SPARK Tutorial 2

Using SPARK with SFML

In this second tutorial, we will learn to integrate particle systems in a universe created with the SFML 1.5 2D engine (Simple and Fast Multimedia Library).

To do that, we will use a pseudo-game context coded for the occasion. Our application will make 4 cars move randomly in a sand environment. The cars will collide with each other and with the borders. The view zill be in pseudo perspective.

In this small application, we will integrate 2 types of particle based effects :

- the sand cloud generated by the displacement of cars over the sand
- the sparks due to a collision

Note that the points already seen in the first tutorial will not be covered here.

I – Systems structure

As we will integrate 2 types of effects, we will build 2 different particle system bases. Those bases, registered in the factory, will be used to build instances of particle systems when needed.

In this application, a class Car has been implemented to handle the cars. The particle system representing the sand cloud will therefore naturally be integrated to this class. Regarding the sparks happening during collisions, they will be handled directly in the main code of the application.

The sand cloud system will be composed of a single group with 2 emitters (on by rear tyre). Those emitters will generate smoke particles (which will be textured quads) in a continuous way. Their flow will be function of the car speed.

The spark system will be very similar to the one seen in the first tutorial to render fireworks : A single group with an emitter that will generate a certain number of particles when created. The system will die when all its particles are dead. The renderer used will be the SFML line renderer.

II - Initialization

Using SPARK with SFML is about the same as using it without except the SFML module has to be included :

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

Regarding the library linking, the SFML module must be linked. The openGL module must be linked as well as the SFML module calls some of its functions/classes. SPARK is either linked dynamically or statically with the SFML (depending on the SPARK libraries used being the dynamic or the static ones).

The namespace of the SFML module is the following :

```
using namespace SPK::SFML;
```

III – SFML module specificities

SFML Renderers

They are currently 4 renderers for the SFML :

- **SPK::SFML::SFMLQuadRenderer** allows to render particles using textured (or not) quads
- **SPK::SFML::SFMLPointRenderer** uses the openGL primitive point to render particles. The points can either be squares, circles or textured (point sprites).
- **SPK::SFML::SFMLLineRenderer** uses the openGL primitive line to render particles. The lines are aligned through the particle direction and their length depends on the particle speed.
- **SPK::SFML::SFMLDrawableRenderer** allows to render particles using SFML primitives. Note that this renderer is the less optimized as it uses directly the SFML engine which was not designed to handle a large amount of particles.

SFML renderers are all 2D. However calculus are still performed within SPARK in 3D. The user has the possibility to use the Z coordinates in the rendering. Z coordinates are multiplied by a factor (defined with **SPK::SFML::SFMLRenderer::setZFactor(float)**) and then added to the Y coordinates. This allows to modify the particles Y function of their altitude (Z). However, 3D rendering in a 2D scene function of an angle of orientation is not possible yet. A patch shall be released to do it later on. In our applicationm we will use a Z factor of 1.0 to give some relief to our systems. Note that it is possible to use SPARK in pure 2D by setting the Z factor to 0 (its default value) and using vectors only with there 2 first parameters (skipping the Z sets it to 0).

Renderers also have a function to cull the particles with the ground. If it is enabled, all particles with a z inferior to 0 will not be renderered. The ground culling can be set with **SPK::SFML::SFMLRenderer::setGroundCulling(bool)**.

SFML System

The class **SPK::SFML::SFMLSystem** is the base class for all particle systems in SFML. It is derived from both **SPK::System** and **sf::Drawable**. A particle system is, in that way, considered the same as any other Drawable and can be transforned using the SFML methods. The blending and the color are however imposed by the renderers. Values of the system are overridden. This is due to the fact that several renderers with different bleding modes can belong to the same system.

In SFML, every Drawable needs a **sf::RenderTarget** to be renderered. Rendering SFML systems can be performed in 2 ways :

- *SFML style* :

As any other Drawable :

```
myRenderTarget.Draw(mySFMLSystem);
```

- *SPARK style* :

A renderTarget must first be bound to the system :

```
mySFMLSystem.setRenderTarget(&myRenderTarget);
```

Then the system is rendered ina classical way :

```
mySFMLSystem.render();
```

Regarding the positionning of particles in the universe, 2 modes are provided :

- In local coordinates : emitted particles are transformed with the system. This means if the system moves or rotates, particles will do the same.
- In world coordinates : Particles are emitted directly in the world. If the system moves or rotates, only its emitters and zones will be transformed but not the emitted particles

Most of the time, emission in world coordinates will be needed. The choice of the coordinates mode is done when constructing the system and cannot be changed afterwards. By default, systems are in world coordinates.

IV First system : the sparks

We gonna start by integrating the spark particles because the system will be very similar to the one from the first tutorial. As in the first tutorial, we will first create a base system, register it in the SPARK factory and eventually copy it when needed (each time a collision happens). We therefore create a function to initialize our base system :

```
// Creates and register the base particle system for collisions
SPK_ID createParticleCollisionSystemBase()
{
	// Creates the model
	Model* sparkModel = Model::create(
		FLAG_RED | FLAG_GREEN | FLAG_BLUE | FLAG_ALPHA,
		FLAG_ALPHA,
		FLAG_GREEN | FLAG_BLUE);
	sparkModel->setParam(PARAM_RED,1.0f);
	sparkModel->setParam(PARAM_GREEN,0.2f,1.0f);
	sparkModel->setParam(PARAM_BLUE,0.0f,0.2f);
	sparkModel->setParam(PARAM_ALPHA,0.8f,0.0f);
	sparkModel->setLifeTime(0.2f,0.6f);

	// Creates the renderer
	SFMLLineRenderer* sparkRenderer = SFMLLineRenderer::create(0.1f,1.0f);
	sparkRenderer->setBlendMode(sf::Blend::Add);
	sparkRenderer->setGroundCulling(true);

	// Creates the zone
	Sphere* sparkSource = Sphere::create(Vector3D(0.0f,0.0f,10.0f),5.0f);

	// Creates the emitter
	SphericEmitter* sparkEmitter =
SphericEmitter::create(Vector3D(0.0f,0.0f,1.0f),3.14159f / 4.0f,3.0f * 3.14159f / 4.0f);
	sparkEmitter->setForce(50.0f,150.0f);
	sparkEmitter->setZone(sparkSource);
```

```
	sparkEmitter->setTank(25);
	sparkEmitter->setFlow(-1);

	// Creates the Group
	Group* sparkGroup = Group::create(sparkModel,25);
	sparkGroup->setRenderer(sparkRenderer);
	sparkGroup->addEmitter(sparkEmitter);
	sparkGroup->setGravity(Vector3D(0.0f,0.0f,-200.0f));
	sparkGroup->setFriction(2.0f);

	// Creates the System
	SFMLSystem* sparkSystem = SFMLSystem::create();
	sparkSystem->addGroup(sparkGroup);

	// Defines which objects will be shared by all systems
	sparkModel->setShared(true);
	sparkRenderer->setShared(true);

	// returns the ID
	return sparkSystem->getID();
}
```

The model is created by enabling the 4 color components. The red is set to 1 while the green will take a random value between 0.2 and 1 and the blue between 0 and 0.2. Generated particles will therefore have a random color betweem red (1,0.2,0) and yellow (1,1,0.2). furthermore, the will progressively fade before dying as the alpha parameter is set to mutable with values 0.8 and 0. The life time of a spark is very short, its values will therefore be between 0.2s and 0.6s.

The renderer used is the line renderer (**SPK::SFML::SFMLLineRenderer**). Particles are renderered using non textured lines. The direction of the line follows the movement of the particle and its length is function of the speed of the particle. The speed is multiplied by a factor to give the length of the line. This factor is the first parameter of the constructor. We set it to 0.1. The second parameter is the line width. It is expressed in pixels and is common to all particles rendered with this renderer. We set it to 1 pixel.

The emitter used is the spheric emitter (**SPK::SphericEmitter**) which allows to define an axis and 2 angles between which particles will be emitted. Here, the axis is towards the top (0,0,1), the minimum angle is PI/4 and the maximum one is 3PI/4. The emitter zone is a sphere (**SPK::Sphere**) of radius 5 pixels and which is located at 10px above the ground. We use a sphere rather than a point in order not to get the feeling sparks are emitted from a single point.

The rest of the creation of the base system looks like the fireworks of the first tutorial except we use a **SPK::SFML::SFMLSystem** instead of a **SPK::System**.

Regarding creation, management and destruction, it is also the same as in the first tutorial :

We write a function to create a system and one to destroy a system :

```
// creates a particle system from the base system
SFMLSystem* createParticleSystem(const Vector2f& pos)
{
	SFMLSystem* sparkSystem = SPK_Copy(SFMLSystem,BaseSparkSystemID);
	sparkSystem->SetPosition(pos);

	return sparkSystem;
```

```
}


// destroy a particle system
void destroyParticleSystem(SFMLSystem*& system)
{
	SPK_Destroy(system);
	system = NULL;
}
```

The function to destroy a system is called in the update loop as soon as a system becomes idle (no more particles are emitted or alive) :

```
deque<SFMLSystem*>::iterator it = collisionParticleSystems.begin();
while(it != collisionParticleSystems.end())
{
	if (!(*it)->update(deltaTime * 0.001f))
	{
		destroyParticleSystem(*it);
		it = collisionParticleSystems.erase(it);
	}
	else
		++it;
}
```

Note that as in the first tutorial, the container used to store systems is a **std::deque** as the general functionning is FIFO (the older the system, the first to be dying).

A new instance will be created at each new collision whether it be car/car or car/border. During the cars update loop, their positions are updated. 2 methods of the class Car allows to verify the collisions and update the positions in case of collisions : one against the border and the other against another car. These 2 methods takes a parameter which is a reference on a vector. If a collision happens, the method will return true and update the vector values with the coordinates of the impact. We then only have to create a new system at this location :

```
for (size_t i = 0; i < NB_CARS; ++i)
{
	Vector2f collisionPos;
	car[i].update(deltaTime * 0.001f);
	if (car[i].checkCollisionWithBox(collisionPos))
		collisionParticleSystems.push_back(createParticleSystem(collisionPos));
}


for (size_t i = 0; i < NB_CARS; ++i)
	for (size_t j = i + 1; j < NB_CARS; ++j)
	{
		Vector2f collisionPos;
		if (car[i].checkCollisionWithCar(car[j],collisionPos))
			collisionParticleSystems.push_back(createParticleSystem(collisionPos));
	}
```

The rendering code is exactly the same as the one in the first tutorial.

V Second system : the sand cloud

The second effect we want to implement is the sand projections of the cars. As it is a tutorial about the creation of particle systems, this effect will be exaggerated. We an instance of system per car. The management of this system will therefore be encapsulated in the class Car.

Here is how the base system initialization looks :

```
void Car::loadBaseParticleSystem(Image& smoke)
{
	// Create the model
	Model* smokeModel = Model::create(
		FLAG_SIZE | FLAG_ALPHA | FLAG_TEXTURE_INDEX | FLAG_ANGLE,
		FLAG_SIZE | FLAG_ALPHA,
		FLAG_SIZE | FLAG_TEXTURE_INDEX | FLAG_ANGLE);
	smokeModel->setParam(PARAM_SIZE,5.0f,10.0f,100.0f,200.0f);
	smokeModel->setParam(PARAM_ALPHA,1.0f,0.0f);
	smokeModel->setParam(PARAM_TEXTURE_INDEX,0.0f,4.0f);
	smokeModel->setParam(PARAM_ANGLE,0.0f,PI * 2.0f);
	smokeModel->setLifeTime(2.0f,5.0f);

	// Creates the renderer
	SFMLQuadRenderer* smokeRenderer = SFMLQuadRenderer::create();
	smokeRenderer->setBlendMode(Blend::Alpha);
	smokeRenderer->setImage(smoke);
	smokeRenderer->setAtlasDimensions(2,2);
	smokeRenderer->setGroundCulling(true);

	// Creates the zone
	Point* leftTire = Point::create(Vector3D(8.0f,-28.0f));
	Point* rightTire = Point::create(Vector3D(-8.0f,-28.0f));

	// Creates the emitters
	SphericEmitter* leftSmokeEmitter =
SphericEmitter::create(Vector3D(0.0f,0.0f,1.0f),0.0f,1.1f * PI);
	leftSmokeEmitter->setZone(leftTire);

	SphericEmitter* rightSmokeEmitter =
SphericEmitter::create(Vector3D(0.0f,0.0f,1.0f),0.0f,1.1f * PI);
	rightSmokeEmitter->setZone(rightTire);

	// Creates the group
	Group* smokeGroup = Group::create(smokeModel,500);
	smokeGroup->setGravity(Vector3D(0.0f,0.0f,2.0f));
	smokeGroup->setRenderer(smokeRenderer);
	smokeGroup->addEmitter(leftSmokeEmitter);
	smokeGroup->addEmitter(rightSmokeEmitter);
	smokeGroup->enableSorting(true);

	// Creates the system
	SFMLSystem* system = SFMLSystem::create();
	system->addGroup(smokeGroup);
```

```
	// Defines which objects will be shared by all systems
	smokeModel->setShared(true);
	smokeRenderer->setShared(true);

	// Gets the ID
	baseParticleSystemID = system->getID();
}
```

The model is a bit different from the ones seen before. Here we do not activate the RGB components because we want to keep the default color (white (1,1,1)) and use the image unaltered. We gonna make the particles get bigger along their life; We realise a linear fade with the alpha; Finally, to prevent the apparation of patterns du to the repetition of the same image in the smoke, we will use 4 different patterns in a single image; by splitting it in 4. We therefore activate the parameter TEXTURE_INDEX which we set to random on [0.0,4.0[. Hence each particle will have an index between 0 and 4 (not included). We will see later how this index will define the part of the image to use. To improve the rendering again, we will given a random orientation to each particle. The parameter ANGLE is of that use. We set it to random between 0 and 2PI which makes an entire circle. Note that neither TEXTURE_INDEX not ANGLE are set to *mutable* in our case, this is to keep the sand cloud quite static. However it is possible to achieve texture animation and rotation by making those parameters mutable.

The renderer used is the quad renderer (**SPK::SFML::SFMLQuadRenderer**). This is the most versatile renderer. It will allow to reder particles as textured quads and, unlike the point renderer (**SPK::SFML::SFMLPointRenderer**) in point sprite mode, to render particles with sizes and orientations that vary. As seen before, we will use several patterns for the smoke. We therefore realize an image with 4 patterns and notify the renderer that our image is split in 4. This is performed with a call to **SPK::SFML::SFMLRenderer::setAtlasDimensions(unsigned int,unsigned int).** The values passed in our case are 2 and 2, which means our image has 2 columns and 2 rows. The indices for each patterns are defined by the occidental norm : from left to right and from top to bottom.

We then create 2 emitters, one per rear wheel and position them so that they are well located on the wheels when the car is not rotated. Then we will translate and rotate the entire system to match the car position and orientation at each frame. As we only want the emitters and zones to be transformed, the system must be in world coordinates (the default case), thus the generated particles are entirely independent from the system transform once spawned.

Finally, we will sort the particles to give a better look to our smoke. The sorting is not necessary in general with an additive blending but can improve the rendering with an alpha blending. To sort the particles of a group at each update, we only have to enable it with **SPK::Group::enableSorting(bool)**. The user must then tell the engine where the camera is located to allow the sorting. This is done with a call to **SPK::System::SetCameraPosition(const Vector3D&)**. A utility function is implemented is the SFML module to help position the camera : **SPK::SFML::SetCameraPosition(CameraAnchor,CameraAnchor,float,float)**. For a complete explanation about the parameters, take a look at the library documentation. The following code allows us to position the camera at the bottom

center of the screen with an altitude :

```
setCameraPosition(CAMERA_CENTER,CAMERA_BOTTOM,static_cast<float>(universeHeight),0.0f);
```

As our camera is fixed, we call this function only once at the initialization but for a moving camera, the function must be called when necessary. Note that the sorting is an expensive operation which scales rather badly as its compleity is o(n log n) and o(n²) in the worst case. The sorting must therefore be avoided when possible.

The update of the system is encapsulated in the class Car and is performed at the end of the car update :

```
void Car::updateParticleSystem(float deltaTime,float angle,float power)
{
	float forceMin = power * 0.04f;
	float forceMax = power * 0.08f;
	float flow = power * 0.20f;
	particleSystem->getGroup(0)->getEmitter(0)->setForce(forceMin,forceMax);
	particleSystem->getGroup(0)->getEmitter(1)->setForce(forceMin,forceMax);
	particleSystem->getGroup(0)->getEmitter(0)->setFlow(flow);
	particleSystem->getGroup(0)->getEmitter(1)->setFlow(flow);

	particleSystem->SetPosition(pos);
	particleSystem->SetRotation(angle * 180.0f / PI);
	particleSystem->update(deltaTime);
}
```

Before updating the system, the force and flow of the emitters is set in function of the speed of the vehicle (the variable named power). A car with no speed will therefore not generate any sand smoke (because the flow of the emitters will be 0). The system is then positionned and rotated by using inherited method of **sf::Drawable**.

The rendering is eventually performed in the main loop, after the cars renderering. It could also have been encapsulated within the class Car.

Note that the class Car is not really complete :

- its destructor should destroy the encapsulated particle system
- its copy constructor and the assigment operator should copy its encapsulated particle system.

However, in our case, as the cars will only be destroyed with the application and are never copied, I didnt do it.

VI Conclusion

In this tutorial we learnt to use SPARK with the SFML 2D engine by integrating 2 types of simple particle systems in an SFML context.

We could have go further by :

- optimizing the systems to reduce the number of batches (unify the systems of the same types). By unifying the systems, the sorting can be performed over the whole particles of smoke (If we take a look when the smoke systems of 2 cars crosses each other, one will be rendered above the other).
- adding new particle systems such as debris after a collision or tyre prints on the sand...
- having the cars interact with the particles (a car passing through a cloud of smoke should make the smoke react)

All those improvement will be implemented in a tutorial to come about advanced particle systems.