SPARK Tutorial 1

General functioning of SPARK and first particle system

In this first tutorial, we will learn to set up a particle system with SPARK and to correctly configure its elements.

The particle system we will build is a basic fireworks. Each time the user presses space, a blast will be created at a random location. The color of each blast will be randomly generated as well.

To do that, we will use the SDL as the windowing library. It will give us an OpenGL context to render our particles on. Thus, the renderers we will use will be the SPARK OpenGL renderers.

I - Structure of the system

The easiest, the more intuitive and the more robust way to create a type of particle system (**SPK::System**) using SPARK is to define a base system and to register it in the SPARK factory (**SPK::SPKFactory**). Each time we will want to create a system of this type, we will have to simply ask the factory to create a new instance of the base system.

Each registerable object has a static method **create**, taking the same parameters as its constructor, that allows to create a new instance of the object and directly registers it into the factory.

The active instances will be held in a container to allow to update them, render them on screen, to add or remove some, to acess them...

In SPARK, a system holds pointers on groups (**SPK::Group**). Those groups contain the particles and defines the rules to create and update them. SPARK uses mainly a semantic of entities to link the classes to one another. Thus, the instances are not necessarily linked to others (composition) but can be shared (association).

In our case, our system will contain only one group which will be the blast. But we can imagine more complex systems holding several groups (a fire with flames and smoke for instance).

II - Initialization

To use SPARK in our program, we must include the needed libraries. In our case, we need to include the core of SPARK and the OpenGL renderers :

```
#include <SPK.h>
#include <SPK_GL.h>
```

We must priorly link the needed libraries either statically or dynamically. In the case of dynamically linking, SPK_IMPORT must be defined as a preprocessor macro. The method to link SPARK to the application will not be detailled here because it is dependant on the IDE/compiler used. However, the linking is classical.

SPARK uses namespaces by libraries. We will use them to avoid prefixing every calls :

```
using namespace SPK;
using namespace SPK::GL;
```

Moreover, SPARK uses a custom optimized pseudo generator. The seed must be initialized at the beginning of the program. We will initialize it with the processor time :

```
randomSeed = static_cast<unsigned int>(time(NULL));
```

III – Particle model

In SPARK, every group holds a pointer to a particle model (**SPK::Model**). A group is created with this pointer and it cannot be modified afterwards. A group keeps the same model all along its life. This is compulsary as the model defines the way particle data is organized internally. However, a model can be modified in real time and shared between several groups.

A model is used to activate or not the particle parameters. Those parameters are for instance the size, the mass, the color, the angle... and are defined by a floating point value. The activation and the way they evolve are defined with flags. There are 3 flags :

- *The enable flag* : It defines which parameters will be stored by individual particles. If a parameter is not enabled and the engines needs it, its default value will be used.
- *The mutable flag* : It defines which parameters will evolve linearly during the particle life. The mutable parameters have 2 values : the birth and the death ones.
- *The random flag* : It defines which parameters will see their value(s) randomly generated. The random parameters have 2 values : a minimal and a maximal one.

A mutable or random parameter must be enabled. Only the enabled parameters can be mutable or random. A parameter can also be both mutable and random. This means both the birth and death value are randomly generated. Those kinds of parameters have therefore 4 values.

In our example, here is how we will organize the particle model for our base system :

```
Model* model = Model::create
(
	FLAG_RED | FLAG_GREEN | FLAG_BLUE | FLAG_ALPHA,
	FLAG_ALPHA,
	FLAG_RED | FLAG_GREEN | FLAG_BLUE
);
```

- The first parameter of the model constructor/ create method is the *enable flag*. We activate the 4 parameters of the color components (red, blue, green and alpha). Note that the red, blue and green are always enabled even if not specified for optimization concerns. Those can therefore be omitted.
- The second parameter is the *mutable flag*. We want our particles to fade as they get older. We therefore make the alpha parameter mutable so that it will automatically evolve through particle life.
- Finally, the third parameter of the constructor is the *random flag*. We activate the color components because we want the particles to have some light color variations within a system to give a better look to our blasts.

Once the flags are defined, we have to specify the parameters values of the model :

```
model->setParam(PARAM_ALPHA,1.0f,0.0f);
model->setLifeTime(1.0f,2.0f);
```

The method **SPK::Model::setParam** exists in 3 version : with 1,2 or 4 arguments. The version with 1 argument is use to specify the values of only enabled parameters, the one with 2 for either mutable **or** random parameters, and finally the one with 4 parameters for both mutable **and** enable parameters.

Here, the alpha parameter is mutable. This means the first value defines its value at the birth of the particle, while the second, at the death of the particle. We want to obtain a linear fade, the first value is therefore set to 1 (opaque) and the second to 0 (transparent). At each particle update, the alpha value will be linearly interpolated by the engine function of the particle age.

We don't specify the values of the other enabled parameters (the color components) because they will be set at the creation of each system.

The particle model is also responsible for the lifetime of the particles. This lifetime is defined at the particle birth by generating a random values between 2 thresholds defined by **SPK::Model::setLifeTime(float min,float max)**.

IV – Particle emitter

An emitter (**SPK::Emitter**) alows to generate particles with a given frequency, position and direction in the universe. Many types of emitters exist in SPARK and it is also possible to create custom types (like many classes in SPARK actually).

In SPARK, an emitter holds a pointer on a zone (**SPK::Zone)** which defines a shape and a location in the universe. When generating a particle, the emitter zone is used to generate its position while the emitter itself generates its initial speed vector.

In our example, we want to obtain a blast from a single point. We will therefore use a random emitter (**SPK::RandomEmitter**) which is the most simple one : particles will be emitted in a random direction. Regarding the zone, we will logically use a point (**SPK::Point**) :

```
// Creates the zone
Point* source = Point::create();

// Creates the emitter
RandomEmitter* emitter = RandomEmitter::create();
emitter->setZone(source);
emitter->setForce(2.8f,3.2f);
emitter->setTank(500);
emitter->setFlow(-1);
```

Source is the emitter zone. It is linked to the emitter thanks to the call to **SPK::Emitter::setZone(Zone*)**. We do not specify the initial position of the point for the base system as it will be defined at the creation of each system.

We have defined a tank of 500 particles for the emitter. This means it will only be able to generate 500 particles (until the tank is filled up). The emitter flow defines the emission rate by unit of time. A negative value means every particles of the tank must be emitted at the first update (corresponds to an infinite flow). A negative value for the tank also means it is infinite, it will never be empty. Note that an emitter with both an inifinite flow and tank is invalid. Here, we have set our tank to 500 and our flow to -1. It means the emitter will emit 500 particles in a row at the first update and then become empty, which corresponds to the desired behavior of a blast.

The force of the emitter defines the force at which particles will be emitted. It allows to act on the emission speed. The emission speed is : speed = force / mass. We set a small delta here between the min and the max to get a natural effect.

V – Renderer

The renderer (**SPK::Renderer**) is the objet responsible for the rendering of particles. Indeed, without renderer, a particle set is only a big array of values evolving along time. The renderer interprets those values to give a graphical representation of a particle group.

SPARK gives several renderers to use for specific rendering. The user can also create his own renderers. All renderers do not interpret every particle parameters. For instance an OpenGL point renderer will ignore the **SPK::PARAM_ANGLE** parameter of a particle.

In our example, we will use the OpenGL point renderer (**SPK::GL::GLPointRenderer**) because it allows to render particles with textured quads facing the camera in an optimized way :

```
GLPointRenderer* renderer = GLPointRenderer::create();
renderer->setType(POINT_SPRITE);
renderer->enableWorldSize(true);
GLPointRenderer::setPixelPerUnit(45.0f * 3.14159f / 180.f,screenHeight);
renderer->setSize(0.1f);
renderer->setTexture(textureIndex);
```

The type **SPK::GL::POINT_SPRITE** is the type of OpenGL point that allows to display a texture instead of a single color. **SPK::GL::GLPointRenderer::enableWorldSize(bool)** allows to define if the points are rendered in pixels (thus with a unique size) or in universe coordinates. Here we chose the universe coordinates so that the size of a particle depends on its distance from the camera.

To allow the convertion from screen coordinates to universe coordinates, it is necessary to set certain static parameters allowing to compute it. We therefore pass the angle of field of view in y (in radians) and the screen height in pixels to the static method **SPK::GL::GLExtInterface::setPixelPerUnit(float,unsigned int)**.

The size of a particle in the universe is then defined (This renderer ignores the model parameter **SPK::PARAM_SIZE**) and so is the texture to use. The value of the texture index corresponds to the index of the OpenGL texture.

We will them parametrize the blending. For our particles, we want an additive blending (which is perfect to simulate particles of light). We will also set the texture blending so that it modulates with the particle color.

```
renderer->setBlendingFunctions(BLENDING_ADD);
renderer->setTextureBlending(GL_MODULATE);
renderer->enableRenderingHint(DEPTH_WRITE,false);
```

The last line specifies not to use the depth write, which means particles will not be written to the z buffer but the z test will be performed anyway.

Our renderer is now configured. The only problem is that it uses OpenGL extensions and if these extensions are not supported on a platform the results will not be those desired (However the programm will not crash). Even though only old graphic cards do not support those extensions, to offer high compatibility we will implement an alternative by using

another OpenGL renderer that will guarantee compatibility with OpenGL 1.1.

SPARK handles OpenGL extensions automatically via a class containing static methods : **SPK::GLExtInterface**. Our renderer above needs 2 OpenGL extensions :

- The *PointSprite* extension that allows to attach a texture to an OpenGL point. If not supported, the texture will be ignored
- The *PointParameter* extension that allows to modulate the size of an OpenGL point function of its distance from the camera. If not supported, all particles will have the same size on screen no matter their distance from the camera.

If at least one of the two extensions is not supported, we will use the OpenGL quad renderer (**SPK::GL::GLQuadRenderer**) which is a lot more configurable but less optimized in our case, otherwise we will keep the OpenGL point renderer.

So, here is the final code to create the renderer which I willl not detailled more but which allows to have a renderer full compatible and optimized for a given plateform :

```
GLRenderer* renderer = NULL;

// Tests whether needed extensions are supported
if
((GLPointRenderer::loadGLExtPointSprite())&&(GLPointRenderer::loadGLExtPointParameter()))
{
	GLPointRenderer* pointRenderer = GLPointRenderer::create();
	pointRenderer->setType(POINT_SPRITE);
	pointRenderer->enableWorldSize(true);
	GLPointRenderer::setPixelPerUnit(45.0f * 3.14159f / 180.f,screenHeight);
	pointRenderer->setSize(0.1f);
	pointRenderer->setTexture(textureIndex);
	renderer = pointRenderer;
}
else
{
	GLQuadRenderer* quadRenderer = GLQuadRenderer::create();
	quadRenderer->setTexturingMode(TEXTURE_2D);
	quadRenderer->setScale(0.1f,0.1f);
	quadRenderer->setTexture(textureIndex);
	renderer = quadRenderer;
}

// Sets the blending
renderer->setBlendingFunctions(BLENDING_ADD);
renderer->setTextureBlending(GL_MODULATE);
renderer->enableRenderingHint(DEPTH_WRITE,false);
```

Renderer points to the right renderer and will b euse by our particle group.

VI – Particle group

Now we have created and configured a model, an emitter and a renderer, we can create our particle group :

```
Group* group = Group::create(model,500);
```

The group is created with a model and a capacity. The capacity defines the maximum number of particles the group can hold in real time. As the emitter will emit 500 particles at the first update and right after become inactive, the maximum number of particles will therefore be 500.

We then need to attach our emitter and renderer to the group :

```
group->addEmitter(emitter);
group->setRenderer(renderer);
```

In SPARK, a class called modifier (**SPK::Modifier**) allow to modify the behavior of a particle group in real time depending on internal and/or external parameters (Modifiers will not be detailled in this tutorial). To ease the implementation of a particle system, a group contains 2 physics parameters often used to impact on the particles behavior :

- The gravity which is a force that pushes a particle in a given direction and its action does not depend on the particle mass
- The friction which is a force created by the resistance of a fluid on an object moving within it. This force is goes against the displacement of a particle and depends on the particle velocity.

In our example we will use those 2 parameters to give a more natural effect to the particles motion in our fireworks :

```
group->setGravity(Vector3D(0.0f,-1.0f,0.0f));
group->setFriction(2.0f);
```

We will eventually attach our group to our system. Note that in our case, using a group might not be necessary because we have only one group to handle and systems were designed to handle several particle groups at a time. Most of the SPK::System interface actually also exists in SPK::Group. We will use a system anyway because it is the top class to be handle by the user.

```
System* system = System::create();
system->addGroup(group);
```

VII – The base system

Our base system is now correctly configured. Moreover, as create static methods were used, the system and all its components are registered within the factory. The factory can registers any type of registerable objects (**SPK::Registerable**).

The factory will thus allow to correctly copy registered objects. Indeed, without using the factory, if a standard copy is performed on a system for instance, only the address of the group will be copied and not the group itself. If one wants the group to be copied, one has to code the correct implementation. With the factory, everything is automatic.

To go further, the user has the possibily to define which children will be copied and which children will have only their address copied during a parent copy. In the second case, we speak of shared registered objects. By default, a

registerable object is not shared. A shared object is less memory expensive and saves copy times. It also allows the modification of a parameter which is common to several systems.

In our example, we will make the renderer and the model shared. To do that, we only have to set them as being shared :

```
model->setShared(true);
renderer->setShared(true);
```

Each time we will copy the base system to get a new system, the model and the renderer will not be copied but those belonging to the base system will be used. After having registered our base system, Each instance registered in the factory is given an ID which unically identifies it :

```
SPK_ID BaseSystemID = NO_ID;
BaseSystemID = system->getID();
```

Our base system is registered in the factory and we have an ID identifying it to allow its copy.

The whole creation process of the base system is performed in a function **createParticleSystemBase**. This function returns the ID of the base system.

VIII – Creation of systems

As seen before, we will use the factory to create new blasts from the base system. To do that, a function **createParticleSystem** is created. Its arguments are a position and a color and it returns a pointer on the newly creqty system :

```
System* createParticleSystem(const Vector3D& pos,const Vector3D& color)
{
	System* system = SPK_Copy(System,BaseSystemID);

	Model* model = system->getGroup(0)->getModel();
	model->setParam(PARAM_RED,max(0.0f,color.x - 0.25f),min(1.0f,color.x + 0.25f));
	model->setParam(PARAM_GREEN,max(0.0f,color.y - 0.25f),min(1.0f,color.y + 0.25f));
	model->setParam(PARAM_BLUE,max(0.0f,color.z - 0.25f),min(1.0f,color.z + 0.25f));

	Zone* zone = system->getGroup(0)->getEmitter(0)->getZone();
	zone->setPosition(pos);

	return system;
}
```

The color components of the model and the position of the emitter zone are defined by the arguments of the function. A random is added to the color to get a slight variation in the particles color within the same system but keeping a main dominant color for the explosion.

This function is called when space is pressed. The position is randomly generated within a box and the color is generated in HSV with a random hue over all the spectrum before being converted in RGB.

The system returned by the function is stored in a STL container of type *deque* as the behavior is FIFO (first in first out)

IX – Update et rendering of systems

All our active systems are stored in a container. To update them, a call to the method **SPK::System::update(float)** sequentially is enough :

```
deque<System*>::const_iterator it;
for (it = particleSystems.begin(); it != particleSystems.end(); ++it)
      (*it)->update(deltaTime * 0.001f);
```

The parameter of the method is the time to update the system of.

To draw a system, **SPK::System::render()** is called in the same manner :

```
deque<System*>::const_iterator it;
for (it = particleSystems.begin(); it != particleSystems.end(); ++it)
      (*it)->render();
```

X – Destruction of systems

A registered object must be destroyed by a specific call. This call will destroy the object and all its children recursively which have no more other references than in the parent to detroy. The factory keeps a counter of references for its registered object. The number of references corresponds to the number of time the address of the object exist in the whole set of the registered objects in the factory. We write a function performing the destruction of a system :

```
void destroyParticleSystem(System*& system)
{
      SPK_Destroy(system);
      system = NULL;
}
```

Note that shared objects (in our case the model and the renderer) will not be destroyed with the system as there is at least another reference to it in the base system (and in some other possibly active system).

Note that an registered object must not be destroyed by a call to delete !

SPARK offers a way to know if a system or a group is inactive. A group is considered as inactive if and only if :

- It has no more active particles
- It has no more active emitters. An emitter is active if its tank is not empty and its flow is different from 0.

A particle system is inactive when all its groups are inactive.

The method **SPK::System::update(float)** returns a boolean specifying whether or not the system is active. We therefore only have to check the return value of the update function and destroy the system if it is false. Our new update loop looks as followed :

```
deque<System*>::iterator it = particleSystems.begin();
while(it != particleSystems.end())
{
```

```
	if (!(*it)->update(deltaTime * 0.001f))
	{
		destroyParticleSystem(*it);
		it = particleSystems.erase(it);
	}
	else
		++it;
}
```

Hence as soon a system becomes inactive, it is destroyed.
Finally, once finished with the base system, we need to destroy it as well (in our case it will be at the end of the program). We can either destroy it the same way as the other systems but the factory also has a method allowing to destroy all the registered objects. We will call this method at the end of the program :

```
SPKFactory::getInstance().destroyAll();
```

Note that this method has no associated macro to lighten the syntax of the call.

XI – Conclusion

In this tutoral, we learnt how to use SPARK to set up a basic particle system. The possibilities of SPARK are however larger than this brief overview and I invite you to take a look at the reference documentation and at the demo applications to use non seen features (rendering optimisation, other renderers, emitters, zones, the modifiers, the callback functions, transforms, base class derivation...)