Rapport concernant la SAE 1.02 G1_A

I- Constantes

Les constantes sont une part essentielle dans le développement d'un jeu, elles permettent de regrouper des valeurs utilisées plusieurs fois à des endroits différents et ainsi avoir un bon aperçu des ressources utilisées dans le jeu.

Parmi les constantes, nous enregistrons la taille de la fenêtre qu'il est plus facile d'appeler depuis le fichier des constantes plutôt que de devoir faire passer partout en paramètre la fenêtre et devoir utiliser la méthode `getWindowSize` à chaque fois.

Nous avons aussi la totalité des chemins relatifs pour accéder aux ressources du jeu, que ce soit pour le fichier config.yaml, leaderBoard.txt ou tous les sprites.

Nous enregistrons aussi des informations sur le jeu telles que la vitesse du joueur, son nombre de vies au début de la partie ou encore le temps qu'il faut pour recharger après avoir tiré. Ceci nous permet de regrouper des variables utilisées à différent dans le code et ainsi changer sa valeur de partout.

II- Jeu

1. Box

La structure de donnée **Box** est une structure qui regroupe 2 vecteurs de la bibliothèque MinGL2, elle représente donc un rectangle.

```
struct Box {
    nsGraphics::Vec2D firstPosition;
    nsGraphics::Vec2D secondPosition;
}; // struct Box
```

On peut lui associer quelques pseudo-méthodes telles que `areColliding` qui permet de déterminer si un vecteur ou une autre Box se situe dans ou à cheval sur une Box donnée.

2. Entity

La structure de donnée **Entity** est une structure qui est utilisée dans le jeu pour chaque objet à gérer. Elle s'écrit sous la forme :

```
struct Entity {
    EntityType type;
    nsGui::Sprite sprite;
    nsGraphics::Vec2D spriteSize;
    int lifePoints;
    nsBox::Box bounds;
    int speed = 10;
    nsGraphics::Vec2D direction = nsGraphics::Vec2D();
    bool canGoOutOfBounds = false;
}; // struct Entity
```

On lui associe les pseudo-méthodes d'affichage dans la fenêtre (`dispEntities`), de mouvement de l'entité (`moveEntities`) qui déplace le sprite selon sa direction et sa vitesse. L'entité possède aussi des limites (de type Box) qui permettent soit de retenir l'entité

pour qu'elle ne dépasse pas, soit de la supprimer lorsqu'elle est en dehors. (voir areColliding dans Box). Une entité possède aussi un type appartenant à l'énumération :

```
enum EntityType {
    SHIP,
    SHIP_BULLET,
    SHIELD,
    INVADER,
    INVADER_BULLET
}; // enum EntityType
```

Ce type est utile pour être comparé à d'autres entités, notamment pour gérer leurs collisions. Nous utilisons un masque qui permet de déterminer quel type d'entité doit collisionner avec quel type :

```
const map<EntityType, vector<EntityType>> entitiesCollider {
    {SHIP,          {INVADER, INVADER_BULLET}},
    {SHIP_BULLET,   {SHIELD, INVADER, INVADER_BULLET}},
    {SHIELD,        {SHIP_BULLET, INVADER, INVADER_BULLET}},
    {INVADER,       {SHIP_BULLET}},
    {INVADER_BULLET, {SHIP, SHIP_BULLET, SHIELD}}
}; // map entitiesCollider
```

Ainsi, un envahisseur ne peut être attaqué que par les projectiles du vaisseau (joueur), tandis que les projectiles du joueur peuvent être détruits par les boucliers, les envahisseurs ainsi que les projectiles de l'envahisseur.

3. Space Invaders

Pour ce qui est de l'organisation du jeu en lui-même, à chaque frame (tour de boucle) on déplace chaque entité du jeu selon son propre vecteur direction que l'on redimensionne à une longueur égale à sa vitesse, de cette manière les entités se déplacent de manière réaliste quelque soit sa direction.

Par la suite, nous faisons tirer les entités, pour le tir du joueur nous regardons si le booléen `canShoot` est vrai et que l'utilisateur appuie sur la touche de tir. Alors nous instancions une nouvelle entité de type *SHIP_BULLET* qui aura une direction constante (direction constante → déplacement en ligne droite). Une fois que le joueur a tiré, on place le booléen `canShoot` à faux. On le repassera à vrai au moment où le chronomètre `lastShot` aura atteint la durée nécessaire pour recharger un tir. Pour le tir des envahisseurs, le même procédé est utilisé à un détail près, là où on attendait le booléen et la touche pour le joueur, on attend uniquement que le booléen soit vrai car il n'y a pas de décision de la part des ennemis, ils tirent dès qu'il peuvent, on choisit alors un envahisseur au hasard comme point de départ du nouveau projectile (de type *INVADER_BULLET*) qui aura aussi un déplacement constant mais vers le bas.

Ensuite, nous vérifions pour chaque entité sa collision avec chacune des autres entités via le masque décrit dans la partie sur les entités. Ainsi, pour deux entités données, on retire une vie (dans la structure Entity) à la première entité seulement si l'autre entité peut collisionner avec elle.

Pour finir il suffit de supprimer toutes les entités qui ont un nombre de vies inférieur ou égal à 0 pour les faire disparaître de l'écran puis on affiche toutes les entités.

III- Fichiers tiers

1. config.yaml

Le fichier config.yaml contient les paramètres de configuration du jeu sous la forme:

`nomDuParametre:valeur`

Lorsqu'on démarre une partie, ce sont ces paramètres qui sont lus dans le fichier afin de définir la manière de jouer. On lit le fichier ligne par ligne puis on cherche pour chaque ligne le séparateur afin de différencier le nom du paramètre de sa valeur.
On accède également à ce même fichier en mode écriture lorsque l'utilisateur change la configuration des paramètres à l'aide du menu. C'est alors un dictionnaire de deux chaînes de caractères qui est envoyé afin de remplacer la ligne correspondante dans le fichier avec la nouvelle valeur.
L'ensemble de ces fonctions sont regroupées dans le fichier `file.cpp`.

2. leaderBoard.txt

Le jeu offre aussi la possibilité au joueur, après avoir terminé la partie, de stocker son score associé à son pseudonyme. Ces informations sont stockées dans le fichier `leaderBoard.txt`.
Ces informations sont triées dans le fichier par ordre décroissant (le score le plus élevé en premier dans le fichier). Pour ce faire, l'ensemble des couples (nom d'utilisateur, score) sont stockés dans un tableau de chaînes de caractères puis séparés en deux tableaux distincts (par rapport au séparateur ":"). Un tableau contient les scores et un autre les noms d'utilisateurs et une fois triés de manière à insérer le nouveau score à la bonne place, ces deux tableaux sont réassemblés dans le tableau d'origine qui servira ensuite à enregistrer ces modifications dans le fichier `leaderBoard.txt`.

IV- Menus

1. Button

La structure de donnée **Button** permet de créer un bouton cliquable qui n'est en fait que la superposition d'un rectangle et d'un texte venant de la bibliothèque MinGL2 :

```
struct Button {
    std::string content;     // just used for initialization

    nsGui::Text text = nsGui::Text(
        nsGraphics::Vec2D(),
        content,
        nsConsts::textColor,
        nsConsts::textFont,
        nsGui::Text::ALIGNH_CENTER,
        nsGui::Text::ALIGNV_CENTER
    );
    nsShape::Rectangle rect = nsShape::Rectangle(
        nsGraphics::Vec2D(),
        nsGraphics::Vec2D(9*content.size()+64, 64),
        nsConsts::btBgColor,
        nsConsts::btBorderColor
```

```
    );
}; // struct Button
```

On allie à cette structure la pseudo-méthode `isPressed` qui permet lorsqu'on lui donne le gestionnaire d'événements de la fenêtre venant de la bibliothèque MinGL2, de dire si le bouton a été cliqué (par un clique gauche) durant la frame.
On ajoute aussi à cette structure la fonction nommée `placeBtns`, qui lorsqu'on lui donne une liste de boutons, les place dans la fenêtre verticalement.

2. Scene

La structure de donnée **Scene** permet d'enregistrer toutes les informations d'une scène telles que son fond, ses boutons ou encore ses entités et ses textes :

```
struct Scene {
    nsGui::Sprite background;
    std::vector<nsButton::Button> buttons;
    std::vector<nsEntity::Entity> entities;
    std::vector<nsGui::Text> texts;
}; // struct Scene
```

Chaque scène peut se faire initialiser (on met des valeurs dans ses attributs) et peut se faire afficher grâce à la fonction `displayScene` qui va simplement afficher le fond puis afficher chaque entité de la scène, et enfin afficher ses boutons et ses textes.

Ainsi, lorsqu'on allie une fonction de calcul à la scène, on peut modifier chaque élément et après afficher la scène.

3. Main

Maintenant que chaque structure a été introduite et le fonctionnement du jeu expliqué, il ne reste plus qu'à savoir comment les données sont utilisées dans le programme principal et comment les scènes sont liées entre elles.

Pour commencer, au démarrage du jeu, on crée une nouvelle fenêtre avec la bibliothèque MinGL2, puis on instancie/initialise toutes nos scènes, c'est-à-dire que l'on crée tous les boutons et tous les textes que l'on peut déjà afficher à l'avance. On crée aussi une variable nommée `currentScene` qui permet de mémoriser la scène actuelle.

Une fois cette étape terminée, nous entrons dans la boucle principale du jeu et on 5 effectue un test sur `currentScene` qui permet de calculer (ie. regarder si des boutons sont cliqués ou si des entrées claviers sont faites) et d'afficher la scène. Ainsi, lorsqu'on démarre le jeu (on arrive sur le menu principal), dès lors qu'on presse un bouton, on peut effectuer des calculs et ainsi changer la valeur de `currentScene` pour afficher une autre scène à la frame suivante.

NB, lorsqu'on termine une vague d'envahisseurs (on a tué tous les ennemis présent dans la liste d'entités de la scène de jeu), le jeu appelle une nouvelle fois la procédure d'initialisation de la scène de jeu ce qui permet de remettre de nouveaux envahisseurs qui seront plus nombreux et plus rapides.