M2103 : Projet de Programmation Orienté Objet

PROJET RUNNER

NÉDÉLEC Guillaume

--

MARCILLOUX Nicolas

Groupe B”

Année 2015-2016

Enseignant Responsable : Romain Giot

Sommaire

I. Introduction

Présentation du projet

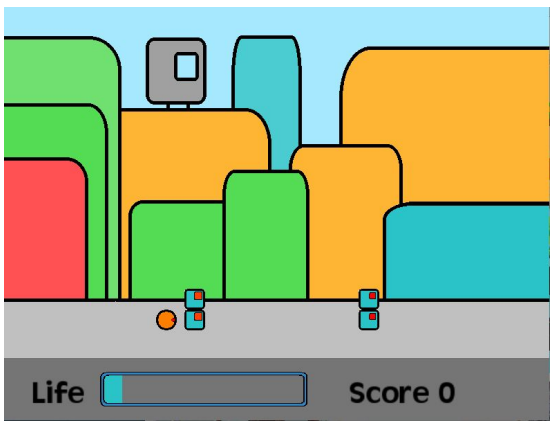
Ce projet a été réalisé en binôme par Nicolas MARCILLOUX et Guillaume NEDELEC dans le cadre du module M2103 de l'IUT informatique de Bordeaux. Le temps de réalisation accordé pour ce projet était d'environ 4 mois (début février - fin mai) .

L'objectif de celui-ci était de réaliser un jeu vidéo de type "Runner" à l'aide de nos connaissances et des notions vues dans le module de programmation orientée objet.

Pour ce faire, le jeu a été programmé en C++ en intégrant une bibliothèque graphique externe : la SFML (*Simple and Fast Multimedia Library*). Ce projet nous a permit aussi d'utiliser les outils que propose la bibliothèque Standard C++ : la STL (Standard Template Library).

Principe

Les jeux de type "Runner" sont des jeux d'arcade où, de manière générale, le joueur doit éviter des obstacles et arriver le plus loin possible pour collecter un maximum de points. La difficulté de ce jeu vient du fait que, plus le jeu avance, plus la vitesse s'intensifie et rend ainsi l'esquive d'obstacles de plus en plus ardue. Le jeu peut être infini (le joueur doit forcement mourir), ou alors il est composé de niveau comme par exemple "Geometry Dash". Néanmoins, ce genre de jeu a souvent des bonus permettant au joueur de cumuler un maximum de points et de pouvoir avancer plus loin. Ce type de jeu étant en soit assez simple, il permet aux développeurs de conceptualiser des fonctionnalités nouvelles permettant d'améliorer l'expérience de jeu.



extrait de la démo fournie par l'IUT

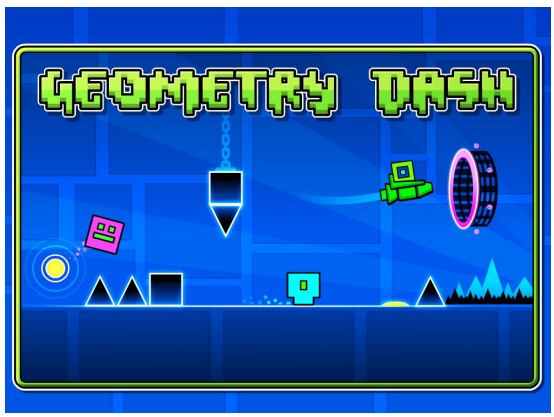


image du jeu "Geometry Dash"

Fonctionnalités du projet

Un certain nombre de fonctionnalités nous étaient indiquées (du minimal demandé, aux fonctions bonus). Voici la liste des fonctionnalités que nous avons implémenté dans notre projet (le jeu est jouable au clavier alors que la navigation dans les menus se fait à la souris):

• Un écran d'introduction du jeu qui présente le cadre du projet ainsi que les développeurs

• Un écran pour le menu principal permettant de :

- Lancer une partie
- Accéder aux options du jeu
- Accéder aux règles du jeu (contenant 3 pages d'explications)
- Accéder au classement des meilleurs scores
- Quitter le jeu

• Le menu des options permet de modifier la langue du jeu (Anglais, Français, Espagnol ou Allemand). On peut aussi modifier la difficulté (Facile, Normal ou Difficile) qui influe sur la vitesse de départ du jeu.

• Un menu de pause est accessible pendant la phase de jeu grâce à la touche "Echap". Ce menu permet un retour au menu principal, un accès au règles ainsi qu'un retour au jeu.

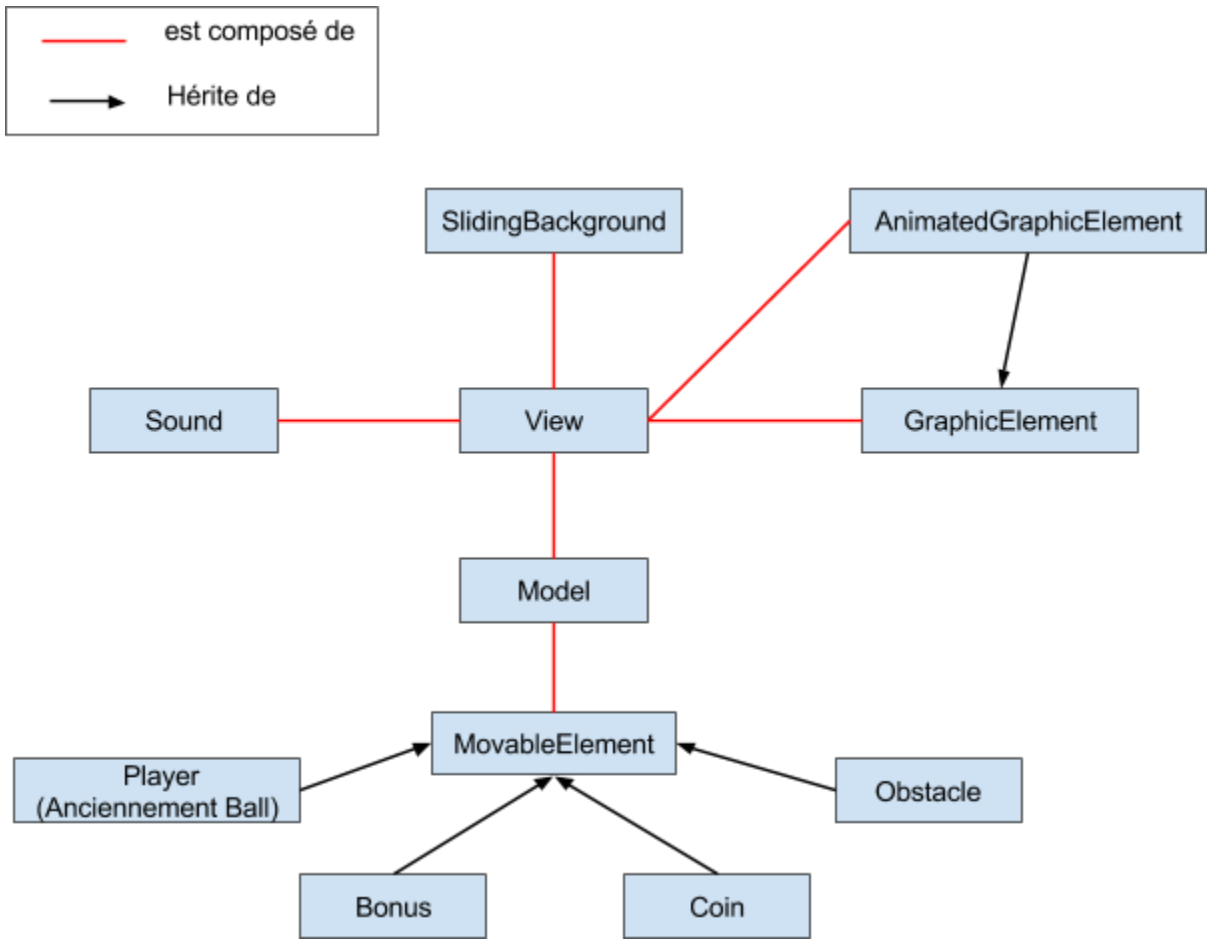
• La phase de jeu contient plusieurs éléments :

- Un personnage se déplaçant horizontalement et qui a la possibilité de sauter
- Un arrière-plan qui défile
- Des obstacles qui apparaissent aléatoirement en hauteur et au sol. Certains sont des obstacles mobiles alors que d'autres défilent normalement (à la vitesse défilement de l'arrière-plan). Les obstacles font perdre 10,15 ou 20 points de vies
- Des pièces à récupérer augmentant le score de 100 points (ou 200 si le bonus double points est actif)
- Un score qui s'incrémente au cours du temps ainsi qu'en récupérant des pièces.
- Une barre de vie initialisée à 100 points de vie
- 5 types de bonus : bonus de vie, double saut, invincibilité, double point, nucléaire (détruit tous les éléments du jeu)
- Un écran de transition à la fin du jeu (le "Game Over" final)

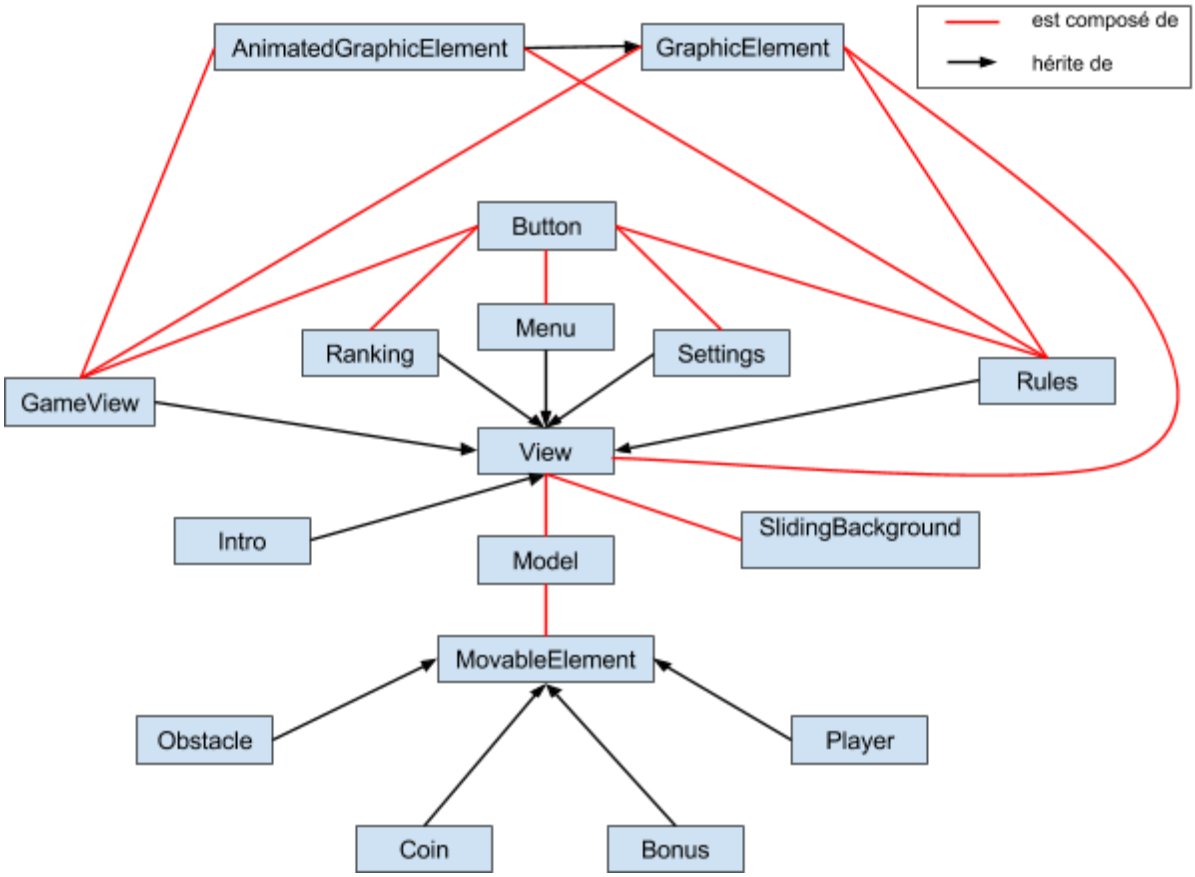
• Une musique est présente dans le projet ainsi que des effets sonores durant la phase de jeu. Il est évidemment possible de les désactiver dans les options ou dans les autres menus proposés, via des boutons indicatifs.

II. Présentation de l'architecture

Nous avons choisi de réaliser une architecture Modèle-Vue qui nous permet de distinguer nettement la partie modèle de données avec toutes les informations inhérentes au jeu et la partie application offrant une illustration graphique du modèle de données.
Au commencement, nous avons réalisé la conception du projet bien avant de passer à la programmation. Voici donc une ébauche de ce à quoi devait ressembler les relations entres les classes de notre projet lors de la conception.



Maintenant voilà l'ébauche des relations du projet que nous avons programmé. Pour plus de détails vous pouvez consulter la documentation doxygen associée sur :
http://info-morgane.iut.u-bordeaux.fr/perso/2016/gnedelec001/Code%20Doxygen/html/index.html



On peut observer que depuis la conception, seulement la partie "Vue" du jeu à été modifié. En effet posséder une seule classe "Vue" était très lourd dans le code. Nous avons donc pris l'initiative de séparer chaque genre de vue pour avoir un code plus propre et aussi beaucoup plus lisible. De plus l'ajout d'une classe "Bouton" a permis de beaucoup alléger le code.

Appel des différentes vues

Au lancement du jeu, une introduction est appelé (et ne le sera plus tout au lng de l'exécution). Arrivée à son terme elle laisse place au menu principal. A partir de celui ci, plusieurs possibilités sont envisageable : Lancer le jeu, accéder aux options, aux règles du jeu, au classement ou encore au licence utilisé pour notre projet.
Quelque soit l'issue choisi, celle-ci vous ramènera au menu principal lorsque vous quitterez la phase que vous avez choisie.
Le jeu peut-être stopper à n'importe quel moment via le bouton fermer de la fenêtre.

III. Présentation des classes

Voici un descriptif des classes utilisées dans notre projet (les héritages sont indiqués dans le graphe précédent).

MovableElement : Définit les propriétés de tous les objets mobiles du jeu comme les pièces, les obstacles ou encore le joueur par exemple. Les propriétés définies sont la position x et y de l'objet, sa hauteur, sa largeur ainsi que sa vitesse horizontale et sa vitesse verticale. Les fonctions associées sont liées au déplacement de l'objet et les collisions possibles.

Player : Une classe fille de "MovableElement". Elle représente le joueur, elle a donc un attribut pour son niveau de vie. De plus cet élément est capable de sauter, cela a impliqué l'ajout d'une nouvelle méthode. Les objets de cette classe ont la particularité de pouvoir bouger à la volonté du joueur, leur vitesse n'est pas constante.

Obstacle : Classe fille de "MovableElement", elle représente les ennemis du joueur, de vitesse non modifiable, les obstacles diffèrent selon leur type. Un attribut pour définir le type est donc présent. Selon le type, les dégâts engendré seront différents et les dimensions de l'objet le seront aussi.

Coin : Comme pour les obstacles, cette classe hérite de "MouvableElement" et sa vitesse n'est pas interchangeable. Cette classe ne comporte aucune particularité en plus que "MovableElement".

Bonus : Cette classe est similaire à celle des Obstacles puisqu'elle hérite de "MovableElement", et un attribut définissant le type de bonus a été ajouté. Les types de bonus sont un bonus d'invincibilité qui empêche la perte de vie pendant un certains temps, le bonus double saut, un bonus permettant de récupérer quelques points de vie, un bonus qui double la valeurs des pièces à récupérer et un bonus nucléaire qui supprime tous les éléments presents à l'instant où le bonus a été récupérer).

Model : Cette classe est le coeur du jeu. C'est dans cette classe que sont appelés tous les éléments (joueur, obstacles, pièces, bonus). Chaque étape du jeu se passe dans le modèle. Il regroupe les informations nécessaires au jeu (ex : le score) ainsi que tous les algorithmes qui en permettent le bon fonctionnement (apparitions aléatoires d'objets, déplacement des objets, ect…).

View : Cette classe est la classe mère de toutes vues qui vont être utilisé pour ce projet. Elle comprend les éléments graphiques qui apparaîtront tout au long du jeu comme le curseur, les boutons pour la musique ou bien les arrière-plans défilants.

Intro : Cette classe réalise l'écran d'introduction au lancement du jeu. Cet écran est simplement composé de différents textes informatifs. Le seul point technique est l'effet de "fondu" réalisé avec le changement d'opacité d'un écran noir à chaque frame.

Menu : Cette classe héritant de "View" permet l'affichage et la gestion de tous les évènements liés au menu principal (clic sur les boutons, changement de couleur ect…).

Rules : Cette classe regroupe l'ensemble des informations liés au jeu. Elles regroupent les informations pour aider l'utilisateur à correctement jouer.

Settings : Cette classe permet d'afficher l'écran des options et ainsi de gérer les évènements permettant de modifier ces réglages.

Ranking : Cette classe permet de lire le fichier de scores afin de les retranscrire graphiquement dans la fenêtre pour l'utilisateur.

GameView : Cette classe est la partie graphique du modèle. En effet, elle recupère toutes les informations du modèle pour les retranscrire graphiquement. (Images des obstacles, des pièces, des bonus, affichage du score, ect…).

GraphicElement : Cette classe permet de charger des images et de les afficher dans la fenêtre. On peut aussi redimensionner les images avec les données voulues.

AnimatedGraphicElement : Cette classe héritant de "GraphicElement" permet elle aussi de charger et d'afficher des éléments graphiques dans la fenêtre mais permet, en prime, d'animer ces éléments graphiques pour que l'image soit mouvante (à la manière d'un gif).

SlidingBackground : Cette classe permet de charger les arrière-plans et de les faire défiler en boucle.

Button : Cette classe permet la génération et la modification de boutons créés avec les outils de la SFML (les sf::RectangleShape et les sf::Text).

IV. Détails des Algorithmes

Saut :

```
'Pression sur HAUT'
        -> saut = vrai
Si (saut == vrai)
        -> Fonction Saut

Action Saut {
        Si (debutSaut == vrai ) {
                vitesse verticale = vitesseRapide
                debutSaut = faux
        }
        Si  (hauteurPerso  < HauteurMaximaleAAtteindre et chute == faux ){
                vitesse verticale *=  0.9 (gravité) //Deceleration
                hauteur += vitesse verticale
        }
        Si (hauteur <= HauteurMaximale)  {
                chute = vrai
        }
        Si (chute == vrai et finSaut == faux)  {
                vitesse verticale /=  0.9 (gravité) //Acceleration
                hauteur -= vitesse verticale
                Si hauteur <= HauteurSol {
                        vitesse verticale = 0
                        hauteur = 470
                        chute = faux
                        finSaut = vrai
                        debutSaut = vrai
                }
        Si finSaut == vrai {
                saut = faux
        }
}
```

La fonction de saut utilise une variable de gravité de 0.9. Cette valeur permet de diminuer la vitesse du personnage pendant la montée du saut et de l'accélérer durant la chute. Cela nous a donc permis de générer une fonction proche du réel. L'utilisation de booléens nous permet de différencier chaque étape du saut.

Gestion des élément mobiles :

Suppression des éléments du set 'poubelle'
Pour chaque obstacle {
    Si (collisionBalle == vrai) {
        perte de vie selon l'obstacle
        insertion de l'obstacle dans la 'poubelle'
    }
    Sinon si (l'objet sort de l'écran) {
        insertion de l'obstacle dans 'poubelle'
    }
    Sinon { déplacement de l'obstacle}
}
Pour chaque bonus {
    Pour chaque obstacles {
        Si (collisionObstacle == vrai) { insertion du bonus dans 'poubelle' }
    }
    Si (collisionBalle == vrai) {
        effet du bonus
        insertion de du bonus dans 'poubelle'
    }
    Sinon si (l'objet sort de l'écran) { insertion du bonus dans 'poubelle' }
    Sinon { déplacement du bonus }
}
Pour chaque pièce {
    Pour chaque obstacles {
        Si (collisionObstacle == vrai) { insertion de la pièce dans 'poubelle' }
    }
    Pour chaque obstacles {
        Si (collisionBonus == vrai) { insertion de la pièce dans 'poubelle' }
    }
    Si (collisionBalle == vrai) {
        incrementation du score
        insertion de la pièce dans 'poubelle'
    }
    Sinon si (l'objet sort de l'écran) { insertion de la pièce dans 'poubelle' }
    Sinon { déplacement de la pièce}
}

Cette fonction gère les évènements liés aux éléments mobiles.
Que ce soit un obstacle, une pièce ou un bonus, la fonction réalise la même chose. Si l'objet A est en collision avec un autre objet B cela signifie qu'il est apparu au meme endroit que l'objet A. On place donc l'objet A dans un set dit "poubelle" qui se vide à chaque tour de boucle.
Sinon, quand l'objet rentre en collision avec le joueur, alors il y a un impact selon le type (les pièces augmentent le score ect…) puis rentre dans la "poubelle".
Si il n'y a aucune collision, l'objet se déplace juste.

Génération de délais d'apparition

```
unsigned int Model::getDuration(Element elem) const
{
    if (elem==Ennemies) //Pour les obstacles
    {
        return (MIN_APPARITION_OBSTACLE + 2*(int)(_speedStaticElement)) + (rand() % ((MAX_APPARITION_OBSTACLE + (int)(_speedStaticElement)) - MIN_APPARITION_OBSTACLE));
    }

    else if (elem==Points) // Pour les pièces
    {
        return MIN_APPARITION_COIN + (rand() % (MAX_APPARITION_COIN - MIN_APPARITION_COIN));
    }

    else if(elem==PowerUp)// Pour les Bonus
    {
        return MIN_APPARITION_BONUS+ (rand() % (MAX_APPARITION_BONUS- MIN_APPARITION_BONUS));
    }
    else
        return 0;
}
```

Dans cette fonction, on cherche à obtenir une durée (en nombre de frames) pour l'apparition du prochain élément (Obstacle, Bonus ou Pièce).
Pour les pièces et les bonus, le délai et choisi entre le nombre maximale de frames et le nombre minimal (choisi par les développeurs).
Par contre, pour les obstacles, la borne maximale diminue en fonction de la vitesse, ce qui a pour effet de faire apparaître de plus en plus d'obstacles à chaque fois que la vitesse augmente.

V. Conclusion

Ce projet fût très enrichissant et nous a permis de développer nos compétences en programmation. Ce fut un véritable plaisir de pouvoir laisser libre court à notre imagination en termes de design ou encore de fonctionnalités.

Nous avons réussi à implémenter les fonctionnalités minimales demandées ainsi que certaines fonctionnalités supplémentaires comme par exemple des nouveaux bonus. Nous avons aussi créé un petit scénario disponible dans la catégorie “Règles du jeu” ce qui fût, pour nous, très amusant. Malheureusement nous n’avons pas eu le temps d’implémenter un magasin ou de proposer plusieurs chartes graphique. Néanmoins les idées était présentes et nous avons préféré proposer moins de fonctionnalités mais implémenter correctement plutôt que de mettre un maximum de choses sans soigner le code.