults( Particle &p );
```

As I explained above, this function initializes the attributes for a given particle. But what if you want to alter the speed of the smoke or change the wind direction that affects all of your smoke systems? This brings us to the next subject: the particle system's constructor. Many particle systems will need their own unique constructors, forcing us to create a virtual constructor and destructor within the base class. In the constructor of the base class, you should enter the following information:

- The number of particles you initially want to have in this particle system.
- The position of the particle system itself.
- The blend mode you want to use for this system.
- The texture or texture file name you want this system to use.
- The system type (its ID).

In my engine, the constructor in the particle system base class looks like this:

```
virtual ParticleSystem(int nr, rcVector3 centerPos, BlendMode
blend=Blend_AddAlpha, rcString file name="Effects/Particles/
green_particle", ParticleSystemType type=PS_Manual);
```

So where do various settings, such as the wind direction for the smoke system, get addressed? You can either add settings specific to the system type (such as wind direction) into the constructor, or you can create a `struct` called `InitInfo` inside each class, which contains all of the appropriate settings. If you use the latter method, make sure to add a new parameter in the constructor, which is a pointer to the new `struct`. If the pointer is NULL, the default settings are used.

As you can imagine, the first solution can result in constructors with many parameters, and that's not fun to work with as programmer. ("Parameter number 14…hmmm. What does that value represent again?") That's the main reason I don't use the first method. It's much easier to use the second

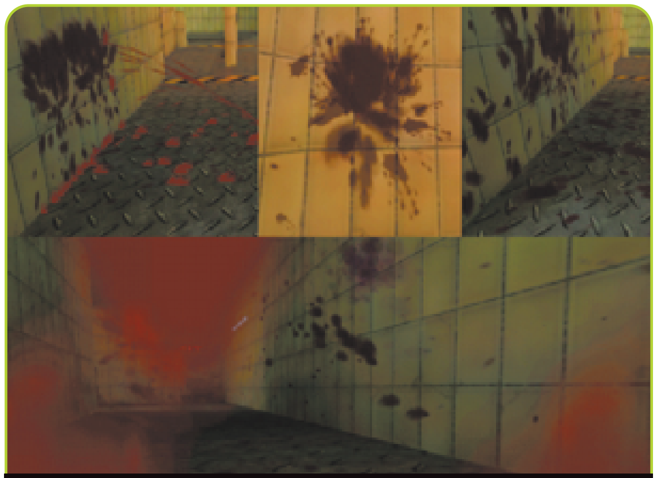

A blood system. Blood colors were set based on the colors in the light maps. Blood on dark areas looks dark as well. The red areas in the bottom screenshot are the blood splats on the camera lens, dripping down the lens.

TABLE 3. *Particle manager class functions.*

`Init`	Initializes the particle manager.
`AddSystem`	Adds a specified particle system to the manager.
`RemoveSystem`	Removes a specified particle system.
`Update`	Updates all active particle systems and removes all systems which died after the update.
`Render`	Renders all active and visible systems.
`Shutdown`	Shuts down the manager (removes all allocated systems).
`DoesExist`	Checks whether a given particle system still exists in the particle manager (if it has not been removed yet).

method, and we can create a function in each particle system class to initialize its `struct` with default settings. An example of this code and a demo application can be found on the *Game Developer* web site (http://www.gdmag.com) or my own site at http://www.mysticgd.com.

The Particle Manager

Now that we have covered the technology behind an individual particle system, it's time to create a manager class to control all of our various particle systems. A manager class is in charge of creating, releasing, updating, and rendering all of the systems. As such, one of the attributes in the manager class must be an array of pointers to particle systems. I strongly recommend that you build or use an array template, because this makes life easier.

The people who will work with the particle systems you create want to add particle systems easily. They also don't want to keep track of all the systems to see if all of the particles died so they can release them from memory. That's what the manager class is for. The manager will automatically update and render systems when needed, and remove dead systems.

When using sporadic systems (systems which die after a given time), it's useful to have a function that checks whether a system has been removed yet (for example, if it still exists within the particle manager). Imagine you create a system and store the pointer to this particle system. You access the particle system every frame by using its pointer. What happens if the system dies just before you use the pointer? Crash. That's why we need to have a function which checks if the system is still alive or has already been deleted by the particle manager. A list of functions needed inside the particle manager class is shown in Table 3.

This was constructed from sparks and a big real-time calculated flare explosion (not just a texture).

This image is the same as the above one, but with some extra animated explosions (animated textures) and shockwaves, which are admittedly very small and may be difficult to see in this image.

The `AddSystem` function will probably have just one parameter: the pointer to the particle system which is of the type of our particle system base class. This allows you to add a smoke or fire system easily depending on your needs. Here is an example of how I add a new particle system in my engine:

```
gParticleMgr->AddSystem( new Smoke(nr
SmokeParticles, position, ...) );
```

During the world update function, I call the `gParticleMgr->Update()` function, which automatically updates all of the systems and releases the dead ones. The `Render` function then renders all visible particle systems.

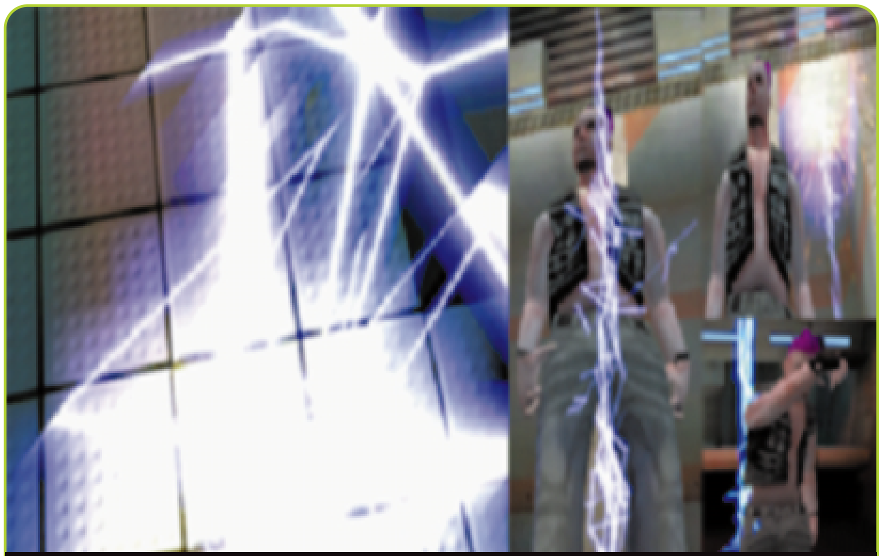
Since we don't want to keep track of all particles across all of our systems every frame to see whether all particles have died (so the system can be removed), we'll use the `Update` function instead. If this function returns TRUE, it means that the system is still alive; otherwise it is dead and ready to be removed. The `Update` function of the particle manager is shown in Listing 2.

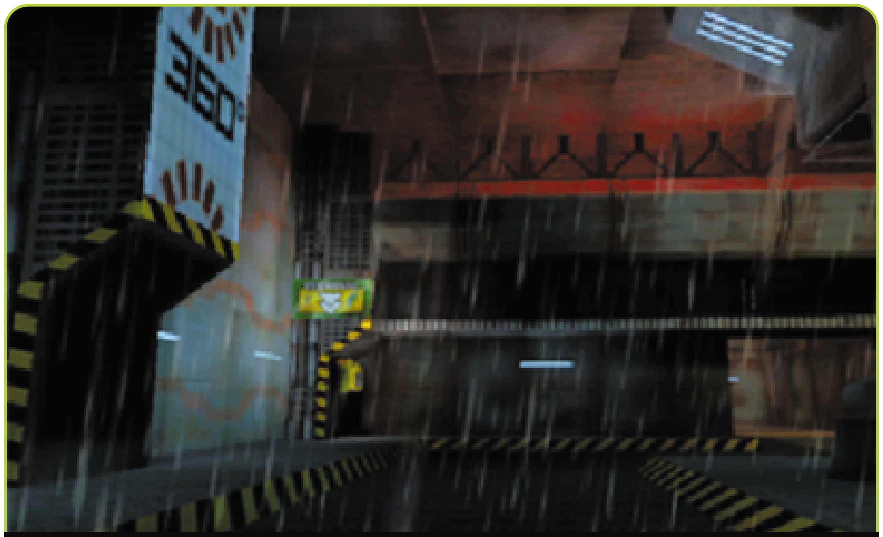
In my own particle system, all particles with the same textures and blend modes assigned to them will be rendered consecutively, minimizing the number of texture switches and uploads. Thus, if there are ten smoke systems visible on screen, only one texture switch and state change will be performed.

Design, Then Code

Designing a flexible, fast, and extensible advanced particle system is not difficult, provided you take time to consider how you will use it within your game, and you carefully design your system architecture accordingly. Because the system I discussed uses classes with inheritance, you can also put the individual particle system types into .DLL files. This opens up the possibility of creating some sort of plug-in system, which might be of interest to some game developers.

You can also download the source code of my particle system, which I have created for Oxygen3D, my latest engine. This source is not a standalone compilable system, but it should help you if you run into any troubles. If you still have any questions or remarks, don't hesitate to send me an e-mail. ■


This electricity has its own render function. A hierarchy tree was constructed to represent the electricity flow using branches and sub-branches. It is a thunderstorm lightning effect with the branches animated. Particle shapes are being constructed for every part in the electricity tree.



A rain effect, using stretched shapes for the particles. The rain also splats on the ground by calling the sparks system with adjusted settings and texture.

LISTING 2. *Update function of the particle manager.*

```
for (int i=0; i<particleSystems.Length())       // traverse all particle systems
{
     if (!particleSystems[i]->Update())         // if the system died, remove it
     {
       delete particleSystems[i];               // release it from memory
       particleSystems.SwapRemove(i);           // remove number i, and fill the gap
                                                // with the last entry in the array
     }
     else
       i++;
}
```