Shmupgine

Documentation

Cyprien Eymond Laritaz

Index

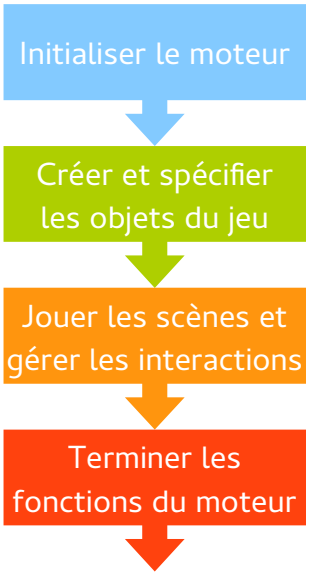
L'éditeur

Le moteur

Introduction

Shmupgine est un moteur de jeux conçu pour créer des jeux de type « Shoot'em up[1] ». Il s'agit d'un ensemble de bibliothèques basées sur SFML[2] Cependant, avec de l'imagination, il est tout à fait possible d'en faire quelque-chose d'autre.

Le coeur du système consiste en la mise en relation de trois classes : la classe `entity`, `attribute` et `scene`. Une scène contient un ensemble d'entités qui elles-même contiennent un ensemble d'attributs. Pour fonctionner convenablement, ces classes-là empruntent des variables et fonctions de l'espace de nom `shmupgine` qui contient par exemple le chronomètre, la fenêtre du jeu, ainsi que des fonctions d'initialisation.



1 Le Shoot'em up est un type de jeu dans lequel le véhicule que l'on contrôle doit évoluer et détruire des opposants devant lui en evitant les projectiles.

2 Simple Fast Multimedia Layer, site officiel.

Constantes

`DEF_BULLET_SIZE 16` : Taille par défaut des entités de type bullet.

Types

`attr_table` : renommage du type `std::list<attributes*>`

Classes

Entity

Les entités sont le cœur du moteur, ce sont des classes très pauvres que l'on enrichit avec des attributs.

La classe a trois constructeurs :

```
entity();
entity(scene* parent);
entity(scene* parent, sf::Vector2f initial_position);
```

Le premier est trivial, le second prend en paramètre un pointeur sur la scène qui contiendra l'entité en question. Le dernier est utilisé pour donner une position initiale à l'entité.

Attributs

- `attributes (attr_table)` : liste des attributs de l'entité.
- `position (sf::Vector2f)` : coordonnées de l'entité sur la scène.
- `parent (scene*)` : pointeur vers la scène qui contient l'entité.

Méthodes

- `allocateAttribute<T> (T* → void)` : attribue à l'entité l'attribut pointé en paramètre.
- `allocateAttribute<T> (void → void)` : alloue un nouvel attribut de type T pour l'entité.
- `run_attributes (void → void)` : procède à l'exécution des fonctions de chaque attribut activé de l'entité.
- `move (sf::Vector2f → void)` : déplace l'entité d'un offset donné en paramètre.
- `move (float, float → float)` : déplace l'entité d'un offset donné en paramètre.
- `setPosition (sf::Vector2f → void)` : place l'entité à la position posée en paramètre.
- `setPosition (float, float → void)` : place l'entité à la position posée en paramètre.
- `setParent (scene* → void)` : attribue à l'entité le parent passé en paramètre. L'entité sera donc actualisé par cette scène.

- `getParent (void → scene*)` : renvoie la scène parente de l'entité.
- `getPosition (void → sf::Vector2f)` : renvoie les coordonnées de la position de l'entité dans la scène.

Créer une entité personnalisée

Pour créer une entité personnalisée, il faut créer une classe héritée de la classe `entity`. Pour ce faire, il faut inclure la bibliothèque `entity.h` ainsi que de faire hériter votre classe de la classe entity. Veillez à programmer vos constructeurs en appelant les constructeurs de la classe `entity` pour la gestion des pointeurs sur la scène parent de l'entité. Ci-dessous l'implémentation de l'entité personnalisée `bullet` :

```
class bullet : public entity {
public:
      bullet();
      bullet(scene* parent);
      bullet(scene* parent, sf::Vector2f initial_position);
      bullet(sf::Vector2f initial_position);
      virtual ~bullet();
};
```

Dans le corps de la classe, il faut ensuite implémenter les différentes caractéristiques de notre nouvelle entité ; ça passe uniquement par les constructeurs.

Dans le corps des constructeurs, il faut allouer les attributs de votre entité et les « régler » à comprendre fixer les attributs de vos attributs. Ci-dessous l'implémentation du corps de l'entité personnalisée `bullet` :

```
#include "bullet.h"
#include "graphicrenderer.h"
#include "physics.h"

bullet::bullet() : entity() {
      allocateAttribute<graphicrenderer>();
      allocateAttribute<physics>();
      getAttribute<graphicrenderer>()->sprite.setTexture(
                              *tilesheet::tex_bullet);
      getAttribute<graphicrenderer>()->sprite.setOrigin(
                              sf::Vector2f(DEF_BULLET_SIZE,
DEF_BULLET_SIZE));
}

bullet::bullet(scene* parent) : entity(parent) {
      allocateAttribute<graphicrenderer>();
      allocateAttribute<physics>();
      getAttribute<graphicrenderer>()->sprite.setTexture(
                              *tilesheet::tex_bullet);
```

```
      getAttribute<graphicrenderer>()->sprite.setOrigin(
                                sf::Vector2f(DEF_BULLET_SIZE,
DEF_BULLET_SIZE));
}

bullet::bullet(scene* parent, sf::Vector2f initial_position) :
                                entity(parent, initial_position) {
      allocateAttribute<graphicrenderer>();
      allocateAttribute<physics>();
      getAttribute<graphicrenderer>()->sprite.setTexture(
                                *tilesheet::tex_bullet);
      getAttribute<graphicrenderer>()->sprite.setPosition(initial_position);
      getAttribute<graphicrenderer>()->sprite.setOrigin(
                                sf::Vector2f(DEF_BULLET_SIZE,
DEF_BULLET_SIZE));
}

bullet::bullet(sf::Vector2f initial_position) : entity(parent,
                                initial_position) {
      allocateAttribute<graphicrenderer>();
      allocateAttribute<physics>();
      getAttribute<graphicrenderer>()->sprite.setTexture(
                                *tilesheet::tex_bullet);
      getAttribute<graphicrenderer>()->sprite.setPosition(initial_position);
      getAttribute<graphicrenderer>()->sprite.setOrigin(
                                sf::Vector2f(DEF_BULLET_SIZE,
DEF_BULLET_SIZE));
}

bullet::~bullet() {
}
```

On voit donc dans les constructeur que l'on alloue l'attribut `graphicrenderer` ainsi que l'attribut `physics` grâce à la méthode `allocateAttribute<T>()`, et que nous les règlons grâce à la méthode `getAttribute<T>()`.

Une fois votre classe spécifiée de cette façon, vous pourrez l'instancier et l'utiliser dans vos projets, et la faire interagir avec les autres objets de la scène (et aussi éventuellement avec des objets appartenant à une autre scène, c'est parfaitement possible).

tl;dr

- Une entité est très pauvre ; sans attribut elle est inutile.
- Une entité a un pointeur vers la scène qui la contient
- On peut ajouter des attributs à volonté pour cette entité pour lui donner un comportement.
- Sans paramètre, une entité a pour position `(0;0)` et pour parent un pointeur sur `NULL`.

Attribute

Un attribut est un « morceau » de comportement d'une entité. Les attributs sont affiliés à une entité, et peuvent être alloués directement au moyen d'une entité. La classe `attribute` est assez pauvre, comme la classe `entity` ; elle contient seulement les bases pour construire des attributs plus spécialisés qui héritent de ce modèle. La classe a deux constructeurs :

```
attribute();
attribute(entity* parent);
```

Il n'est pas besoin de détailler le premier ; le second construit l'attribut affilé à l'entité `parent`.

Attributs

- `enabled (bool)` : interrupteur qui permet de savoir si l'attribut est actif. S'il ne l'est pas, il ne se jouera pas lorsque son entité parent va exécuter les fonctions de tous ses attributs (`run_attributes`).
- `parent (entity*)` : pointeur vers l'entité parent de l'attribut.

Méthodes

- `run (void → void)` : produit le comportement de l'attribut. C'est la méthode qui sera prise en compte par la scène pour effectuer les commandes de l'attribut.

Créer un attribut personnalisé

Pour créer un attribut personnalisé, il faut commencer par inclure la bibliothèque `attribute.h`. Une fois cela fait, il faut créer une classe qui va hériter de la classe `attribute`. Prenons par exemple l'implémentation de l'attribut `physics` :

```
#include <SFML/Graphics.hpp>
#include "attribute.h"

class physics : public attribute {
public:
	physics(entity* parent);
	physics(entity* parent, sf::Vector2f force, float velocity);
	virtual ~physics();
	virtual void	run();
	void		setForce(sf::Vector2f force);
	void		setVelocity(float velocity);

protected:
	sf::Vector2f		force;
	float		velocity;
};
```

On peut voir que les constructeurs sont personnalisés avec les différents attributs propres à la classe (`force` et `velocity`). En réalité, là où toute la particularité d'un attribut personnalisé reposait sur la programmation des constructeurs, ici ce sera évidemment quelque peu le cas, mais l'intérêt de l'attribut va résider dans la programmation de la méthode `run()`, c'est là où tout se joue. Voyons comment l'attribut personnalisé physics est programmé :

```
#include "physics.h"
#include "shmupgine.h"
#include "entity.h"

physics::physics(entity* parent) : attribute(parent) {
	this->force = sf::Vector2f(0, 0);
	this->velocity = 0;
}

physics::physics(entity* parent, sf::Vector2f force, float velocity) :
                                attribute(parent) {
	this->force = force;
	this->velocity = velocity;
}

physics::~physics() {

}

void physics::run() {
	parent->move(0.001f * force * velocity * (float)shmupgine::
                                clock.getElapsedTime().asMilliseconds());
	shmupgine::clock.restart();
}
void physics::setForce(sf::Vector2f force) {
	this->force = force;
}

void physics::setVelocity(float velocity) {
	this->velocity = velocity;
}
```

On voit donc le corps de méthodes triviales telles que setForce et consorts, mais la méthode intéressante, la méthode run, elle effectue ce pour quoi l'attribut a été créé, à savoir donner un comportement physique à l'objet. Ici c'est très sommaire bien sûr, mais voilà comment concevoir un attribut.

tl;dr

- Un attribut est très pauvre, c'est ses classes dérivées qui seront complexes.
- Une méthode est à considérer avec beaucoup d'attention, la méthode `run`.
- Un attribut est affilié à une entité parent (elle-même affiliée à une scène).

Scene

Une scène est une classe qui donne un contexte pour des entités. Des entités s'inscrivent dans une scène et c'est la scène qui va gérer les appels des fonctions des entités. Une scène n'est pas une classe qu'il est important de dériver, elle fournit un ensemble d'entités et de règles les concernant (comme l'ordre d'exécution des entités) dans l'unique but de pouvoir changer à la volée de contexte (phases dans un boss de jeu, systême de niveaux) sans avoir à créer de fonctions pour vider la liste d'entités et de tout recréer.

Attributs

- `entities (std::list<entity*>)` : liste des entités de la scène, c'est là qu'une entité est stockée lorsque en alloue une nouvelle.
- `to_be_removed (std::vector<entity*> to_be_removed)` : liste des entités à être supprimées.

Méthodes

- `run (void → void)` : démarre la scène.
- `update (void → void)` : boucle principale de la scène ; efface la fenêtre, exécute les entités, affiche la fenêtre et supprime les entités à supprimer.
- `run_attributes (void → void)` : parcoure la liste d'entités et exécute leurs attributs activés.
- `add_entity (entity* → void)` : ajoute à la scène l'entité passée en paramètre.
- `remove_entity (entity* → void)` : ajoute l'entité passée en paramètre dans la liste des entités à supprimer.
- `remove_entities (void → void)` : vide de la liste des entités toutes les entités qui sont aussi dans la lsite des entités à supprimer.

Utiliser une scène

tl;dr