SPARK Tutoriel 1

Fonctionnement général de SPARK et premier système de particules

Dans ce premier tutoriel, nous allons apprendre a mettre en place un système de particules à l'aide de SPARK et à paramétrer correctement les différents éléments le composant.

Le système de particules que nous allons construire est un feux d'artifices basique. A chaque fois que l'utilisateur appuie sur espace, une explosion se produira à une position déterminée aléatoirement. La couleur de chaque explosion sera également déterminée aléatoirement.

Pour se faire nous allons utiliser la SDL comme bibliothèque de fenêtrage qui nous fournira un contexte OpenGL sur lequel rendre les particules. Les rendereurs utilisés seront donc les rendereurs OpenGL de SPARK.

I - Structure du système

La façon la plus simple, la plus intuitive et la plus robuste de créer un type de système de particules dans SPARK (**SPK::System**) est de définir un système de base et de l'enregistrer dans la fabrique de SPARK (**SPK::SPKFactory**). A chaque fois que l'on désirera créer un système de ce type, il suffira de demander à la fabrique de créer une nouvelle instance de ce système.

Chaque objet enregistrable dans la fabrique détient une fonction statique **create,** prenant les mêmes paramètres que son constructeur, qui permet de créer une nouvelle instance de l'objet directement enregistrée dans la fabrique.

Les instances actives seront conservées dans un conteneur pour permettre de les mettre à jour, les dessiner a l'écran, en rajouter, en supprimer, y accéder...

Dans SPARK, un système contient des pointeurs sur des groupes (**SPK::Group**). Ce sont ces groupes qui contiennent les particules et définissent les règles selon lesquelles elles seront créées et évolueront. SPARK utilise principalement une sémantique d'entités pour lier les différentes classes entre elles. Ainsi les instance ne sont pas nécessairement liées à d'autres (composition) mais peuvent être partagées (agrégation).

Dans notre cas, notre système ne contiendra qu'un seul groupe de particules qui sera l'explosion. Mais on peu imaginer des système à plusieurs groupes (un feu avec ses flammes et sa fumée par exemple).

II - Initialisation

Pour utiliser SPARK dans notre programme, il faut avant tout inclure les bibliothèque nécessaires. Dans notre cas, il faut inclure le noyau de SPARK ainsi que les rendereurs OpenGL :

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

Il faut bien sûr avoir, au préalable, lié les bibliothèques nécessaires à l'application soit statiquement ou dynamiquement. Dans le cas de la liaison dynamique, **SPK_IMPORT** doit être définie comme macro pour le préprocesseur. Le méthode

pour lier SPARK à l'application n'est pas abordée ici car elle dépend de l'EDI/compilateur utilisé. Néanmoins, c'est une liaison tout à fait classique.

SPARK utilise des espace de nommage par bibliothèque. Nous allons les utiliser pour éviter d'avoir à préfixer tous nos appels :

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

De plus, SPARK utilise un générateur pseudo-aléatoire interne optimisé. Il faut donc initialiser la graine de génération en début de programme. Nous l'initialisons sur le temps processeur :

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

III – Modèle de particules

Dans SPARK, chaque groupe contient un pointeur vers un modèle de particules (**SPK::Model**). Un groupe est créé avec un pointeur de modèle et ce pointeur ne peut être modifié une fois le groupe créé. Un groupe garde donc le même modèle tout au long de sa vie. Ceci est nécessaire car le modèle définit la façon dont les données des particules sont organisées en interne dans le groupe. Un modèle peut cependant être modifié en temps réel et partagé entre plusieurs groupes.

Un modèle permet d'activer ou non les paramètres des particules. Les paramètres des particules sont par exemple la taille, la masse, la couleur, l'angle... et consiste en une valeur à virgule flottante. L'activation et la manière dont ces paramètres évoluent sont définit à l'aide de flags. Ces flags sont au nombre de 3 :

- *le flag enable* : Il définit quels paramètres seront stockés individuellement par particule. Si un paramètre n'est pas présent dans ce flag et que le moteur en a besoin, la valeur par défaut du paramètre sera utilisée.
- *Le flag mutable* : Il définit quels paramètres évolueront linéairement durant la vie de la particule. Les paramètres *mutable* comportent 2 valeurs : une de début et une de fin.
- *Le flag random* : Il définit quels paramètres verront leur(s) valeur(s) générées aléatoirement. Les paramètres *random* comportent 2 valeurs : une minimale et une maximale.

Un paramètre *mutable* ou *random* doit être *enable*. Seuls les paramètres *enable* peuvent être *mutable* ou *random*. Un paramètre peut également être *mutable* et *random*. Ce qui signifie que la valeur de début est determinée aléatoirement entre 2 valeurs ainsi que sa valeur de sa fin. Ce type de paramètre est donc définit par 4 valeurs.

Prenons notre exemple. Voilà comment nous allons initialiser le modèle de particules pour le système de base :

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

- Le premier paramètre du constructeur/methode create d'un modèle est le *flag enable.* Nous activons les paramètres des 4 composantes de couleur (rouge,vert,bleu et alpha). Notez que le rouge, le bleu et le vert sont des paramètres qui sont automatiquement activés pour des raisons d'optimisation dans la structure des données,

il peuvent donc être omis.

- Le deuxième paramètre est le *flag mutable*. Nous souhaitons que nos particules fade lorsqu'elle vieillissent. Nous rendons donc le paramètre alpha *mutable* pour qu'il puisse évoluer automatiquement en fonction de la vie de la particule.
- Enfin, le troisième et dernier paramètre du constructeur est le *flag random.* Ici nous allons activer la couleur car nous souhaitons que les particules aient de légères variations de couleurs dans chaque système pour donner un plus bel effet à nos explosions.

Une fois les flags définis, il faut spécifier les valeurs des paramètres du modèle :

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

La méthode **SPK::Model::setParam** existe en 3 versions : à 1,2 ou 4 arguments. La version à 1 argument est utilisé pour spécifier la valeur des paramètres *enable* seulement, celle à 2 des paramètres *mutable* **ou** *random* et celle à 4, des paramètres *mutable* **et** *random*.

Ici le paramètre alpha est *mutable*, la première valeur définit donc sa valeur à la naissance de la particule, la seconde sa valeur à la mort de la particule. Nous souhaitons obtenir un fade linéaire, nous settons donc la valeur de naissance à 1 (opaque) et la valeur de mort à 0 (transparent). A chaque update d'une particule, sa valeur alpha sera donc interpolé linéairement par le moteur en fonction de son âge.

Nous ne définissons pas les valeurs des autres paramètres *enable* (les composantes couleurs) car celles ci seront settées à la création de chaque système.

Le modèle de particules est également responsable de la durée de vie des particules. Celle ci est définie à la naissance de la particule en tirant une valeur aléatoire entre un minimum et un maximum définis par **SPK::Model::setLifeTime(float min,float max)**.

IV – Emetteur de particules

Un émetteur (**SPK::Emitter**) permet de générer des particules avec une fréquence, une position et une direction donnée dans l'univers. Il existe plusieurs types d'émetteurs dans SPARK et il est également possible d'en créer des personnalisés (comme pour de nombreuses classes de SPARK).

Dans SPARK, un émetteur contient un pointeur sur une zone (**SPK::Zone**) qui définit sa forme et sa position dans l'univers. Lors de la génération d'une particule, la zone de l'émetteur génère sa position alors que l'émetteur en lui même génère son vecteur vitesse initial.

Dans notre exemple, nous souhaitons obtenir une explosion a partir d'un point. Nous allons donc utiliser un émetteur aléatoire (**SPK::RandomEmitter**) qui est l'émetteur le plus simple : Les particules vont être émise dans une direction aléatoire. Pour la zone, nous allons logiquement utiliser un 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 est la zone de l'émetteur. Elle est liée à celui ci avec l'appel à **SPK::Emitter::setZone(Zone*)**. Nous ne définissons pas la position du point pour le système de base car elle sera définie lors de la création de chaque système.

Ici nous avons défini un réservoir (*tank*) de 500 particules pour l'émetteur. Ce qui signifie qu'il ne pourra générer que 500 particules (jusqu'à ce que le réservoir soit réapprovisionné). Le flux (*flow*) de l'émetteur définit la fréquence d'émission par unité de temps. Une valeur négative signifie que toute les particules du réservoir doivent être émise à la première update (cela correspond à un flux infini). Une valeur négative pour le réservoir signifie également que celui ci est infinie, c est à dire qu'il ne s'épuisera jamais. Notez qu'un émetteur avec à la fois une valeur de flux et de réservoir infini est invalide. Ici, nous avons setté le réservoir à 500 et le flux à -1. Ce qui signifie que l'émetteur va émettre 500 particules d'un coup à la première update puis être vide, ce qui correspond bien au comportement d'une explosion.

La force de l'émetteur définie la force avec laquelle les particules vont être émises. Cela permet d'agir sur la vitesse d'émission. La vitesse d'émission correspond a : *vitesse = force / masse.* Nous mettons ici un petit delta entre la force minimale et la force maximale pour obtenir un effet plus naturel.

V – Rendereur

Le rendereur (**SPK::Renderer**) est l'objet responsable du rendu des particules. En effet sans rendereur, un set de particules n'est qu'un gros tableau de valeurs évoluant en fonction du temps. Le rendereur interprète ces valeurs pour obtenir une représentation graphique d'un groupe de particules.

SPARK propose plusieurs rendereurs à utiliser en fonction du rendu souhaité. L'utilisateur à également la possibilité de créer ses propres rendereurs. Tous les rendereurs n'interprètent pas l'intégralité des paramètres des particules. Par exemple un rendereur de points ignorera le paramètre **SPK::PARAM_ANGLE** d'une particule.

Dans notre exemple, nous allons utiliser le rendereur de points OpenGL (**SPK::GL::GLPointRenderer**) car il permet de rendre une particule à l'aide d'un quad texturé faisant toujours face à la caméra de façon optimisée :

```
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);
```

Le type **SPK::GL::POINT_SPRITE** est le type de point OpenGL qui permet d'afficher une texture plutôt qu'une couleur unie. **SPK::GL::GLPointRenderer::enableWorldSize(bool)** permet de définir si les points son rendu en pixels (donc avec tous les même taille) ou en coordonnées univers. Ici nous choisissons en coordonnées univers pour que la taille du point soit fonction de la distance à la caméra.

Pour permettre la conversion de coordonnées écrans à coordonnées univers, il est nécessaire de setter certains paramètres statiques permettant de calculer cela. Nous passons donc l'angle du champs de vision en y (en radians) et la hauteur de l'écran en pixel à la méthode statique **SPK::GL::GLExtInterface::setPixelPerUnit(float,unsigned int)**.

La taille d'une particule dans l'univers est ensuite définie (En effet ce rendereur ignore le paramètre de modèle **SPK::PARAM_SIZE**), ainsi que la texture à utiliser. Le paramètre pour la texture correspond à l'index d'une texture

OpenGL.

Nous allons ensuite paramétrer le *blending* du renderer. Pour nos particules, nous souhaitons un blending additif (qui correspond bien à des particules de lumières pour notre explosion). Nous allons également régler le blending de la texture pour qu'elle se module avec la couleur de la particule :

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

La dernière ligne permet de ne pas utiliser le depth write, c'est a dire que les particules ne seront pas ecrite dans le z buffer mais le z test sera effectué neanmoins.

Notre rendereur est maintenant configuré. Le seul problème étant qu'il utilise des extensions OpenGL et que si ces extensions ne sont pas prise en charge sur une configuration, les résultats ne seront pas ceux escomptés (toutefois le programme ne plantera pas). Même si seulement des cartes graphiques bien anciennes ne vont pas gérer ces extensions, par soucis de compatibilité, nous allons proposer une alternative en utilisant un autre rendereur OpenGL qui, lui, garantira la compatibilité à partir d'OpenGL 1.1.

SPARK gère automatiquement les extensions OpenGL via une classe comportant des méthodes statiques : **SPK::GLExtHandler**. Notre rendereur ci dessus a besoin de 2 extensions OpenGL :

- L'extension *PointSprite* qui permet d'attacher une texture à un point OpenGL. Si elle n'est pas supportée, la texture sera ignoré.
- L'extension *PointParameter* qui permet de moduler la taille d'un point OpenGL en fonction de sa distance à la caméra. Si elle n'est pas supportée, toutes les particules feront la même taille à l'écran quelque soit leur distance de la caméra.

Si une des deux extensions n'est pas supportée, nous allons donc utiliser le rendereur de quad OpenGL (**SPK::GL::GLQuadRenderer**) qui est beaucoup plus modulaire et portable mais mois optimisé dans ce cas précis, sinon nous gardons le rendereur de point OpenGL.

Voilà donc le code final du rendereur que je ne détaillerai pas plus mais qui permet d'avoir un rendereur hautement compatible et optimisé :

```
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 pointe donc vers le bon rendereur et sera utilisé par le groupe de particules.

VI – Groupe de particules

Maintenant que nous avons créé et configurer un modèle, une émetteur et un rendereur, nous pouvons créer notre groupe de particules :

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

Le groupe est créé avec un modèle et une capacité. La capacité définit le nombre maximale de particules que peut contenir le groupe en temps réel. Comme l'émetteur va émettre 500 particules a la première update puis devenir inactive, le nombre maximale de particules sera donc de 500.

Il faut ensuite attacher notre émetteur et notre rendereur au groupe :

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

Dans SPARK, il existe les modifieurs (**SPK::Modifier**) qui permettent de modifier le comportement des particules d'un groupe en temps réel en fonction de paramètres internes et/ou externes (Les modifieurs ne sont pas abordés dans ce tutoriel). Pour faciliter l'implémentation d'un système de particules, les groupes contiennent 2 paramètres physiques souvent utilisés influant sur le comportement des particules :

- La gravité qui est une force poussant la particule dans une direction donnée et dont l'action est non dépendante de la masse de la particule.
- La friction qui est une force créé par la résistance d'un fluide sur un objet en déplacement dans celui ci. Cette force s'oppose donc au déplacement de la particule et est fonction de sa vitesse.

Dans notre exemple nous allons utilisé ces 2 paramètres pour donner un déplacement plus naturel aux particules de nos explosions de feux d'artifices :

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

Finalement, nous attachons notre groupe à notre système. Notez que dans notre cas, utiliser un système n'était pas forcément nécessaire car nous n'avons qu'un seul groupe et qu'un système existe avant tout pour regrouper plusieurs groupes de particules. En fait la plupart de l'interface de **SPK::System** existe dans **SPK::Group**. Nous utilisons tout de même un système car cette classe a été prévu pour être manier par l'utilisateur.

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

VII – Système de base

Notre système de base est maintenant correctement configuré. De plus comme les methodes statiques create ont été utilisé, le système est tout ses composants est enregistré dans la fabrique. La fabrique peut enregistrer tout type d'objet enregistrable (**SPK::Registerable**).

La fabrique va ainsi nous permettre de copier correctement des objets enregistrés. En effet sans utiliser la fabrique, si on copie par exemple le système, seule l'adresse du groupe sera copiée et non le groupe. Si l'on veut que le groupe soit copié, il faut coder soit même le comportement. Avec la fabrique, tout est automatique.

Pour aller plus loin, l'utilisateur a la possibilité de définir quels enfants seront copiés et quels enfants auront juste leur adresse de copiée lors d'une copie de leur parent. Dans le deuxième cas, on parle d'objets enregistrables partagés (*shared registerables*). Par défaut un objet enregistrable n'est pas partagé. Un objet partagé permet d'économiser de la mémoire et du temps de copie et permet également la modification d'un paramètre commun à plusieurs systèmes.

Dans notre exemple, nous allons partager le rendereur ainsi que le modèle. Il suffit pour cela de les configurer comme étant partagés :

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

A chaque fois que nous copierons notre système de base pour obtenir un nouveau système, le modèle et le rendereur ne seront pas copiés mais ceux du système de base seront utilisés. Tout objet enregistré dans la fabrique détient un ID qui l'identifie de façon unique notre système :

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

Notre système de base est donc enregistré dans la fabrique et nous avons un ID l'identifiant pour permettre sa copie.

Toute la création du système de base est réalisé dans une fonction **createParticleSystemBase**. Cette fonction nous retourne l'ID du système enregistrés dans la fabrique.

VIII – Création de systèmes

Comme vue précédemment, nous allons utiliser la fabrique pour créer une nouvelle explosion à partir du système de base. Pour ce faire, une fonction **createParticleSystem** est créée. Elle prend en paramètre une position et une couleur et retourne un pointeur sur le nouveau système :

```
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;
}
```

Les composantes de couleur du modèle ainsi que la position de la zone de l'émetteur sont fixés par les arguments de la fonction. Un random est ajouté à la couleur pour avoir une légère variation dans les particules d'une même explosion, tout en gardant une couleur unique par explosion.

Cette fonction est appelé lors de l appuie sur la touche espace. La position est généré aléatoirement dans une boite et la couleur est généré en mode HSV en tirant uniquement la teinte aléatoirement. La couleur est ensuite convertie en RGB avant d'être passée à la fonction.

Le retour de la fonction est stocké dans un conteneur de la STL de type *deque* car nous désirons un comportement FIFO (first in first out).

IX – Update et rendu de systèmes

Tous nos systèmes actifs étant stockés dans un conteneur, pour les updater, il suffit d'appeler la méthode **SPK::System::update(float)** pour chacun d'entre eux :

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

Le paramètre passé à la méthode étant le temps duquel il faut avancer le système.

Pour dessiner un système, il suffit d'appeler la méthode **SPK::System::render()**. De la même façon :

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

X – Destruction de systèmes

Un objet enregistré dans la fabrique doit être détruit avec un appel spécifique. Cet appel va détruire l'objet ainsi que tous ces enfants qui n'ont plus de références autre que le parent à détruire. Le fabrique conserve en effet un compteur de références sur ses objets enregistrés. Le nombre de références correspond au nombre de fois où l'adresse de l'objet existe dans l'ensemble des objets enregistrés dans la fabrique. Nous écrivons donc une fonction permettant de détruire un système :

```
void destroyParticleSystem(System*& system)
```

```
{
	SPK_Destroy(system);
	system = NULL;
}
```

Notez que les objets partagés (dans notre cas le modèle et le rendereur) ne seront pas détruit avec le système à détruire car il existe une référence à ceux ci dans le système de base (et éventuellement dans d'autres systèmes actifs).

Notez qu'un objet enregistré dans la fabrique ne doit pas être détruit avec un delete !

SPARK offre un moyen de savoir si un système ou un groupe de particules est inactif. Un groupe est considéré comme inactif si :

- Il n'a plus aucune particules active
- Il n'a plus aucun émetteurs actif. Un émetteur est actif s'il lui reste des particules dans son réservoir et si son flux est différent de 0.

Un système, quant à lui, est considéré comme inactif lorsque tous ces groupes sont inactifs.
La méthode **SPK::System::update(float)** retourne un booléen spécifiant si oui ou non le système est actif. Il suffit alors de détruire les systèmes dont la méthode update retourne *false*. La nouvelle boucle update nous donc donc :

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

Ainsi dès qu'un système sera deviendra inactif, il sera détruit.

Il reste ensuite à détruire le système de base, une fois que l'on en a plus besoin (dans notre cas ce sera à la fin du programme). Nous pouvons soit le détruire de la même façon que les autres systèmes, mais il existe également une méthode de la fabrique permettant de détruire tous les objets enregistrés. Nous appelons donc en fin de programme :

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

Notez que cette méthode n'a pas de macro associée permettant d'alléger la syntaxe de l'appel.

XI – Conclusion

Dans ce tutoriel, nous avons appris à utiliser SPARK pour mettre en place des systèmes de particules simples. Les possibilités de SPARK sont bien plus étendue que ce bref aperçu et je vous invite jeter un coup d'œil à la documentation de référence et aux applications de démos pour utiliser les fonctionnalités non abordées (optimisations de rendu, différents rendereurs, emetteurs, zones, les modifieurs, les fonctions callback, transformations, dérivations des classes de base...).