UE: Programmation Orientée Objet (I4AEIL11)
Sujet de TP

Yannick Pencolé
yannick.pencole@laas.fr

2023-2024

1 Description générale de l'application

Au cours de ce TP, vous allez mettre en œuvre un embryon de jeu en deux dimensions (2D). Pour comprendre rapidement quel est le résultat final du logiciel que vous allez développer, vous pouvez consulter la vidéo qui présente un résultat possible de votre mise en œuvre. Cette vidéo se trouve sur moodle à l'endroit où vous avez téléchargé le sujet de TP.

Pour cette application de jeu, vous allez exploiter des classes de la bibliothèque standard C++ qui ont été vues en cours (`std::string`, `std::vector`,...) mais également la bibliothèque de classes C++ qui se nomme `SFML` [1]. Alors que l'espace de nommage de la bibliothèque standard est `std`, celui de la bibliothèque `SFML` est `sf`.

L'application ouvre une fenêtre graphique. Dans cette fenêtre, votre objectif final sera de placer des éléments de décors (des arbres, des tas de bois) et d'instancier un personnage (le héros) et des monstres. Le héros devra circuler dans cet environnement graphique en évitant de foncer dans les décors et dans les monstres. Le héros sera contrôlé avec les touches flèchées (déplacement) et la touche `s` (stop). Au contact des décors et des monstres, le héros perd des points de vie et le jeu se termine quand le héros trépasse...

La suite de ce sujet présente les différentes étapes pour la mise en œuvre de cette application.

2 Installation du projet sur votre compte

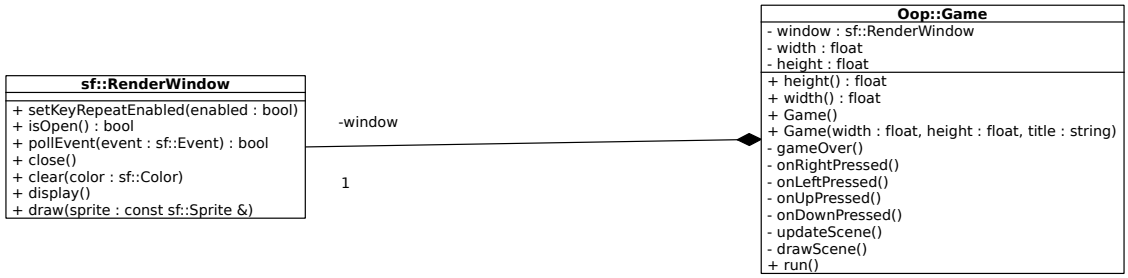
Pour cette mise en œuvre, nous allons utiliser Netbeans comme en TD. N'oubliez pas d'activer C++ dans Netbeans si ce n'est pas déjà fait (voir TD1). Le projet Netbeans pour votre mise en œuvre a déjà été créé et dispose de tous les éléments nécessaires. Téléchargez l'archive contenant le projet `tp_poo.tgz`, décompressez l'archive dans un répertoire de votre compte informatique. Ouvrez Netbeans, et cliquez sur *File* → *Open Project...*, allez

[1] `SFML`: *Simple and Fast Multimedia Library* (`https://www.sfml-dev.org/`)

dans le répertoire où vous avez décompressé l'archive et sélectionnez le répertoire `tp_poo`. Le projet devrait se charger. Compilez-le et exécutez-le, une fenêtre noire apparaît. Si vous appuyez sur les flèches ou sur `s` des messages devraient apparaître dans la console de Netbeans. Si vous appuyez sur ESC, la fenêtre disparaît, le programme se termine. Votre projet est installé, son développement peut démarrer. Un ensemble de fichiers est déjà disponible `Game.h, Game.cpp, SettingData.cpp, SettingData.h, main.cpp`. Les fichiers `SettingData.cpp, SettingData.h` sont complets et n'auront pas à être modifié. Initialement, le programme fonctionne de la façon suivante: il procède à des initialisations (générateur aléatoire et création d'un `SettingData` que vous utiliserez plus tard) puis il instancie un objet de type `Game` à partir duquel on appelle la méthode `run`. Cette méthode exécute la boucle d'exécution du jeu, c'est en particulier cette méthode qui affiche la fenêtre du jeu que vous allez mettre en œuvre. La méthode `run` ne se termine qu'à la fin du jeu. Cette méthode est déjà complète, vous n'avez pas à la modifier. [2]

3 La classe `Game`

La classe `Game` est la classe centrale de votre jeu, c'est elle qui va gérer l'affichage des différents éléments, la gestion des touches claviers. Pour le moment, elle est partiellement définie, elle dispose d'un constructeur par défaut et elle contient un attribut `_window` de type `sf::RenderWindow` qui met en œuvre une fenêtre d'affichage `SFML` (fourni par cette bibliothèque). Remarquez que la classe `Game` est mise en œuvre dans un espace de nommage qui lui est propre, c'est l'espace `Oop`. [3]



Le diagramme UML ci-dessus présente la relation entre la classe `Oop::Game` et la classe `sf::RenderWindow`, de quel type de relation s'agit-il ? Qu'est-ce que cela implique ?

Q1 Le constructeur par défaut crée une fenêtre de taille fixe 800 par 300 et lui attribue un titre par défaut, on veut pouvoir paramétrer tout cela. Mettre en œuvre le constructeur paramétré décrit sur le diagramme UML. Le tester en remplaçant dans le `main` le jeu par défaut avec celui contenant vos propres paramètres.

Q2 Pour le moment le destructeur de `Game` est vide. Est-ce normal sachant la relation

[2] ou alors vous savez ce que vous faites...

[3] Oop: *Object Oriented Programming.*

entre `Oop::Game` et `sf::RenderWindow` ? Le compléter.

Q3 Mettre en oeuvre les attributs et accesseurs `width()` et `height()` qui devront retourner la largeur et la hauteur de la fenêtre créée par l'un des deux constructeurs.

Q4 Dans la classe `Oop::Game`, identifier les méthodes publiques et privées et analyser le fonctionnement de `run`. Dans la suite, la méthode `run` n'aura pas à être modifiée, seules les méthodes privées exploitées par `run` le seront au moment voulu.

4 Préparatif à l'affichage: la classe `Element`

Cette section est dédiée à la création d'une nouvelle classe: la classe `Element`. Cette classe sera dans l'espace de nommage `Oop`. Cette classe va servir de classe de base à tous les objets que le jeu va afficher. Elle nous servira de base à la création des `sprites` dans notre jeu.

4.1 À lire en préambule: à propos des sprites

Un *sprite* (lutin en français) est dans le jeu vidéo un élément graphique qui peut se déplacer sur l'écran (ou pas si c'est un élément de décor). Dans `SFML`, un objet de type `sf::Sprite` existe déjà et il correspond à un élément graphique qui peut se déplacer dans une fenêtre graphique de type `sf::RenderWindow`. Un sprite va représenter une petite surface rectangulaire fixe d'une certaine largeur et d'une certaine hauteur dans la fenêtre dans laquelle va s'afficher une image (cette image est extraite d'une texture).

4.1.1 Placement d'un sprite dans la fenêtre du jeu

Le placement d'un sprite dans la fenêtre d'affichage s'appuie sur un système de coordonnées associé à la fenêtre: le point $(0, 0)$ est le point le plus en haut et le plus à gauche de la fenêtre, l'axe des x va de gauche à droite, celui des y de haut en bas. Le placement du sprite est défini par un couple de coordonnées (x, y) dans la fenêtre d'affichage qui fixe le point supérieur gauche de l'image rectangulaire associée au sprite (voir la figure 1).

La position courante d'un objet `sf::Sprite` peut être modifiée à l'aide du mutateur:

```
void setPosition(float x, float y);
```

De même, on peut récupérer la position courante du sprite à l'aide de l'accesseur:

```
sf::Vector2f getPosition();
```

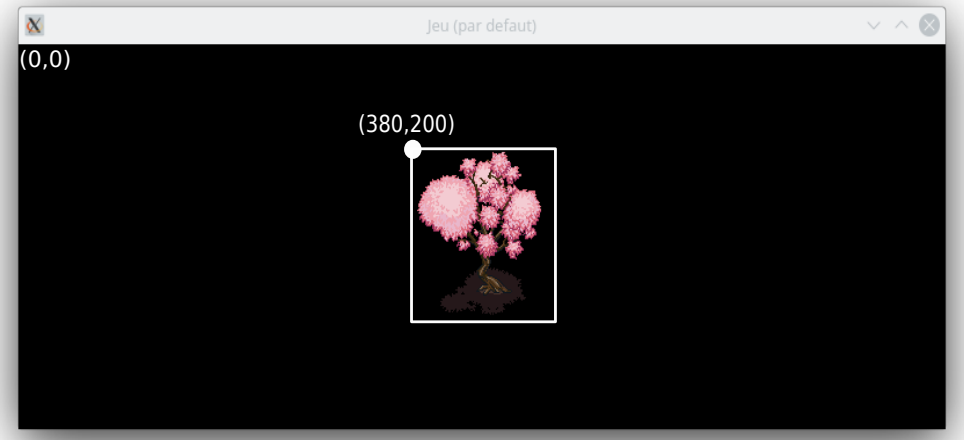


Figure 1: Affichage du sprite (décor du type `SettingType::TREE1`) à la position `(380,200)`.

La classe `sf::Vector2f` a deux attributs *publiques*,[4] les attributs `x` et `y` de type `float`. Ainsi, par exemple, pour récupérer la coordonnée x d'un sprite `s` à l'écran, il suffit d'écrire:

```
s.getPosition().x;
```

On peut également modifier la position courante en y ajoutant un déplacement relatif avec

```
void move(float offsetX, float offsetY);
```

4.1.2 Construction du sprite par extraction de texture

Une texture est en générale une image de grande taille qui est constituée d'un ensemble de petites images (voir les textures `settings.png` et `hero.png` par exemple qui sont incluses dans ce projet). Et l'image que le sprite doit afficher dans le jeu est définie par une zone rectangulaire de même taille qui est dans la texture (on dit que la texture est plaquée dans le sprite), voir la figure 2.

Il faut donc définir la position et la taille de ce rectangle dans la texture. Dans `SFML`, les zones rectangulaires sont représentées par des objets `sf::IntRect`. Par exemple, on peut construire le `rectangle` de largeur 100, hauteur 50 dont le point supérieur gauche est $(200, 250)$ comme suit:

```
sf::IntRect rectangle(200,250,100,50);
```

[4] C'est le choix de `SFML` en contradiction avec le paradigme objet qui aurait obligé à protéger ces attributs, mais la vie n'est-elle pas faite de contradictions ?

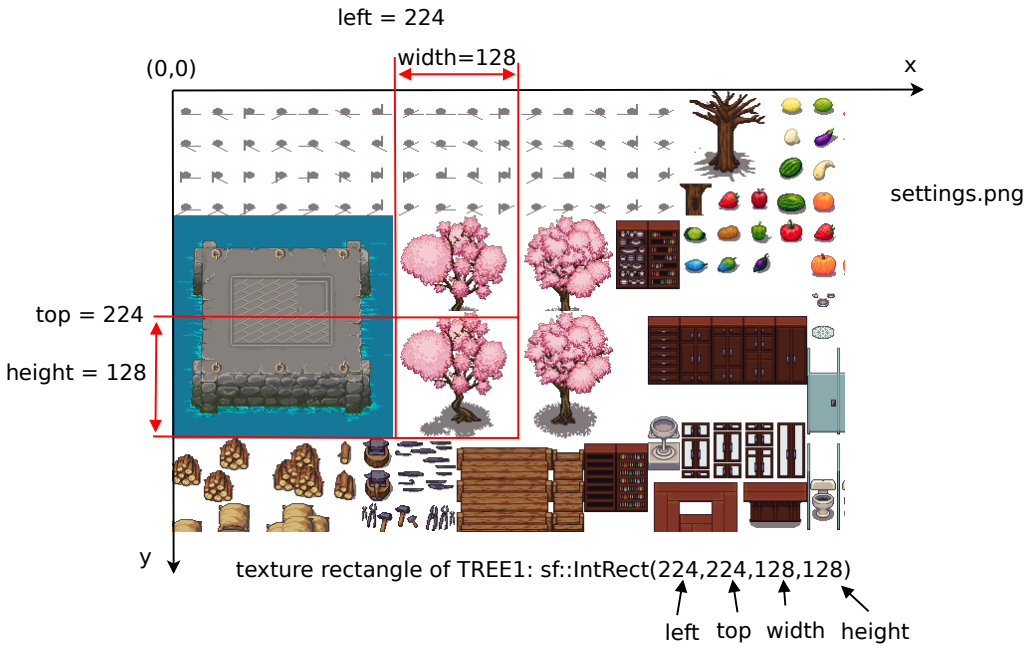


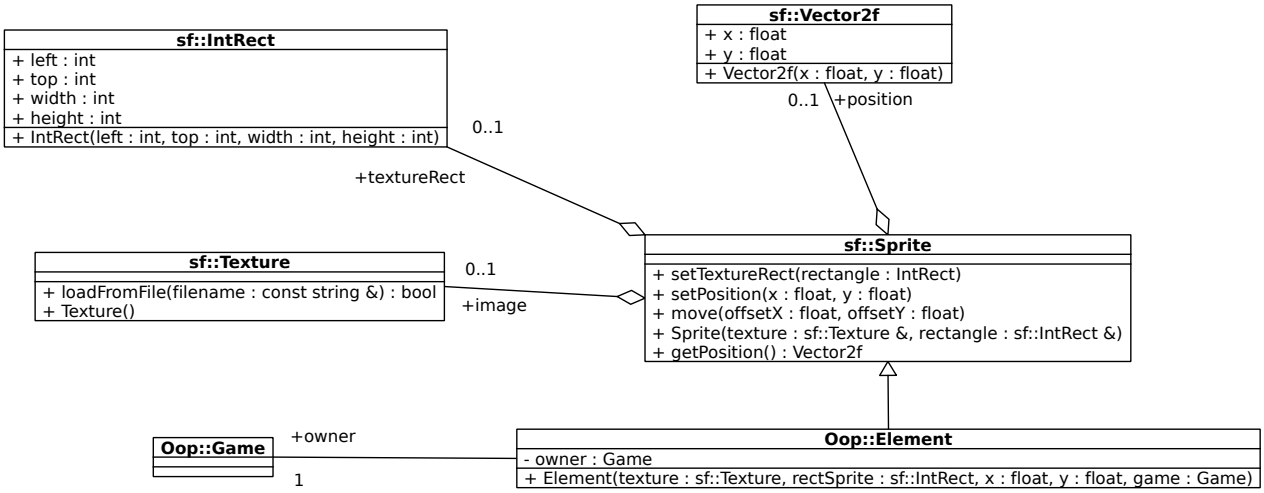
Figure 2: Rectangle de texture du sprite de type `SettingType::TREE1` dans la texture `settings.png`.

Cette classe `sf::IntRect` a également 4 attributs publiques: `left, top, width, height`. Ainsi, pour construire directement un sprite en `SFML`, on charge d'abord une texture, on détermine dans cette texture la zone rectangulaire que l'on veut extraire et on crée le sprite. Par exemple,

```
Texture texture;

texture.loadFromFile("image.png"); // on charge la texture en memoire

IntRect zone(240,300,64,32); // zone commencant en (240,300)

                             // de largeur 64 hauteur 32

sf::Sprite sprite(texture,zone); // creation d'un sprite associé

                                 // à la zone repérée dans la texture
```

4.2 À faire: La classe `Element`

Dans votre jeu, vous ne manipulerez pas directement des objets de classe `sf::Sprite`, vous allez procéder par dérivation en créant la classe `Element` qui hérite de la classe `sf::Sprite`. Le diagramme de classe suivant décrit les relations de la nouvelle classe `Element` avec les classes de `SFML` présentées ci-dessus et la classe `Oop::Game`.



Q5 Créer les fichiers `Element.cpp/.h` de la classe `Element`. Si Netbeans a créé automatiquement un constructeur par défaut et un constructeur par copie lors de la création de ces fichiers, supprimez-les. Mettre en œuvre le constructeur paramétré décrit sur le diagramme UML. Rappel: sur un diagramme UML, on décrit des références à des objets, en C++ une telle référence est mise en œuvre par un pointeur, ainsi la signature du constructeur paramétré de `Element` en C++ est:

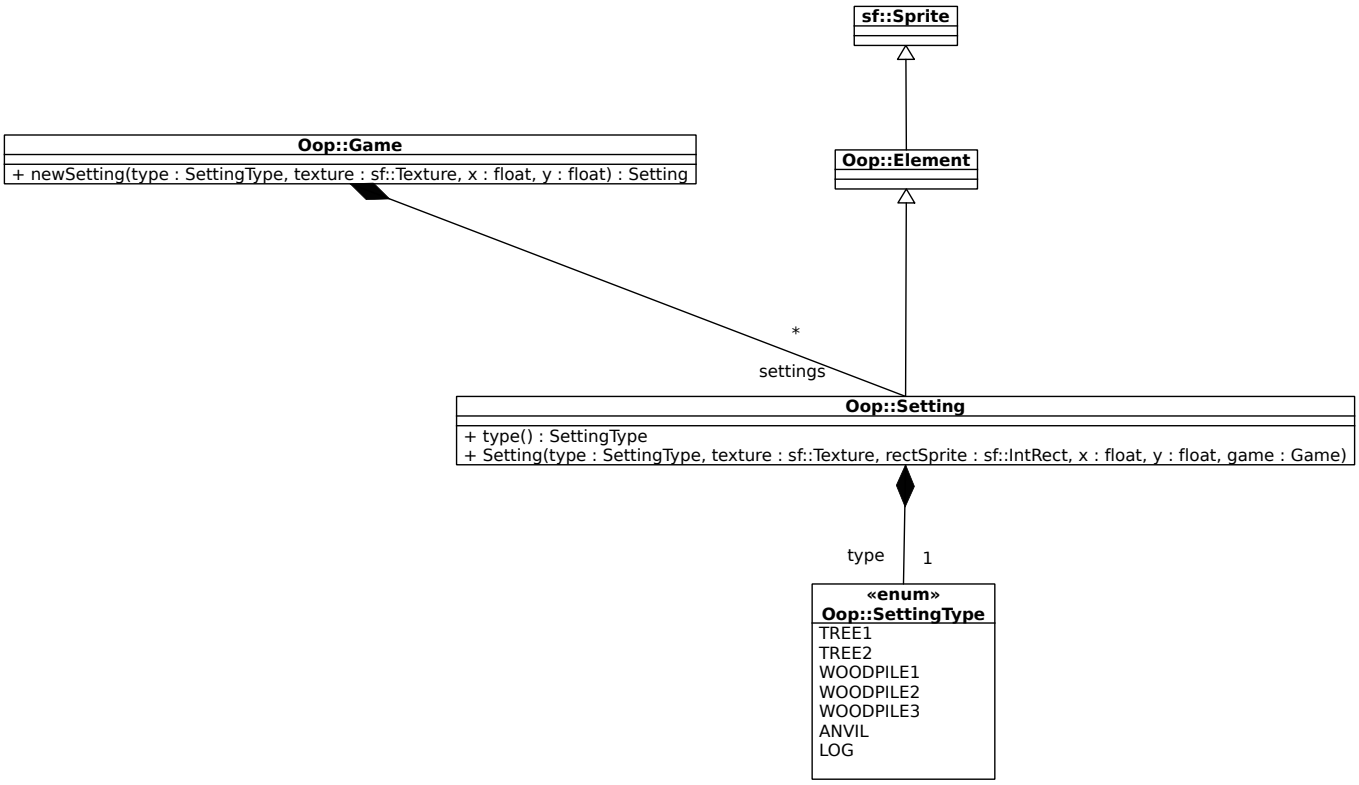
```
Element(sf::Texture * texture, sf::IntRect * rectSprite, float x, float y,
                                                    Game * game)
```

Ce constructeur devra appeler un constructeur de `sf::Sprite` et initialiser l'attribut `_owner` de l'objet `Element`.

Q6 Mettre en œuvre le destructeur de la classe `Element`. Selon le diagramme UML, ce constructeur est-il vide ou non ?

5 Création du décor

L'objectif de cette section est d'afficher dans le jeu des éléments de décor. Ces éléments de décor seront des objets de type `Oop::Setting`, nouvelle classe qui hérite de la classe `Oop::Element`.



La figure ci-dessus décrit le diagramme UML de la classe `Oop::Setting` et de ses relations. La classe `Setting` stocke un attribut de type `Oop::SettingType` qui est une énumération (déjà fournie dans le fichier `SettingData.h`). Noter que `Oop::SettingType` étant une simple énumération, cet objet est petit en mémoire (comme un entier), il est donc passé par copie et non par référence dans les paramètres des méthodes.

Q7 En vous appuyant sur la classe `Oop::Element`, mettre en œuvre la classe `Oop::Setting` telle que spécifiée sur le diagramme UML.

Le diagramme UML spécifie qu'un jeu est *composé* d'objets de type `Oop::Setting`.

Q8 Mettre en œuvre la relation de composition entre `Oop::Game` et `Oop::Setting` en ajoutant à `Oop::Game` les attributs nécessaires et en mettant en œuvre la méthode:

```
Setting * newSetting(SettingType type, sf::Texture * texture,  float x,
                     float y)
```

Cette méthode *instancie* des objets `Setting` et les stockent dans `Oop::Game`. Le pointeur retourné est un pointeur sur l'objet créé. Faut-il modifier le destructeur de `Oop::Game` ?

Maintenant, nous sommes prêts pour afficher notre premier élément de décor ! Dans le fichier `main.cpp`, la texture `settingTexture` a déjà été chargée, (c'est l'image contenue dans

`settings.png`). Un objet `data` de type `SettingData` a également été créé avec la fonction `Oop::createSettingData()`. Cet objet contient les zones rectangulaires `textureRect` à utiliser pour définir les sprites dont le type est prédéfini dans l'énumération `SettingType`.

Q9 Dans `main()`, après avoir initialisé le jeu, ajoutez-lui un élément de décor (par exemple de type `Oop::SettingType::TREE1`) en s'appuyant sur la texture `settingTexture` et sur les zones rectangulaires disponibles dans `data`.

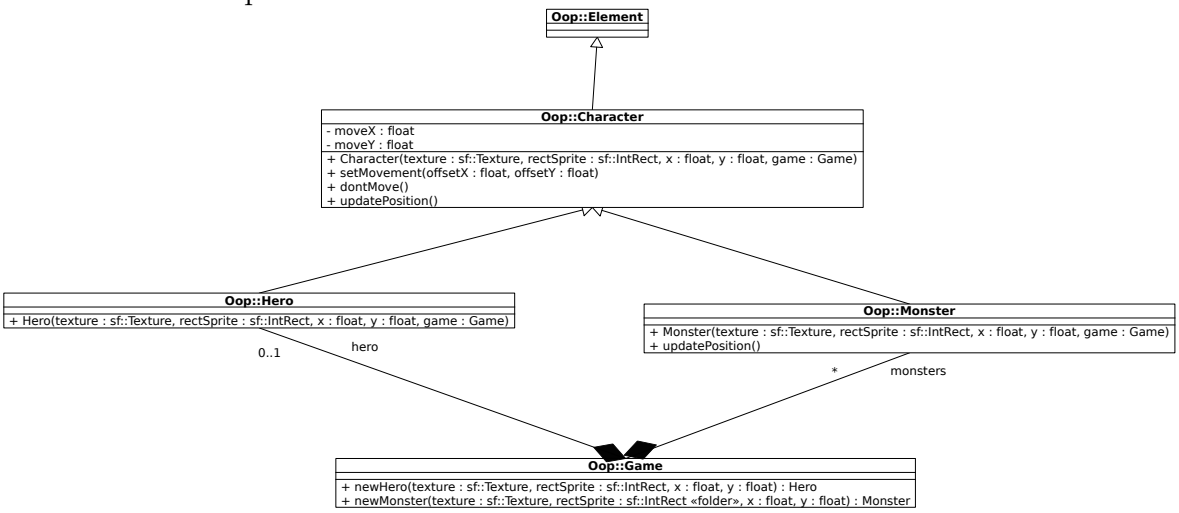
Il ne suffit pas d'ajouter un sprite dans le jeu pour qu'il s'affiche à l'écran. Il faut maintenant demander à la fenêtre d'affichage (`_window` de la classe `Game`) de dessiner le sprite. Pour dessiner un sprite à l'écran, il faut utiliser la méthode `draw(sf::Sprite & sf)` [5] de la classe `sf::RenderWindow` (voir le diagramme UML associé à la classe `Game`). Dans la classe `Game`, c'est la méthode privée `drawScene()` qui a la responsabilité de demander à `_window` l'affichage de tous les sprites quand la méthode principale `run()` le lui demande.

Q10 Dans `Game.cpp`, modifier la méthode `drawScene` pour qu'elle affiche les éléments de décor disponibles dans le jeu. Compiler. Exécuter. Le premier élément de votre décor doit s'afficher à l'écran à l'endroit de votre choix.

Libre à vous d'ajouter autant d'éléments de décor dans votre jeu.

6 Le héros

Votre héros est un sprite qui a la capacité de se mouvoir, tout comme les monstres (décrits dans la section suivante). Ce trait commun au héros et aux monstres est mis en œuvre dans la classe `Character` qui dérive de `Element`.



[5] Attention, la méthode `draw` demande une référence C++ et non pas un pointeur, ainsi si vous disposez d'un pointeur, par exemple `Setting * tree1`, pour afficher l'arbre il faudra écrire `_window->draw(*tree1);`, il faut en effet déréférencer le pointeur pour obtenir une référence C++ sur l'objet.

Q11 Créer la classe `Character` comme spécifiée sur le diagramme de classe. `_moveX`, `_moveY` stockent la distance en x et en y effectuée par le personnage à chaque pas d'itération (boucle dans `Game::run()`). Si les deux attributs sont nuls, le personnage ne bouge pas. `setMovement` est le mutateur de ces attributs. `dontMove()` modifie les attributs pour que le personnage ne bouge pas à la prochaine itération. `updatePosition()` met à jour la position du personnage (voir `sf::Sprite`) en fonction de `_moveX`, `_moveY` pour la prochaine itération.

Q12 Créer la classe `Hero` comme spécifiée sur le diagramme de classe.

Q13 Mettre en œuvre la relation entre la classe `Hero` et la classe `Game` en ajoutant les attributs nécessaires à `Game` et en ajoutant la méthode `newHero`. Remarquez bien la cardinalité de cette relation. Faut-il modifier le destructeur de `Game`?

Q14 La texture des héros est dans le fichier `hero.png`. Chaque personnage de ce fichier est un carré de 64 par 64. Dans le `main()`, charger la texture des héros et définissez une zone rectangulaire pour selectionner le sprite de votre choix et à l'aide de `newHero`, créez le héros de votre jeu.

Q15 Dans `Game.cpp`, modifier la méthode `drawScene` pour qu'elle affiche le héros dans le jeu (comme pour les `Setting`). Attention l'ordre est important, les sprites se dessinent les uns sur les autres dans l'ordre des appels à `_window->draw(...)`. Compiler. Exécuter. Votre héros doit s'afficher à l'écran à l'endroit de votre choix au milieu du décor.

Mais pour le moment, votre héros est toujours immobile. Vous devez le mettre en mouvement. L'idée est la suivante, quand une touche du clavier est pressée la méthode privée correspondante `onRightPressed(), onLeftPressed(),...` est automatiquement lancée. C'est donc à l'intérieur de ces méthodes que l'on va mettre à jour le mouvement du `Hero`, flèche gauche, il va à gauche, flèche bas il va en bas, `s` il s'arrête. Puis à chaque itération de `run`, la méthode `updateScene()` est appelée, c'est dans cette méthode, que l'on va calculer la nouvelle position du héros à afficher avant que `drawScene()` soit appelée pour l'affichage effectif.

Q16 Modifier les méthodes `onRightPressed(), onLeftPressed(),...` pour gérer les mouvements du `Hero`, puis modifier la méthode `updateScene()` pour le calcul de la nouvelle position du `Hero`. Compiler. Exécuter. Votre héros doit s'afficher à l'écran à l'endroit de votre choix au milieu du décor et vous devez pouvoir le contrôler avec les touches fléchées et la touche `s`.

7 Les monstres

Les monstres sont des personnages comme le héros à la seule différence que le mouvement du monstre est aléatoire et non pas guidé. Autrement dit, la méthode `updatePosition()` se comporte différemment sur un monstre. À chaque itération, elle doit fournir une nouvelle position basée sur un mouvement aléatoire. Pour générer un nombre aléatoire, vous pouvez utiliser la fonction `rand()`. Par exemple, l'appel `rand() % 10` va générer un entier aléatoirement entre 0 et 9 (`%` est le modulo).

Q17 Créer la classe `Monster` comme spécifiée sur le diagramme de classe et en particulier, redéfinissez la méthode `updatePosition()`. N'oubliez pas la virtualisation des méthodes dans les classes parentes.

Q18 Mettre en œuvre la relation entre la classe `Monster` et la classe `Game` en ajoutant les attributs nécessaires à `Game` et en ajoutant la méthode `newMonster`. Faut-il modifier le destructeur de `Game`?

Q19 La texture des monstres est dans les fichiers `monster1/2.png`. Chaque monstre de ces fichiers est un carré de 64 par 64. Dans le `main()`, charger la texture des monstres de votre choix et définissez des monstres pour votre jeu, de la même façon que pour le héros.

Q20 Dans `Game.cpp`, modifier la méthode `updateScene()` pour le calcul de la nouvelle position de chaque monstre et modifier la méthode `drawScene` pour qu'elle affiche les monstres.

À cette étape, vous devez voir des monstres se mouvoir aléatoirement dans le décor, et vous devez pouvoir contrôler le héros avec le clavier.

8 De vie à trépas (si vous avez le temps)

Votre héros peut se balader tranquillement dans ce monde plein d'embuches et de monstres sans le moindre souci, la vie n'est pas ainsi faite, ce héros n'est pas immortel, vous devez remédier à cela.

Q21 Dans la classe `Hero`, mettre en place un système de gestion de point de vie (attribut, consultation, modification).

Votre héros doit perdre des points de vie s'il est en contacte avec des monstres ou des éléments du décor. Pour cela vous allez ajouter des collisionneurs (*collider*). Un collisionneur est une zone rectangulaire associée au sprite. Si les collisionneurs de deux sprites s'intersectent, il y a une collision et donc une perte de 1 point de vie si c'est le héros qui entre en collision. Le collisonneur est un `sf::IntRect` défini de façon relative à un sprite. Par exemple, si le sprite est de taille 64x32 alors `sf::IntRect(0,0,64,32)` est le collisonneur qui entoure le

sprite. Mais il peut être plus petit: `sf::IntRect(10,16,44,16)` est un collisonneur qui se situe en bas du sprite.

Q22 Modifier la classe `Element` pour intégrer un nouvel attribut `sf::IntRect _collider` et les méthodes `void setCollider(float left,float top, float witdh, float height)`, `sf::IntRect collider();` qui vont permettre de définir/consulter des collisionneurs pour toutes les classes dérivées.

Le collisonneur est relatif au sprite. Quand le sprite est dans une position donnée à l'écran, implicitement son collisonneur est aussi présent à l'écran relativement à la position courante du sprite. Cete zone rectangulaire décrite en coordonnées globales est ce que l'on appelle l'enveloppe du sprite (*bounding box*).

Q23 Modifier la classe `Element` pour intégrer la méthode `sf::IntRect boundingBox()` qui retourne la zone où se trouve le collisionneur en coordonnées absolues en fonction de la position courante du sprite.

Q24 Modifier `Game.cpp` de telle manière que si l'enveloppe du héros est en collision avec l'enveloppe d'un autre élément à un instant donné, il perde un point de vie. Le héros est en collision si son enveloppe intersecte celle d'un autre élément (voir la documentation de `sf::IntRect::intersect`).

Q25 Modifier `Game.cpp` pour que le jeu s'arrête dès que le `Hero` n'a plus de point de vie.

Q26 Ajouter dans `main()` les collisionneurs pour le héros et les monstres. Pour les éléments de décor, des collisionneurs sont disponibles dans l'objet `data`.

9 Et il danse, danse, danse... (si vous avez le temps)

Pour le moment, on a toujours associé une seule image à un sprite, mais on peut très bien modifier la zone rectangulaire de ce sprite pour changer la façon dont il s'affiche à chaque mouvement. Cela permet de simuler une animation.

Q27 Ajouter un moyen pour créer une séquence d'animation avec les sprites quand le héros se déplace pour le faire danser. Pour cela utiliser la planche de sprites du héros et changer l'image en boucle.