Demo 5: Particle Engine

In this demo we introduce particle effects into the engine. The goal of this demo will be to demonstrate how to develop a particle engine. Although there are many ways to implement a particle engine, many of the concepts are the same. We’ve tried to stick with a basic design, but robustness was also a key factor. Therefore, in this demo you’ll learn the basic concepts of a particle engine, as well as specific details about the design we chose here.

Key Topics

The following topics will be covered in this document:

- Design Overview
  - Billboard Particles
  - XML Structure
- Particle
- Particle Effect
  - Type Defines
  - Effect Properties
  - Initial Particle Values
  - Inner Workings
- Particle System
- Particle Engine
Design Overview

Before we get started, we should go over the general design of this particular particle engine. The particle engine (which we’ll call the engine just for this tutorial) is made up of four tiers of classes. We will cover each of these classes in more detail in the following sections. At the bottom of the hierarchy, we have our particle. Next, we have a particle effect, which is a collection of particles. Next we have a particle system, which is a collection of effects. Finally, we have a particle engine, which is a collection of systems. The engine is also provides the public interface to the programmer.

Billboard Particles

All the particles are billboards which always face the camera. Graphics libraries such as DirectX have simple ways to draw in this manner. This comes with some advantages and disadvantages. One advantage is that only the position and size are required to draw the particle. The library will take care of orienting towards the camera. Related to that is you don’t have to pass in a vertex buffer with the corners, since the library will know how to get the corners of the square from the position and size that you pass in. However, we feel this can be too limiting for some effects. You may not always want square particles (or at least drawn on squares). You may want to use a rectangular billboard, or even skew one of the corners to get an irregular shape. You may also want to rotate the particles; this typically can’t be done using built-in functions. Therefore, since robustness was more a requirement than simplicity, we favored not using the built-in function to draw our particles.

XML Structure

All the particle systems are defined in an .xml file. XML has nice properties that work for us in this way. The engine will look through the xml structure when you request a particle system to be created. If the system name is there, the engine creates a system, passing in the xml structure starting at that point in the file where the system is defined. The system then creates an effect for each effect tag defined in the system tag, passing in to each effect constructor the xml element that defines that effect. Each effect then looks through the effect tag to find the properties defined for that effect.

Particle

The Particle class represents a single particle; it stores everything we need to draw a single particle to the screen. The class has no methods, so it merely stores information.
Position and velocity help us calculate the position of the particle each frame.
Drag is used to affect the movement of the particles. How drag is used will be discussed in the next section, titled Particle Effect.
Size is a relative value since it’s based on the units used in the game.
Color is used in conjunction with the texture to get the final color value.
Lifeleft is an amount of time in seconds that the particle will be drawn. Once lifeleft becomes zero, the particle is thought of as dead. This may not be the end of our particle, however, as the effect may reuse, or rebirth if you will, the particle if it’s called for.
Birthed is a flag that tells us if the particle has been used yet. Once a particle is born, this flag is set to true and will remain true from then on. More will be discussed on this in the following section.
Rotation values are used by the effect to rotate the particle. Rotating the particles can be an effective way of disguising the reuse of textures. For instance, if you have an effect of 100 particles, and they all use the same texture, it can be easy to pick out individual particles by that texture. This may ruin the effect in cases of trying to simulate smoke or fire. By having the particles rotate, you can sometimes get better results.
The uv coordinates let us define separate texture coordinates for each particle. This can also help to break up the repetition caused by reusing the same texture. We assume you will want to use square coordinates, since we only track the left, right, top, and bottom values. However, if you need non-square coordinates, feel free to change the four scalar values to four 2D values (x,y).
The distance value is only used to sort the particles. We calculate the distance and store it in this variable, then use that to put the particles in sorted order based on their distance from the camera.
That’s it for the Particle class. Although robustness was a requirement for the particle engine, we couldn’t think of everything you may want to do with it. However, the design should allow you to add features easily, and doing so may require you to add variables here at the Particle level.

**Particle Effect**

The ParticleEffect class manages a group of similar particles. These particles will have the same basic behavior, making it easy for the effect to manage them. Before we get started looking at the ParticleEffect code, keep in mind that we’re using the STL fairly heavily here, and that the code can look confusing and ugly. Adding to the ugliness, we use function pointers in conjunction with the STL. Don’t be scared. The syntax is easier to grasp than it may at first appear. Let’s start by taking a look at the ugly typedefs we’re using at this level.

**Type Defines**

```cpp
typedef VertexBuffer<RenderVertexL> VertexLBuffer;
typedef Vector3 (*DistributionFunc)();
typedef void (ParticleEffect::*UpdateFunc)();
typedef void (ParticleEffect::*InitFunc)(Particle*);
typedef std::vector<UpdateFunc> UpdateFuncArray;
typedef UpdateFuncArray::const_iterator UpdateFuncIter;
typedef std::vector<InitFunc> InitFuncArray;
typedef InitFuncArray::const_iterator InitFuncIter;
typedef bool (ParticleEffect::*PropertyFunc)(TiXmlElement*);
typedef std::pair <std::string, PropertyFunc> PropPair;
typedef stdext::hash_map<std::string, PropertyFunc> PropertyMap;
```

That’s not so bad. OK, it’s pretty bad, but we cover each of these in detail.

- **VertexLBuffer** – This is just a VertexBuffer that uses lit vertices.
- **DistributionFunc** – This is a function pointer to a function that defines how our particles spread out. We’ve created a few default functions, but feel free to add your own.
- **UpdateFunc** – This is a pointer to a function that will update our particles every frame.
- **InitFunc** – This is a pointer to a function that will initialize our particles whenever they are born or reborn.
- **UpdateFuncArray** – This is an array of update function pointers. This will allow us to use multiple update functions, so we can keep each update function simple and modular.
- **UpdateFuncIter** – This is an iterator or index we can use for an UpdateFuncArray.
- **InitFuncArray** – This is an array of initializing function pointers. Similar to the UpdateFuncArray, this will allow us to use multiple initializing functions, so we can keep each initializing function simple and modular.
- **InitFuncIter** – This is an iterator or index that we can use for an InitFuncArray.
• PropertyFunc – This is a pointer to a function that defines a certain property in the effect. The function takes a TiXmlElement pointer as a parameter.
• PropPair – This is a pairing of a string and a PropertyFunc. We use these pairings to map an element in the particle xml file to a particular property function.
• PropertyMap – This is the map structure that operates on PropPairs.

**Effect Properties**

Now that we have an idea of what these typedefs are, we can proceed to discussing how the ParticleEffect class works. Take a look at the effect properties.

```
Particle *m_Particles;
int *m_DRAWORDER;
int m_nTotalParticleCount;
int m_nLiveParticleCount;
int m_nEmitRate;
float m_fElapsedTime;
float m_fEmitPartial;
bool m_Sort;
bool m_bCycleParticles;
Vector3 m_vecPosition;
Vector3 m_vecGravity;
VertexBuffer *m_VertBuffer;
IndexBuffer *m_IndexBuffer;
bool m_bIsDead;
bool m_IsDying;
int m_textureHandle;
UpdateFuncArray m_UpdateFunc;
InitFuncArray m_InitFunc;
```

• m_Particles – The particles in our effect.
• m_drawOrder – This is an array of indices that determine what order to draw the particles in. The values in this array are indices into the m_Particles array.
• m_nTotalParticleCount – Total number of particles in our effect.
• m_nLiveParticleCount – Number of live particles in our effect. Up to this many values in m_drawOrder are valid.
• m_nEmitRate – Number of particles we spawn per second.
• m_fElapsedTime – The amount of time that has elapsed since last update.
• m_fEmitPartial – Tracks the fraction of a particle that didn’t get spawned each frame. Usually not needed, unless the spawn rate is very slow (less than 1 per frame). The amount we actually spawn each frame is rounded down, so if it’s too slow, no particles would ever get spawned. This value ensures that we still get new particles.
• m_sort – Whether the particles need to be sorted. This is an expensive operation, but sometimes it’s needed to achieve the right look.
• m_bCycleParticles – Whether the particles should be recycled. If this is set to true, the effect will continue on indefinitely.
• m_vecPosition – This is the position of the effect. This is the position from which new particles are spawned.
• m_vecGravity – All particles in an effect are accelerated by this vector, but it doesn’t have to be the same for all effects.
• m_vertBuffer – This is our vertex buffer. We lock and write to this buffer every frame.
• m_indexBuffer – This is our index buffer. We never need to write to this, so it’s a static buffer.
• m_bIsDead – The effect will mark this as true when the last particle has expired. An effect that recycles particles will never die.
• m_IsDying – This is set to true when the last particle has been spawned, but only if the effect doesn’t recycle particles. We test this value before trying to create a new particle, or when trying to determine if the effect is dead.
• m_textureHandle – This is the handle to the texture for the particles. Each effect can only have one texture, but you can set the particle’s texture coordinates in such a way to simulate having multiple textures.
• m_UpdateFunc – Array of update functions to call every frame. We use function pointers to allow for a plug-in sort of feel to the effect.
• m_InitFunc – Array of initializing function to call every time a particle is initialized. We use function pointers to allow for a plug-in sort of feel to the effect.

Initial Particle Values
Now let’s look at the values we use to initialize particles. We store these values when the effect is created, and they remain unchanged throughout the life of the effect.

```c
float m_fPILife;
float m_fPISpeed;
float m_fPISize;
unsigned int m_cPIColor;
float m_fPIDragValue;
float m_PIFadeIn;
float m_PIFadeOut;
float m_PIPadeMax;
float m_PIRotationSpeed;
float m_PIRotationStopTime;
DistributionFunc m_distFunc;
```

• m_fPILife – This value indicates how long the particles will live once they’re spawned.
• m_fPISpeed – This value is the magnitude of the particle’s initial velocity.
• m_fPISize – This is the size of the particle initially. You could add an update function that alters the size of the particles each frame.
• m_cPIColor – This is the particle’s initial color.
• **m_fPIDragValue** – This is the amount of drag each particle will experience initially. The term drag is used loosely here, and is not calculated as true drag would be.

• **m_PIFadeIn** – This value indicates when a particle will have its max alpha value. It can be between zero and one inclusively. The alpha value will increase linearly with time, starting at zero and ending at the max alpha value.

• **m_PIFadeOut** – This value indicates when a particle will begin decreasing its alpha value. It can be between zero and one inclusively. The alpha value will decrease linearly with time, starting at this time and falling to zero the moment the particle dies.

• **m_PIFadeMax** – This is the maximum alpha value the particle will have. It can be between zero and one inclusively.

• **m_PIRotationSpeed** – This is the number of radians per second that the particle will rotate. This can be any value, positive or negative.

• **m_PIRotationStopTime** – This is the time at which the particle stops rotating. This can be used to slow the rotation down over time.

• **m_distFunc** – This is the function used to define the shape that the effect makes. These distribution functions return a vector in a particular distribution that the effect uses to give each particle an initial direction.

### Inner Workings

Now we’re going to discuss how the ParticleEffect class gets the job done. We need to cover what happens when we create an effect and how the effect is updated.

#### Creating the effect

When an effect is created, a number of things happen. In the constructor, we set sensible defaults to all the member variables. We also call three initializing functions, `initProperties()`, `initParticles()`, and `initIndexBuffer()`. The last two, `initIndexBuffer()` and `initParticles()`, are pretty straightforward, and don’t need much explanation. We’ll just cover the first one here.

**initProperties()**

This function takes a pointer to an xml tag containing the effect definition. It loops through each property defined under the tag and finds the name of that property. For each property, the function gets the function pointer associated with that property. Using the mapped function pointer, we execute that property function.

```cpp
ParticlePropertyMapper *mapper = ParticlePropertyMapper::getInstance();
```

These mappings from property names (strings) to function pointers are done in a special class called ParticlePropertyMapper, defined in the first part of the ParticleEffect.cpp file. This class only handles these mappings and nothing else, and is designed to be a singleton. To access this class, you need a pointer to it. You can get this by calling the static method getInstance(). Just remember to not delete it. If you want to add supported
properties to the ParticleEffect class, you’ll need to add the property mappings to this class’s constructor.

**Updating the effect**

Once the effect has been created, the update function will be called once per frame. The update function calls two maintenance functions, `killParticles()` and `birthParticles()`, updates all the particles’ positions, and finally calls the additional update functions specified by the effect definition.

**`killParticles()`/**`killParticle()`

In the function `killParticles()`, we loop through the living particles looking for any that have expired. If a particle has expired, we call `killParticle()` passing in the index of the dead particle. Killing a particle is very easy. We only need to decrement the number of living particles, shift all the particles after the dead one forward one space, and put the dead particle at the back. Since we use an index array to determine draw order, it’s very easy to do the shifting.

**`birthParticles()`**

This function is responsible for initializing new particles each frame. First task is to determine the number of particles that need to be created or initialized. To do this, we take advantage of the way C++ handles integers and floats.

We calculate the number of particles to create by multiplying the emit rate (particles created per second) by the amount of time in seconds that has elapsed since last update. The result is a float, but we don’t emit partial particles. We could just throw away the fraction each frame and not worry about it. This may cause problems with low emit rates though. Consider the following example.

Suppose your emit rate was 30 particles per second, and your game ran at 60 frames per second. This leaves us with creating 0.5 particles every frame when we calculate the number to emit. When we convert this to an integer, it gets rounded down to zero and no particles are created. This is why we need to worry about the fraction.

```c++
// To track partials, we use explicit conversions to get at the partial data.
// Add to what we have already from last time
m_fEmitPartial += (float)m_nEmitRate * m_fElapsedTime;

// Set emit to be the number of complete particles to create
int emit = (int)(m_fEmitPartial);

// Recompute the partial
m_fEmitPartial -= (float)emit;
```

**`initParticle()`**
First this function checks to see if the effect is dying by testing whether the particle being initialized has already been born. If so and if the effect does not recycle particles then it is dying. Otherwise, we initialize the particle to the defined initial values. Finally, we call any additional initializing functions needed. Currently the only additional initializing function we have is initParticleRotation(), but more can easily be added.

Additional Update Functions

We have implemented a couple of additional update functions. We have one to handle changing the alpha values of the particles, and one to handle rotating the particles. Remember that these are optional functions; they’re not necessarily called for every effect, but only for the effects that have these properties defined. We will discuss these kinds of functions in more detail later on.

```c
Particle *part = NULL; // shorthand for the current particle
float maxAlpha = 255.0f * m_PIFadeMax; // precalculate max alpha

for(int i=0; i<=m_nLiveParticleCount; i++)
{
    part = &m_Particles[m_drawOrder[i]];
    int alpha = 0;
    float percentLife = 0;

    // calculate percent of life lived from life left
    percentLife = 1.0f - (part->lifeleft / m_fPILife);

    if(percentLife < m_PIFadeIn) // fade in to max alpha value
    {
        alpha = (int)((percentLife / m_PIFadeIn) * maxAlpha);
    }
    else if(percentLife > m_PIFadeOut) // fade out to alpha value of zero
    {
        alpha = (int)((1.0f - percentLife)/(1.0f - m_PIFadeOut) * maxAlpha);
    }
    else // maintain max alpha value
    {
        alpha = (int)maxAlpha;
    }

    if(alpha < 0)
        alpha = 0;
    else if(alpha > 255)
        alpha = 255;

    part->color = ((part->color & 0x00FFFFFF) | (alpha << 24));
    part++;
}
```

The updateFade() function computes the current alpha value for each particle individually based on that particle’s age. Each particle tracks its remaining life left in seconds, so it’s easy to convert that to life lived as a ratio. Then we test that value to find out if the
particle is fading in, holding steady, or fading out. If the particle is fading in, then the ratio of life lived will be less than the fade in value. Likewise, we test for the particle maintaining alpha value and fading out. Look out the calculation for fading out.

Remember that percentLife is a value from 0 to 1, and represents the ratio of life lived. When that value is 1, the particle should be completely faded out (transparent). To achieve this, we need a function that goes from 1 to 0, starting at the point when the particle should begin fading out, and ending when the particle has lived its entire life. To do this, we use the following function.

\[
\text{alpha} = \frac{(1 - \text{percentLife})}{(1 - \text{m_PIFadeOut})} \times \text{maxAlpha}
\]

Notice that when percentLife is equal to m_PIFadeOut, that alpha is equal to maxAlpha. This happens just as we begin using this function, since this is when the if-clause evaluates to true. As the particle lives on, percentLife continues to grow towards 1. At that point, the top value \((1 - \text{percentLife})\) moves towards 0, but the bottom value remains constant. This gives us a function that steadily decreases from maxAlpha to zero during the period of time that the particle is supposed to be fading out.

```
for(int i=0; i<m_nLiveParticleCount; i++)
{
    Particle *part = &m_Particles[m_drawOrder[i]]; // shorthand
    float angularSpeed =
        part->rotationSpeed * (part->rotationStopTime / m_PIRotationStopTime);
    part->rotation += angularSpeed * m_fElapsedTime;
    part->rotationStopTime -= m_fElapsedTime;
    if(part->rotationStopTime < 0.0f)
        part->rotationStopTime = 0.0f;
}
```

The updateRotation() function is a little simpler. Here we calculate the rotation speed first, then update the rotation of the particle based on that speed. Notice that the rotation will slow down linearly with time until the particle’s rotationStopTime has reached zero.

**Particle System**

A particle system, as we have defined it, is really just a collection of particle effects. The class is only responsible for being an interface into the collection of effects. The idea is that the effects in a system act enough like a single effect, that you can treat them that way on a higher level.

Our ParticleSystem class offers methods to initialize, start, update, and render, as well as set the position of the system. Note that the majority of the interface is private, and only
accessible to the ParticleEngine class. Most of these functions should only be called by
the engine, and we’ll see how that works in the following section. For now, let’s focus on
how the ParticleSystem class manages effects. We’ll look at the important maintenance
functions.

The init() function takes an xml tag defining a group of effects, and creates those effects.
This is done in the following way. First, we count the number of effects defined in the
xml tag. Then we create that number of effect pointers using the new operator. Then we
iterate through the effects defined in the xml tag and create an instance of ParticleEffect
for each, passing in the effect xml tag to the constructor. This gives us all the effects for
the system.

The start() function iterates through the effects calling their start() functions. As we’ve
seen, this initializes all the particles in each effect and sets reasonable starting values.
This should be called once, before any calls to update or render the system.

The update() function iterates through the effects, doing two things for each. First, it
updates the position of each effect. This allows the engine to set the position of the
system, without having to worry about each individual effect. Second, it calls the update()
function of the effect, allowing it to update itself as needed, given the amount of time that
has elapsed.

The render() function iterates through the effects calling their render() function. Are you
starting to see a pattern here? Remember that the ParticleSystem is simply a manager of
effects, allowing us to treat multiple little effects as one big coordinated effect.

**Particle Engine**

The ParticleEngine class gives the programmer an interface into all this particle
goodness. The game engine (SAGE) is built on distinct phases, such as initializing,
updating (processing), rendering, and shutting down. The particle engine offers an
interface that works with the SAGE design. We will look at these functions, as well as an
example of how we used the particle engine in the Ned3D demo.

**Interface**

In the ParticleEngine.h file, you can find the interface of the ParticleEngine class. You’ll
see a series of typedefs used to create short names for several of the structures that rely
on the STL. You should be familiar with how these typedefs are used from the section
above on the Particle class. Looking in ParticleEngine.h file, you’ll see init(), shutdown(),
updateSystems(), and render(). We will take a look at each of these shortly. There are a
few functions which we will not cover, but that are trivial in their implementation. For
example, getSystemName() returns the name of a system belonging to a particular ID.
init()

This function is responsible for getting the engine ready to be used. Here we handle all allocations of systems and initializing of variables. The function takes the name of a file containing the particle engine configuration values. This is a very simple configuration file which currently only needs the name of the particle system definition file. Below is a listing of the file pengine.xml which is the configuration file for our Ned3D demo. Although we only store one value in this file, you could extend its use to include other engine values.

```xml
<?xml version="1.0" encoding="utf-8"?>
<!-- Particle Engine config file -->
<config>
  <deffile value="particle.xml" />
</config>
```

We need to explain how the systems are stored in the engine to better understand what this function is doing. All the particle systems in the engine are created when this function is called, instead of later on when a system is requested. By creating the systems up front, we save time later on, but we may have unnecessary memory allocated until then. Each system has a certain number of copies specified in the system definition file. These systems are created in this function, and initialized for use in the future.

```cpp
while(systemDef != NULL)
{
  std::vector<ParticleSystem*> systems;

  m_Systems.push_back(systems);
  SystemCatalogIter iter = m_Systems.end();
  iter--;
  m_TypeMap.insert(NameTypePair(systemDef->Attribute("name"), numSystemTypes));

  int numCopies;
  systemDef->Attribute("numcopies", &numCopies);

  TiXmlElement* effect = systemDef->FirstChildElement("effect");
  if(effect == NULL)
    ABORT("Invalid file format found while initializing ParticleEngine: filename %s", defFile);

  for(int i=0; i<numCopies; i++)
  {
    ParticleSystem *sys = new ParticleSystem();
    iter->push_back(sys);
    sys->init(systemDef);
  }

  systemDef = systemDef->NextSiblingElement("system");
  numSystemTypes++;
}
To store these particle systems, we use an array of arrays of pointers to particle systems called m_Systems. While we are reading in system tags from the xml file, we put an empty array of system pointers into m_Systems for each system tag. Then we create a new system and store the pointer to that system in the new array. This may seem complicated, but we’ve done it this way to allow for varying numbers of copies of each system.

**shutdown()**

This function deletes all the systems in the engine, and sets values back to default. The bulk of the work is done by calling clear() which actually goes through all the systems and deletes the memory. This function also takes care of removing the mapped ID of the system and releasing the ID from the ID generator. The shutdown() function, besides just calling clear(), releases the xml file and finally asserts that no systems are still mapped in m_UIDMap.

**render()**

Although render() is a short function, there are some interesting things going on here. The engine draws all systems from back to front to help with alpha blending. Although this does not solve the problem of drawing out of order entirely, it helps in most cases. An example of where this method will not help is when two systems are close to each other, for instance one behind the other. The one further away (we will call this one A) is drawn before the other (we will call this one B). However, if they are close enough the particles may mix together. This means that particles from A may have moved forward far enough to be in front of some of the particles from B which have moved backward. To fully solve the problem you could sort each particle that needed to be drawn each frame. Sorting is fundamentally an n*log(n) operation, and the number of particles being drawn per frame can be very large. These factors make this solution prohibitively expensive. But you should feel free to try this or some other solution.

To sort these systems, we use a special set class from the STL. What makes this a special set class is that we have declared our own sorting mechanism for the set to use. Below is the code we used to do this. We create a comparator struct that will be passed into the set class as a template parameter. To create your own comparator, you need only to write an operator for () which takes two elements in the set and returns true if the first element is “smaller” than the second. We use distance to the camera to determine sorted order. Where the distance is the same, we use the value of the IDs to arbitrate.

This is called as elements are inserted into the set. Each frame we insert each active system into the set and all the set to sort the systems for us. Then we render the systems in reverse order, since we sorted with closest systems to the front and we would like a back to front rendering.
The render() function takes a boolean parameter called doUpdate. This parameter allows you to turn on or off updating of the systems before rendering. This is useful for techniques such as rendering water reflections, where you need to render the scene multiple times per frame, since you do not want the systems to update multiple times per frame.

```cpp
struct myCompare
{
    typedef ParticleSystem* P;
    bool operator()(const P& x, const P& y) const
    {
        float xdist = Vector3::distanceSquared(gRenderer.getCameraPos(),
            x->getPosition());
        float ydist = Vector3::distanceSquared(gRenderer.getCameraPos(),
            y->getPosition());

        if(xdist < ydist)
            return true;
        else if(xdist == ydist)
            return x->getUID() < y->getUID();
        else
            return false;
    }
};
typedef std::set<ParticleSystem*, myCompare> SystemSortedSet;
```

**updateSystems()**

This function is fairly trivial; it loops through all the mapped systems (currently alive) and calls update. The only points of interest are that we move the iterator to the next system before trying to update the current system, and that we test if the system is dead at this point and kill it if so. The first is done in case of the second, meaning if we need to kill the current system, we need to go ahead and move to the next system in the array before doing so. This is because killing the system will remove that system from the array on which we are operating.

**Supported Effect Properties**

We have implemented support for several properties that are basic to any system, and a few that are not. The basic properties are there just for functionality, but the extended ones have a two-fold purpose. First, we wanted an example of extended functionality to give the reader. Second, we needed specific functionality for our crow feathers that was not supported in the engine. We will cover the basic properties in this section and the extended functionality in the next section.

The effect definition file is an xml file that describes the properties of the effects that make up each system. We have already discussed the format of the file, but now we want to look at some of the data found in the file. You may want to look over the particle.xml
file before going any further to get an idea of what we will be covering. In particular notice the data inside the effect tag, including the attributes of the tag as well as the properties nested inside the tag.

The properties defined inside the effect give information in the form of data for variables or functionality. By convention, the properties with the prefix of particle- in the name are properties specific to the particles in the effect. Otherwise they apply to the entire effect. This is only by convention however, and you will see that you can choose whatever you want to name your properties.

**Built-in Properties**

This section describes the built-in properties supported by the ParticleEffect class.

**emit**

This is the rate at which particles are spawned and the overall distribution of the particles in the effect. The rate value is in particles per second. For example, a rate value of 100 means 100 particles will be spawned every second. If your framerate is 50 frames per second, then two particles will be spawned every frame. The different shapes supported are shellsphere, solidsphere, ring, disc, and solidcube. The functions used to generate these distributions can be found in ParticleDefines.cpp and are described briefly in the following.

- **shellsphere** – This distribution returns a uniformly distributed random vector on a unit sphere.
- **solidsphere** – This distribution returns a random vector within a unit sphere. This function uses the shell sphere function result and multiplies it by a random value between zero and one. However, this does not result in a uniform distribution throughout the sphere, but instead the vectors tend towards the sphere’s origin. In most cases this is unnoticeable.
- **ring** – This distribution returns a uniformly distributed random vector on a unit circle in the x,z plane.
- **disc** – This distribution returns a uniformly distributed random vector within a unit circle in the x,z plane. The method used here works, but is not the fastest possible solution. This method continues picking points inside a unit square in the x,z plane until it finds one that is also inside the unit circle.
- **solidcube** – This distribution returns a uniformly distributed random vector within a unit cube.

**sort**

This determines whether the effect tries to sort the particles from back to front. If you are noticing visual artifacts that look like they may be related to draw order, try setting this value to one (1). Sorting can be an expensive operation, so by default sorting is turned off. We use a simple bubble sort to sort the particles and the reason why is explained in the comments in the ParticleEffect::sort() function.
**gravity**

Somewhat a misnomer, this determines the general direction of acceleration of the particles in the effect. You don’t have to use it as gravity. For example you can set it to accelerate the particles upward for a smoke effect. The values are the x, y, and z parameters of the acceleration direction.

**cycle**

This determines whether the effect will continue forever, or die once all the particles have been spawned and killed. If the effect cycles, it we reuse the particles after they die over and over. An effect that cycles will only stop when the program has requested the system containing the effect to be killed explicitly.

**particlelife**

This is the amount of time each particle will live in seconds. When each particle is spawned, its lifeleft value is initialized to this value.

**particlespeed**

This is the magnitude of the initial velocity of each particle. By default, this value is multiplied by the particle’s initial direction to get its initial velocity.

**particlecolor**

This sets the vertex color of the particles. The effect multiplies the texture color by the vertex color to get the final color (using render states.) Generally you will want the texture to completely white, varying only the alpha values to get the shape of the particle in the texture. Then the final color will be whatever value you put in the effect definition.

For example, assume you have a opaque white texture and a particlecolor value of \{255, 255, 100, 100\}. The elements in the particlecolor value represent the alpha, red, green, and blue channels, and can be values between 0-255. This particular color is a reddish color. To calculate the final color of the particle, first you need to convert these channel values to floats between 0.0f – 1.0f by dividing each by 255. The converted value is \{1.0f, 1.0f, 0.39f, 0.39f\}. This new color gets multiplied by the texture color on a per channel basis. The texture is \{1.0f, 1.0f, 1.0f, 1.0f\} (assumed opaque white texture) so the final outcome of the particle is \{1.0f, 1.0f, 0.39f, 0.39f\}. Notice that if parts of the texture were transparent (alpha of 0.0f) the corresponding part of the particle would be transparent since the texture coordinates of the vertices that make up the particle map to the corners of the texture.

**particlesize**

This is the relative size of each particle in the effect. Usually you will have to experiment with this value until the effect looks right, since the units vary from game to game.

**particledrag**

This is the relative amount of decrease in speed of the particle over time. This is far from a physically correct calculation involving drag, but suffices to have the particles slow to
halt after being spawned. We treat drag like acceleration. So to update the velocity, we calculate all the forces acting on the particle, which include gravity and drag. The drag force is the drag coefficient times the current velocity, but in the opposite direction of the current velocity. The final equation looks like

\[ v' = (g - v\times d) \times dt \]

where \( v' \) is the new velocity, \( g \) is the gravity, \( v \) is the old velocity, \( d \) is the drag coefficient, and \( dt \) is the change in time. Again, this is a bad approximation of drag, but it’s easy to conceptualize. The higher the drag value the faster the particles will slow down.

**Extending the Particle Engine**

By this point, you may have some interesting ideas for effects in your own game, and you are wondering how to modify the engine to pull them off. This section will talk about one of the two extended properties that we included to give you an idea of how to extend the engine yourself. These properties are particlefade, which controls the fading in and out of particles, and particlerotation, which controls the rotation of the particles to make them look like they are spinning about their z-axis.

**particlefade**

This property will allow each particle to fade in and out based on its age. It will allow for a smoother transition visually of the particle being spawned and of the particle dying. This section will discuss how we extended the engine to handle this functionality.

**Decide on New Variables**

We could use a single offset to control this. For example, let \( c \) be the amount of time in seconds to fade in and out. Then during the first \( c \) secs the particle would fade in, and during the last \( c \) seconds the particle would fade out. However, what if you wanted the particles to fade in quickly and fade out very slowly? Planning for this scenario, which is the more general case, we could use two values to control the amount of time to fade in and out independent of each other.

Now we have two values we want to add, but there is still another thing to consider before we start looking at the code. The amount of transparency may need to be controlled. We keep saying we want the particles to fade in and out, but we may want to control to what extent this happens. In most cases, when particles fade out they should fade out completely. This means that each particle’s alpha should be zero by the time it dies. This can be handled trivially, and doesn’t require an extra variable since we are assuming that all particles should fade out completely. We will want an extra variable to control how much the particle fades in. With most particle effects, the particles are intended to always be semi-transparent throughout the life of the effect. This translates to an alpha value of less than one (remember that 1.0f alpha is fully opaque).
We have three variables we want to add to the effect class to be able to fully implement the fading in and out of particles, which are fade-in time, fade-out time, and maximum alpha. The ParticleEffect class will need to track these variables, so adding them to the class definition would be a good place to start. We have added m_PIFadeIn, m_PIFadeOut, and m_PIFadeMax. The m_PIFadeIn and m_PIFadeOut variables will be values between zero and one, and will represent ratios of life left. The m_PIFadeMax variable is also between zero and one, and will represent a maximum alpha value. For example, if this is set to 0.5f then the particle’s alpha will be at most 0.5f.

```cpp
float m_PIFadeIn; // How long until the particle is at maximum alpha
float m_PIFadeOut; // How long before it begins to fade to 0 alpha
float m_PIFadeMax; // Maximum alpha
```

**Handle the XML data**

Ultimately we should be able to define and manipulate the specific values of the effects from the xml file. To do this, we need to create a function to handle this property and make sure it gets called when the effect sees the property while parsing through the effect definition. We will write the function first, and then we will show how it fits into the effect parsing.

Before we write the xml function, we should discuss the xml functions in general. All the functions that parse xml have the same return type and take the same parameter. Why this is will become clear shortly. The return type to these functions is a bool which should indicate whether the property was loaded properly. These functions take a pointer to a TiXmlElement which is the xml tag containing the property data. From this pointer our function will get all the property data set by the xml file. One possibility you may need to consider is what happens if not all the data is there, or the data is invalid. We have not coded this to check rigorously for missing or invalid data, but instead have relied upon the developers to ensure proper data. This was mainly because that kind of code is lengthy and makes understanding the function more difficult.

Our function only needs to read in three values from the xml data, and store them into our newly created class variables. In the xml file, we will call these values fadein, fadeout, and fademax. We need to translate those values into m_PIFadeIn, m_PIFadeOut, m_PIFadeMax respectively. Below is a listing of the code we used to do just that. We test for the presence of each attribute in the tag separately. If the attribute is there, we store its value into our class variable.
bool ParticleEffect::setParticleFade(TiXmlElement *prop)
{
    if(prop->Attribute("fadein") != NULL)
    {
        double tmp;
        prop->Attribute("fadein", &tmp);
        m_PIFadeIn = (float)tmp;
    }

    if(prop->Attribute("fadeout") != NULL)
    {
        double tmp;
        prop->Attribute("fadeout", &tmp);
        m_PIFadeOut = (float)tmp;
    }

    if(prop->Attribute("fademax") != NULL)
    {
        double tmp;
        prop->Attribute("fademax", &tmp);
        m_PIFadeMax = (float)tmp;
    }

    return true;
}

Now we just need for this function to get called when this kind of property is found in the xml file. To do this, we need a mapping of the string name of the property, which we will call particlefade, to this function. At the beginning of the ParticleEffect.cpp file, we have defined a class called ParticlePropertyMapper. The sole purpose of this class is to handle these mappings that we need. To add our new function to the map, we will add a single line of code to the constructor of this mapping class (shown below.) At this point, our function will be called when the effect object reaches a property tag named particlefade.

```
    m_propertyMap.insert( ParticleEffect::PropPair( "particlefade",
        &ParticleEffect::setParticleFade));
```

**Write the Utility Functions**

We still need to do something with this data that we have read from the xml file, and for that we will need a second function. In this particular case, we will only need an update function that is called every frame. However, in many cases you will also want a function to initialize some data in the particle, and so you would need a third function that is only called once per particle initialization.

We will call the function that updates our fade values updateFade(), a name that seems to be appropriate given what the function will do. For an explanation of how this function works, refer to the previous section (Particle Effect) which discusses how this particular function works. We still need to make sure this function gets called during every update.
Using function pointers again, we can add a pointer to this function to our list of update function pointers. Let us revisit the setParticleFade() function to add one small detail now that we have an appropriate update function to call. Just before we return a successful operation, we need to push a pointer to our new update function onto the list. Below is the single line of code we add to do just that.

```cpp
m_UpdateFunc.push_back(&ParticleEffect::updateFade);
```

At this point, everything is in place to have our particles fade in and out of existence. If your idea requires an initializing function, you would need to add one more line of code to the xml reading function which would push the pointer to your initializing function onto the initializing function pointer list.

We would like to point out that if you need to store data at the particle level to implement your idea for an effect, feel free to modify the Particle class. The particle rotation property does this, so you can look at that as a reference. Also, the update and initializing function pointers are pushed onto their respective lists in the order in which they are read from the xml file. If the order in which these functions are called is an issue, you will need to figure out a way to ensure that order. We assumed that the order would not be an issue.