SPARK Tutoriel 2

Utilisation de SPARK
avec la SFML

Dans ce second tutoriel, nous allons apprendre à intégrer des systèmes de particules dans un univers réalisé à l'aide du moteur 2D de la SFML 1.5 (Simple and Fast Multimedia Library).

Pour ce faire, nous allons utiliser un contexte de pseudo-jeu codé pour l'occasion. Notre application mettra en scène 4 voitures vu en pseudo-perspective se déplaçant aléatoirement dans un environnement de sable et pouvant rentrer en collision les unes avec les autres et avec les bords.

Dans cette petite application, nous allons intégrer deux types d'effets à base de particules :

- la fumée de sable générée par le déplacement des voitures.
- les étincelles dues à une collision.

Notez que les points abordés dans le premier tutoriel ne seront pas revus ici.

I - Structure des systèmes

Comme nous allons intégrer 2 effets différents, nous allons construire deux modèles de systèmes de particules. Ces modèles, enregistrés dans la fabrique de SPARK, serviront ensuite à construire des instance de nos systèmes des 2 types selon le besoin.

Dans l'application, une classe Car a été implémenté pour gérer les instances de voitures. le système de particules représentant le sable brassé par le déplacement sera donc naturellement intégré à cette classe. Les étincelles lors des collisions seront quand à elles directement gérées dans le code principale de l'application.

Le système de fumée sera composé d'un unique groupe avec deux émetteurs (un par pneu arrière). Ces émetteurs généreront des particules de fumée (qui seront des quads texturés) de façon continue. Leur flow sera fonction de la vitesse de la voiture.

Le système d'étincelle, quand à lui, sera très semblables au système utilisé pour les feux d'artifices du premier tutoriel : Un groupe avec un émetteur qui générera un certains nombre de particules dès sa création et qui mourra lorsque toutes ses particules seront mortes. L e rendereur utilisé sera le rendereur SFML de lignes.

II - Initialisation

L'utilisation de SPARK avec la SFML est identique a son utilisation sans, mis à part le fait qu'il faille inclure le module SFML :

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

Concernant la liaison des bibliothèques, le module SFML doit être lié. Le module OpenGL doit également être lié car le module SFML fait appel à certaines de ses classes/méthodes. SPARK se lie soit dynamiquement, soit statiquement avec

la SFML (dynamiquement pour la version dynamique et statiquement pour la version statique).

L'espace de nommage relatif au module SFML est le suivant :

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

III – Spécificités du module SFML

Les Rendereurs SFML

Il existe actuellement 4 rendereurs pour la SFML :

- **SPK::SFML::SFMLQuadRenderer** qui permet de rendre des particules en utilisant des quads texturés (ou non).
- **SPK::SFML::SFMLPointRenderer** qui utilise la primitive OpenGL de point pour rendre les particules. Les points peuvent être soit carré, soit rond ou alors texturés (en utilisant les pointSprites).
- **SPK::SFML::SFMLLineRenderer** qui utilise la primitive openGL de ligne pour rendre les particules. Les lignes sont dans la direction du déplacement de la particule et leur longueur et fonction de la vitesse de la particule.
- **SPK::SFML::SFMLDrawableRenderer** qui permet de rendre les particules en utilisant les primitives de base de la SFML. Notez que ce rendereur est de loin le plus lent car il utilise directement le moteur de la SFML qui n'est à la base pas concu pour rendre de grandes quantités de particules.

Ces rendereurs ont la particularité d'être 2D. Cependant les calculs dans SPARK sont réalisés en 3D. L'utilisateur a alors la possibilité d'utiliser les coordonnées en Z des particules dans le rendu. Les Z sont alors multipliés par un facteur (définie par **SPK::SFML::SFMLRenderer::setZFactor(float)**)puis ajouté au Y, cela permet de modifier la position en Y des particules en fonction de leur altitude (Z). Malgré tout, le rendu en 3D des particules dans une scène 2D en fonction d'un angle d'orientation n'est pas encore possible. Un patch devrait venir permettre de le faire à terme. Dans notre application, nous allons utiliser un facteur de 1.0 pour donner un peu de relief à nos systèmes. Notez qu'il est possible d'utiliser SPARK en pure 2D en settant le facteur Z a 0 (c'est sa valeur par défaut) et en ne renseignant que les 2 premiers paramètres des vecteurs.

Les rendereurs ont également une fonction permettant de culler les particules avec le sol. Si elle est activée, toutes les particules ayant un z inférieur à 0 ne seront pas affichées. Le culling avec le sol est setté avec **SPK::SFML::SFMLRenderer::setGroundCulling(bool)**.

Système SFML

La classe **SPK::SFML::SFMLSystem** est la classe de base de tout système de particules en SFML. Elle hérite à la fois de **SPK::System** et de **sf::Drawable**. Un système de particule est alors considéré dans la SFML comme n'importe quel autre drawable et peut donc être positionné en utilisant les fonctions de la SFML. Le blending et la couleur sont, quand à eux, imposé par le rendereur. Les valeurs du système sont donc écrasée. Ceci est du au fait que plusieurs rendereurs au sein d'un même système peuvent avoir des modes de blending différents.

Dans SFML, tout Drawable a besoin d'un **sf::RenderTarget** pour pouvoir être rendu. Le rendu de systèmes SFML peut être réalisé de 2 façons :

- *Façon SFML* :

A la manière de n'importe quel drawable :

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

- *Façon SPARK* :

Un renderTarget doit avant tout être associé au système :

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

On peut ensuite rendre le système de façon classique :

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

Concernant le positionnement des particules dans l'univers, il existe 2 manières de le faire :

- En coordonnées locales : Les particules émises sont positionnées en fonction du système. Cela signifie que si le système bouge ou rotationne, les particules le suivront.
- En coordonnées monde : Les particules sont émises dans le monde. Si le système bouge ou rotationne, seuls les émetteurs et les zones bougeront et non les particules.

La plupart du temps, les positionnement des particules en coordonnées monde sera utilisé. Le choix du mode de transformation se fait lors de la construction du système et ne peut être changé par la suite. Par défaut les systèmes sont en coordonnées monde.

IV Premier système : les étincelles

Nous allons commencer par intégrer les particules d'étincelles car ce système est très proche de celui du premier tutoriel. Tout comme dans le premier tutoriel nous allons créer un système de base, l'enregistrer dans la fabrique puis le copier pour chaque collision. Nous créons donc une fonction permettant d'initialiser le système de base :

```
// 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();
}
```

Le modèle est créé en rendant les 4 composantes couleurs *enabled*. Le rouge est fixé a 1 alors que le vert prendra une valeur aléatoire entre 0.2 et 1 et le bleu entre 0 et 0.2. Les particules générées auront donc une couleur aléatoire entre le rouge (1,0.2,0) et le jaune (1,1,0.2). De plus elle disparaitront progressivement avant de mourir puisque le paramètre alpha est setté en *mutable* de 0.8 a 0. La durée de vie d'une étincelle se doit d'être courte, elle est donc fixée entre 0.2s et 0.6s.

Le rendereur utilisé est rendereur de ligne (**SPK::SFML::SFMLLineRenderer**), les particules sont alors rendues sous forme d'un trait non texturé. La direction de ce trait suit le déplacement de la particule. La longueur du trait dépend de la vitesse de la particule. La vitesse est multiplié par un facteur pour obtenir la longueur du trait. Le facteur *length* est notre premier paramètre du constructeur. Nous le settons à 0.1. Le deuxième paramètre est la largeur du trait. Elle est définie en pixels et est commune à toute les particules rendues avec le rendereur. Ici, nous mettons 1.

L'émetteur utilisé est un émetteur sphérique (**SPK::SphericEmitter**) qui permet de définir 1 axe et 2 angles entre lesquels seront émis les particules. Ici, l'axe est dirigé vers le haut (0,0,1), l'angle minimum est PI/4 et l'angle maximum 3PI/4. La zone de l'émetteur est une sphère (**SPK::Sphere**) qu'on positionne à 10 pixel d'altitude et qui a un rayon de 5 pixels. On utilise une sphère plutôt qu'un point pour éviter d'avoir l'impression que toutes les étincelles sont issue d'un point unique.

Le reste de la création du système de base est assez identique a celle des feux d'artifices du premier tutoriel. Si ce n'est qu'on utilise un **SPK::SFML::SFMLSystem** plutôt qu'un **SPK::System**.

Au niveau de la création, du management et de la destruction des instances du système, c'est également la même chose que pour le premier tutoriel :

On a une fonction pour créer un système et une autre pour détruire :

```
// 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;
}
```

La fonction pour détruire un système est appelé dans la boucle d'update dès qu'un système est inactif (il n'émet plus de particules et il n'a plus de particules vivantes) :

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

Notez que comme pour le premier tutoriel, un **std::deque** est utilisée comme container car le fonctionnement général est en FIFO (le système le plus vieux sera celui qui mourra en premier).

Au niveau de la création de nouvelles instances, celle ci doit se faire à chaque nouvelle collision quelle soit voiture/voiture ou voiture/bord. Lors de la boucle d'update des voitures, leur position sont mises à jour. 2 méthodes de la classe Car permette de mettre à jour les positions en vérifiant les collisions : Une avec les bords et l'autre avec une autre voiture. Ces 2 méthodes prennent en paramètre,une référence sur un vecteur. Si il y a collision, en plus de mettre à jour les positions, elles retournent true et remplisse le vecteur passé en paramètre avec les coordonnées du point d'impact. Il suffit alors de créer une nouvelle instance du système à la position indiquée :

```
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));
      }
```

Le rendu quand à lui s'effectue de la même manière que pour le premier tutoriel.

V Second système : la fumée de sable

Le deuxième effet que nous allons implémenté sera les projections de sables des voitures. Comme c'est un tutoriel sur la création de système de particules, cet effet sera volontairement exagéré. Nous avons besoin d'une instance de système par voiture. La gestion de ce système sera donc encapsulé dans la classe car.

Tout d'abord voilà a quoi ressemble l'initialisation du système de base :

```
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();
}
```

Le modèle est un peu différents des précédents systèmes. Ici on n'active pas les composante RGB car on veut conserver la couleur par défaut (le blanc (1,1,1)) et utiliser uniquement la couleur de l'image. On va faire grossir les particules de fumée au long de leur vie; On réalise un fade linéaire avec l'alpha; Enfin, pour éviter l'apparition de motifs du au fait que la fumée est rendu avec une seule et même image, nous allons utiliser une image séparée en 4 motifs. On active donc le paramètre TEXTURE_INDEX que l'on set random sur [0.0,4.0[. Ainsi chaque particule aura un index entre 0 et 4 non inclus. Nous verrons plus tard comment cet index définira la partie de l'image a utiliser. Pour améliorer encore le rendu, nous allons donner une orientation aléatoire a chaque particule. Le paramètre ANGLE sert à cela. Nous le settons en random entre 0 et 2PI donc un cercle complet. Notez que ni TEXTURE_INDEX, ni ANGLE ne sont *mutable* dans notre cas, ceci dans le but de garder la fumée de sable assez statique. Cependant il est tout à fait possible de réaliser des animation de texture et des rotations en rendant ces paramètres mutables.

Le rendereur utilisé est le rendereur de quad (**SPK::SFML::SFMLQuadRenderer**). C'est le rendereur le plus polyvalent. Il va nous permettre de rendre les particules sous forme de quads texturés et, contrairement au rendereur de point (**SPK::SFML::SFMLPointRenderer**) en mode point sprite, de rendre les particules avec des tailles et des orientations qui varient. Comme vu précédemment nous souhaitons utiliser plusieurs motifs pour la fumée. Nous réalisons donc une image avec 4 motifs et spécifions au rendereur la façon dont ils sont organisés dans l'image. Ceci est fait à l'aide de la méthode **SPK::SFML::SFMLRenderer::setAtlasDimension(unsigned int,unsigned int).** Nous passons les valeurs 2 et 2 ce qui signifie qu'il y a 2 colonnes et 2 lignes de motifs. Les index pour chaque motif sont définie par la norme occidental : de gauche à droite et de haut en bas.

Nous créons 2 émetteurs, un par roue arrière et nous les positionnons de façon a ce qu'ils soient bien sur les roues

arrières lorsque la voiture n'a pas de rotation. Ensuite nous translaterons et orienterons le système entier pour qu'il corresponde à la position/orientation de la voiture à chaque frame. Comme nous souhaitons seulement que les émetteurs et les zones soient transformés, le système doit être en coordonnées monde (c'est le cas pas défaut), ainsi les particules générées deviennent totalement indépendantes de la transformée de leur système.

Finalement, nous allons trier les particules pour donner un meilleur effet à notre fumée. Le tri n'est pas nécessaire en général avec un blending additif mais pour un blending alpha il peut améliorer le rendu. Pour effectuer un tri sur les particules du groupe a chaque update, il suffit d'activer le tri avec **SPK::Group::enableSorting(bool)**. L'utilisateur doit cependant préciser au moteur ou se situe la caméra pour permettre le tri. Cela se fait avec un appel a **SPK::System::SetCameraPosition(const Vector3D&)**. Une fonction utilitaire existe dans le module SFML pour aider à positionner la caméra : **SPK::SFML::SetCameraPosition(CameraAnchor,CameraAnchor,float,float)**. Pour une explication des différents paramètres, jetez un coup d'oeil à la documentation. Le code suivant nous permet de positionner la caméra sur le bas de l'écran, au centre et en altitude :

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

Comme notre caméra est fixe, nous appelons cette fonction uniquement à l'initialisation mais pour une caméra qui bouge, elle doit être appelé lorsque nécessaire. Notez que le tri est une opération qui est couteuse et qui se scale assez mal puisque sa complexité est o(n log n) et o(n²) dans le pire des cas. Le tri doit donc être éviter lorsque c'est possible.

L'update du système est encapsulé dans la classe Car et s'effectue à la fin de l'update de la voiture :

```
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);
}
```

Avant d'updater le système, le force et le flow des émetteurs est setté en fonction de la variable power qui correspond a la vitesse de la voiture. Ainsi, une voiture à l'arrêt ne générera pas de fumée de sable (car le flow des émetteurs sera alors 0). Le système est ensuite correctement positionné et orienté en utilisant les méthodes héritées de **sf::Drawable**.

Finalement le rendu s'effectue dans la boucle principale du programme, après le rendu des voitures. Il aurait pu également être encapsulé.

Notez que la classe Car n'est pas vraiment complète :

- Le destructeur de Car devrait détruire le système de particules
- Le constructeur par copie est l'opérateur d'affection devrait copier le système de particules

Ici comme les voitures sont détruites lorsque l'on quitte l'application et qu'elles ne sont jamais copiées, je n'ai pas pris la peine d'implémenter cela.

VI Conclusion

Dans ce tutoriel, nous avons appris a utiliser SPARK avec le moteur 2D de la SFML en intégrant 2 types de systèmes de particules simples dans un contexte SFML.

On pourrait par la suite aller plus loin :

- Optimiser les systèmes pour réduire le nombre de batch (réunir les systèmes de même types en un). En réunissant les systèmes, le tri pourra être effectuer sur l'ensemble de la fumée (si on regarde lorsque les systèmes de fumée de 2 voitures se croisent, l'un passe au dessus de l'autre)
- Ajouter de nouveaux systèmes de particules (projection de débris sur le sol lors d'une collision, trace de pneus sur le sable...)
- Faire interagir les voitures avec les particules (une voiture passant dans de la fumée va la chasser par exemple)

Toutes ces améliorations seront vu dans un prochain tutoriel pour réaliser des systèmes plus avancés.