Rapport de projet C++

THERE IS NO PLANET B

Maupu Noah Raimbaud Quentin | C++ | 22/01/2022

Table des matières

Description de l'application développée

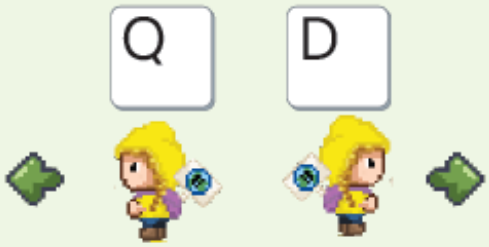
HOW TO PLAY

MAKE THE WORLD GRETA GAIN

01 VOUS INCARNEZ GRETA THUNBERG

Vous luttez contre les climatosceptiques qui rapprochent peu à peu là Terre de sa destruction. Il n'y a pas de planète B : gardez bien cela en tête et faites tout votre possible sauver votre planète A. Combattez les zombies capitalistes, et mettez fin au règne du tyran !

02 POUR SE DÉPLACER

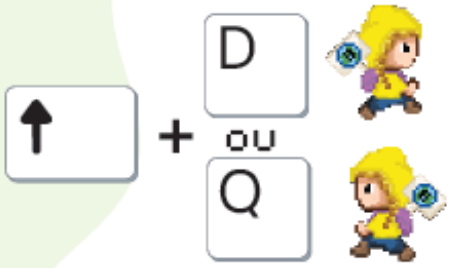


Appuyer sur la touche "Q" pour aller vers la gauche.

Appuyer sur la touche "D" pour aller vers la droite.

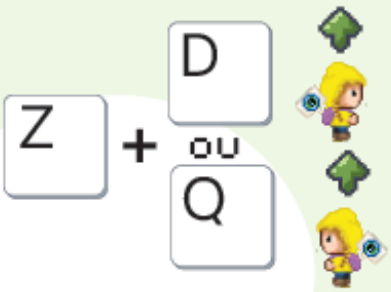
03 ACCÉLÉRER

Maintenez "Shift" pour courir.



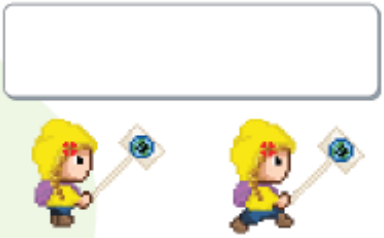
04 SAUTER

Sautez en appuyant sur "Z".



05 ATTAQUER

Donnez un coup de pancarte en appuyant sur "Espace"

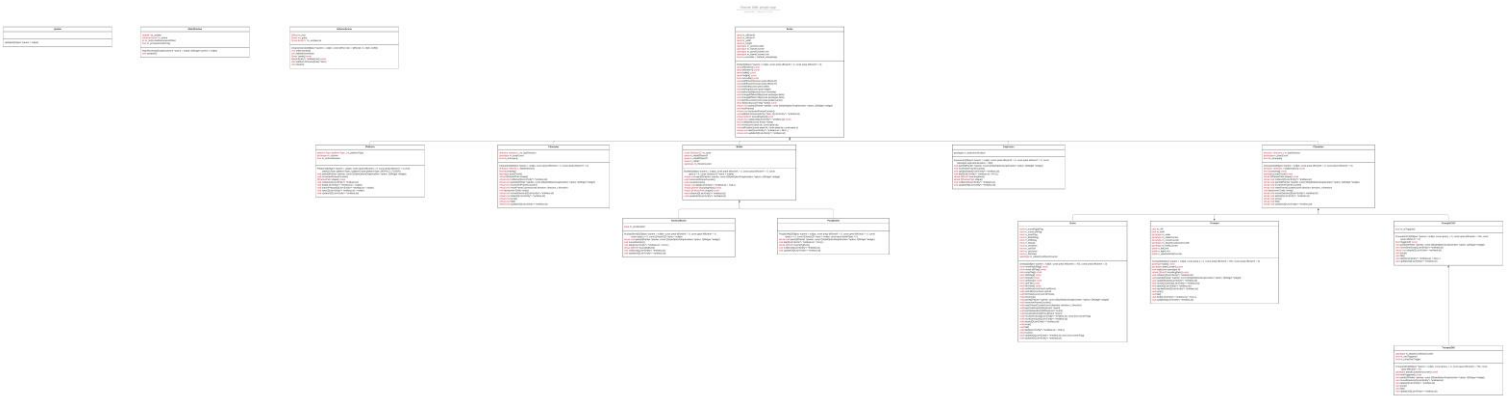


06 INVINCIBLE MODE

Le jeu est assez difficile et demande de bien nombreux essais avant d'y arriver à bout, vous pouvez devenir invincible en appuyant sur "G".

BONNE CHANCE, NOUS COMPTONS SUR VOUS POUR TOUS NOUS SAUVER !

Diagramme UML de l'application



Explications des choix et contraintes logicielles

Nous avions une idée : développer un jeu de plateforme, dont le personnage jouable serait Greta Thunberg, et l'ennemi à abattre, Donald Trump (nous tenons à préciser qu'il s'agit là d'un concept humoristique et dénoue d'intérêt : nous ne cherchons pas à dénigrer l'image de l'un des deux protagonistes précédemment cité).

Il nous fallait faire une interface graphique (première grosse contrainte). Beaucoup de bibliothèques donnent les outils nécessaires pour en faire une, les plus connues étant STL, SFML et Qt.

C'est sur Qt que notre choix s'est porté. Multiplateforme mais initialement à visée plus applicative que vidéo-ludique, Qt nous donne la praticité d'une compilation portable et adaptative, mais impose des contraintes de codes conséquentes.

Il n'y a pas d'outil "simple" pour démarrer un plateformeur sur Qt. Il nous a fallu créer notre physique et notre caméra à partir de formules mathématiques.

Nous avons commencé par rendre possible le déplacement d'un rectangle avec le clavier, puis nous avons intégré des « sprites » (graphismes) au rectangle pour en faire un personnage toujours déplaçable. Vient ensuite la création d'une plateforme faisant office de sol, et d'une intégration de gravité. Notre personnage pouvait tomber, nous étions satisfaits.

La suite s'est enchaînée bien vite : ajout d'une caméra, ajout des sauts, de l'attaque, d'autres ennemis, du boss final et de la musique.

Une précision notable sur les déplacements est que notre personnage se déplace bel et bien quand la caméra ne peut plus le suivre sans dépasser les limites de la carte (c'est à dire, pour un x entre 0 et la moitié de la largeur de votre écran). Dans tous les autres cas, Greta reste au centre de l'écran, et ce sont tous les autres éléments graphiques qui se déplacent dans la direction opposée, de sorte à avoir une caméra toujours centrée sur notre personnage, gagnant ainsi en fluidité. Ce "non" déplacement de Greta implique de créer une variable "x efficace" qui simule sa position dans le niveau si elle s'était contentée d'avancer. Elle est utilisée par beaucoup de nos classes pour des calculs relatifs à la "position effective" de Greta.

Explications de l'architecture du projet

Enfin, avant de vous partager l'UML de notre projet, parlons un peu de l'architecture de celui-ci.

Qt propose plusieurs classes pour développer une interface graphique, mais la plus propice au jeux vidéo est, à notre sens, QGraphicsItem et toutes les classes filles qui en découlent, en particulier QGraphicsPixmapItem. Cette dernière consiste en un objet dessinable possédant une zone de collision, zone de dessin et une fiche de sprites (qui permettront d'animer l'objet graphique en question). C'est pourquoi nous avons commencé par créer une classe "Entity" abstraite héritant de QGraphicsPixmapItem (et de QObject, pour avoir accès à beaucoup trop de fonctions très importantes de Qt).

Cette classe est un élément graphique avec collision et « sprite » ayant des attributs et méthodes supplémentaires propres à notre jeu. Nous voulions en faire une classe mère "base" de tout futur objet, possédant des méthodes abstraites redéfinies par les classes filles.

La classe "Platform", hérité directement de cette classe "Entity", et consiste en un élément graphique à collision situé à des coordonnées précises.

La classe "Bullet" est le missile tiré par l'un des ennemis. Elle hérite de "Entity" comme toutes classe graphique et est héritée par "PoopBullet", qui modifie le sprite et redéfinie comportement du projectile à l'impact. Enfin, "PoopBullet" est héritée par "NuclearBomb", tirée par le même ennemi que sa classe mère (i.e : "Trumpet", dont le détail vient plus bas).

La classe "Character" hérite de "Entity" et est toujours une classe abstraite, qui définit tout "être vivant" du projet, que ce soit Greta (qui en hérite directement) ou tous les ennemis.

La classe "Trumpet" hérite de "Character" et représente le boss final du jeu. Les méthodes virtuelles et virtuelles pures sont redéfinies, et notre boss est fonctionnel. Il possède plusieurs phases, et utilise soit des "PoopBullet" (classe décrite plus tôt), soit des "NuclearBomb", décrite plus tôt également.

En parlant d'ennemis, la classe "TrumpetCAC" hérite de "Character" et est un zombie qui se déplace et vous tue s'il vous touche ("CAC" ou "corps à corps"). Elle est héritée par autre classe d'ennemi, "TrumpetDIS", qui représente les business man zombies qui vous attaquent avec des billets. Ce sont eux qui tirent des "Bullet", une classe expliquée précédemment.

"Explosion" hérite de "Entity" et consiste uniquement en une explosion, à une position donnée et animée via sa fiche de sprites.

Nous avons aussi créé deux classes "InGameScene" et "MainWindow" héritant respectivement de "QGraphicsScene" et "QGraphicsView". Elles servent, une fois mises ensemble, à afficher la scène du jeu. "MainWindow" possède aussi la fonction principale du jeu, celle qui est appelée 62.5 fois par secondes : "updateX", qui met à jour le jeu et l'affichage en fonction des différents inputs.

Enfin, la classe "Update" hérite de la classe "QTimer" et permet d'appeler 62.5 fois par seconde la fonction "updateX" de la classe "MainWindow".

Mettre en valeur l'utilisation des contraintes

Vous connaissez l'architecture de notre projet, détaillons maintenant la pertinence de l'intégration de vos contraintes :

- En ce qui concerne les 8 classes, nous ne nous sommes pas spécialement forces : nous avions créé autant de classes que nécessaire à nos yeux.

- Pour les 3 niveaux de hiérarchie, idem, nous n'avons pas travaillé en connaissance de cette contrainte. Sans compter toutes les classes héritées par notre classe mère "Entity", les trois niveaux de hiérarchie pour atteindre "Greta" sont très justifiées. Nous avons des éléments graphiques qui ne sont pas des personnages, et des personnages qui sont des ennemis ou non.

- Nous avions déjà plus de deux fonctions virtuelles pures dans notre classe "Entity". Tout d'abord, la fonction "die" est appelé lorsqu'une entité prends un coup. Son comportement est différent pour chaque entité, pour chaque classe, et chacune d'entre elle doit le redéfinir. Un autre exemple et le plus parlant, la fonction "updateX" (qui n'est pas la même que celle présente dans "MainWindow", mais est appelée dans celle-ci). Cette fonction contient toutes les fonctions appelées par une entité à chaque frame pour l'actualiser, prendre en compte son comportement (et potentiellement ses inputs). Elle est nécessaire à toute entité que l'on souhaite faire vivre, et donc doit être redéfinie dans toutes nos classes filles. D'autres fonctions purement virtuelles sont présentes dans notre code source, toutes justifiées également.

- Nous n'avions pas surchargé le moindre opérateur, il fallait donc en rajouter pour répondre à la contrainte. La première et la plus simple est la surcharge de l'opérateur "<<" pour la classe "QDebug", qui corresponds à la surcharge du même opérateur pour la classe "ofstream". Le second opérateur surcharge est un peu plus technique : il s'agit de l'opérateur "+=" pour la classe "Entity". Nous l'avions développé de sorte qu'ajouter une valeur flottante à une entité via ce "+=" permet de déplacer cette entité de la partie entière du nombre passé en paramètre en abscisse (x), et de sa partie décimale en ordonnée. Par exemple, pour une "Entity" nommée "e", écrire "e += 5.8;" revient à déplacer "e" de 5 pixels vers la droite, et de 8 vers le haut.

- Nous avions utilisé plusieurs QList en tant que conteneur template d'un type paramétrique, ainsi qu'un QVector. Nous n'avions pas fait particulièrement attention aux contraintes, leurs utilisations sont justifiées.

Procédure d’installation (bibliothèques...) et d’exécution du code

Installation et exécution Pour pouvoir jouer, il vous faut tout d'abord compiler le code source. Pour cela, deux moyens s’offrent à vous :

- Le premier, importer le projet dans Qt Creator (IDE). Run le projet, et vous vous retrouvez avec un jeu compilé !

- Le second est d’ouvrir un terminal, de vous déplacer dans le répertoire de votre projet, puis d’exécuter en tant que super utilisateur (“sudo“ ou “su“) : qmake Project_cpp.pro. Un Makefile sera ainsi généré. Libre à vous d’utiliser les différentes règles de compilations qui y seront définies, telles que “all“ ou “clean“.

Dans les deux cas, vous aurez besoin des paquets de Qt 6.2.2. Problème : Qt 6.2.2 ne fait pas encore partie du repository. Vous devrez donc imiter les développeurs sous Windows et télécharger Qt depuis le site officiel :

- Allez sur l'adresse suivante : https://www.qt.io/download.
- Sélectionnez "Download for open source users".
- Descendez jusqu'à pouvoir cliquer sur "Download the Qt Online Installer".
- Cliquez sur Download
- Ouvrez un terminal, puis déplacez-vous jusqu'à vos téléchargements (probablement "~/Téléchargements", ou "/tmp/VOTRE_NAVIGATEUR/", mais cela peut-être une destination toute autre), puis ajouter les droits d'exécutions à l'installer de qt avec la commande : chmod +x NOM_INSTALLER.
- Exécutez cet installer en tant que super user ("sudo" ou "su").
- Connectez-vous avec votre compte Qt. "Suivant"
- Cochez avoir lu et accepter les conditions de Qt, et être un particulier. "Suivant"
- "Suivant"
- Choisissez ou non d'envoyer des données statistiques. "Suivant"
- Sélectionnez où installer Qt, puis cochez "Custom installation". "Suivant"
- Faites dérouler le menu "Qt", puis sélectionnez tout "Qt 6.2.2". "Suivant"
- Cochez avoir lu et accepter les conditions de Qt. "Suivant"
- Appuyez sur "Suivant" jusqu'à ce que l'installer ait terminé. Libre à vous de vous référer aux eux méthodes de compilations évoquées plus tôt !

Les parties de l'implémentation dont vous êtes les plus fières

S'il nous fallait désigner les implémentations dont nous sommes les plus fiers, la réponse ne nous viendrait pas en un éclair. Il nous semble préférable d'énumérer simplement certaines parties plus techniques à nos yeux.

- Premièrement, l'utilisation d'un QTimer personnalisé ("Update") pour appeler à fréquence régulière une fonction faisant office de "while dans un main" résout tous nos problèmes liés à la nécessité de "temps réel" dans un plateformeur.

- Ensuite, il n'y a rien de « transcendant », mais la gestion des projectiles est intéressante bien que classique : ils ont tous un vecteur directionnel propre, et ont un effet à l'impact.

- Aussi, rendre fonctionnel la sortie audio du jeu (la musique) à partir d'un chemin relatif était extrêmement compliqué, et très mal documenté. Peu de gens en parle, la doc officielle de Qt fait peu d'efforts, et il m'a fallu tester beaucoup de choses pour tomber sur la solution.

- La gestion des collisions est aussi compliquée : nous avons tout d'abord utilisé ce que Qt mettait à notre disposition, mais ce système de collision graphique était coûteux et fonctionnait mal, c'est pourquoi 70% du code ne l'utilise pas et préfère un système plus mathématique. Avoir opté pour ce changement peut être considéré comme une sorte de fierté.

Finalement, nous sommes fiers de notre projet dans sa globalité. Il combine gestion des collisions, utilisation de slots et de signaux, dessins partiellement dessinés par nos soins (à l'exception des sprites de Trump, des tuyaux et du fond d'écran), développement événementiel et média sonore. Le tout est fonctionnel, et le jeu est plutôt marrant ! C'est difficile de parler de certaines implémentations quand le jeu forme un tout agréable !

Nous vous remercions pour votre lecture de ce compte-rendu, et espérons que vous passerez un agréable moment sur notre jeux-vidéo !