Rapport de projet C++

Sujet : THERE IS NO PLANET B

LAURENT Franck & ROSIER Bruce

Table des matières

1. Description du jeu

25 SECONDS TO SURVIVE est un remake du célèbre jeu **60 SECONDS TO SURVIVE** dans lequel vous incarnez le rôle d'un père de famille nommé JP. Le **19 mai 2060**, une alerte à la bombe nucléaire retentie et vous avez **25 secondes** pour **sauver votre famille** et **ramener avec vous le maximum d'objets** du quotidien utile pour survivre le plus longtemps possible dans votre bunker avant l'impact. Vous pourrez emmener avec vous jusqu'à 10 objets dont vos propres enfants et votre femme. Il faudra donc être stratégique pendant ce choix pour **survivre le plus longtemps possible enfermer**.

Se **réfugier dans le bunker n'est que la première étape**. C'est maintenant que la survie commence ! L'objectif est de survivre un maximum de jours dans le bunker. Il faudra donc **rationner les ressources**. L'eau et le cassoulet constitueront les besoins vitaux des personnages présents dans le bunker. Chaque jour, vous aurez le choix de rationner vos ressources et de donner à manger et/ou à boire aux membres de votre famille. Vous aurez également le choix **d'envoyer un membre de votre famille en expédition** à l'extérieur du bunker afin de tenter de **ramener des objets** qui pourront vous être utile pour survivre. Soyez stratégique, l'expédition ne sera pas la même suivant le membre de la famille qui part. Ce sera à vous d'explorer le panel des possibilités qui s'offrent à vous. **Chaque histoire sera différente**, ce sera à vous de décider **quel joueur vous voulez incarner**.

Tous ces choix s'effectueront dans le livre de survie qui se trouve en haut à gauche de votre écran. Faites attention, chaque choix est définitif, vous ne pourrez pas revenir en arrière.

2. Comment jouer :

Chaque nouveau jour, vous pouvez voir dans votre bunker les membres de votre famille ainsi que tous les objets que vous possédez. Il vous suffira ensuite de **cliquer sur le livre en haut à gauche** de votre écran pour avoir des précisions sur **l'état de santé de vos personnages**. Lorsque vous aurez tourné la page en cliquant sur la petite flèche en bas du livre, vous aurez la possibilité de **rationner vos ressources.** Il suffira donc de cliquer sur la bouteille d'eau située en dessous d'un personnage pour lui donner à boire, sur la boîte de cassoulet pour lui donner à manger et sur la trousse de soin pour le soigner si le personnage est tombé malade. Vous saurez si un personnage est malade lorsque sa couleur de peau vire au jaune. Vous aurez donc 7 jours pour le soigner avant qu'il ne meure. En ce qui concerne la quantité de nourriture et d'eau, prenez en compte qu'une bouteille d'eau et une boite de conserve permettent de faire boire et nourrir 4 personnages respectivement. Vous pouvez voir en temps réel la quantité d'eau et de nourriture qu'il vous reste.

Une fois le rationnement effectué, vous pouvez **décider d'envoyer quelqu'un en expédition** pour tenter de **rapporter des ressources** nécessaires à votre survie dans le bunker. Vous ne pouvez envoyer qu'un seul personnage à la fois avec un seul objet maximum. Lorsqu'un personnage est parti en expédition, il n'apparaît plus dans le bunker. De ce fait, vous ne pouvez pas savoir si ce personnage est mort pendant l'expédition. Petit Tips, la durée maximum d'une expédition est de 4 jours, retenez donc bien le jour auquel vous avez envoyé votre personnage en expédition. Vous pouvez également décider de n'envoyer personne en expédition mais vous ne pourrez donc récupérer aucun objet pour le jour suivant. Une fois votre sélection de faite, vous passez au jour suivant.

Le **jeu se termine lorsque tous les personnages** que vous avez sélectionnés en début de jeu **sont mort**. Vous pouvez voir toutes les actions qui se sont déroulées et le nombre de jours pendant lesquels vous avez survécu.

3. Conception du code :

Pour pouvoir réaliser ce jeu, nous avons utilisé une classe principale dans laquelle tout **l'affichage** du jeu s'effectue. On appelle cette classe **mainwindow**, elle hérite de la classe QMainWindow de Qt. C'est dans cette classe qu'on **affichera toutes les différentes scènes de jeu**. De la même façon, on utilise une autre classe principale **Histoire** qui contiendra toutes les **méthodes de déroulement de jeu** qui seront appelé par mainwindow. Pour rationner les ressources par exemple, la classe mainwindow sera chargée de récupérer les sélections du joueur (qui mange et qui boit). Et la classe mainwindow appellera la méthode rationnement de la classe Histoire avec les informations récupérées en paramètre. Pour pouvoir **récupérer les sélections du joueur**, plusieurs classes qui héritent de **QGraphicsPixmapItem** ont été nécessaires. On retrouve tout d'abord la classe **CustomPixmapItem** qui permet de récupérer la sélection du joueur en redéfinissant la méthode **mousePressEvent** pour que lorsqu'on appuie sur un objet, celui-ci se place dans l'inventaire. Puis on dispose de la classe **Besoin** qui récupère la volonté du joueur de nourrir, d'hydrater et/ou de soigner ses personnages en redéfinissant la même méthode pour que lorsqu'on appuie sur un objet celui-ci devient opaque ou non opaque en fonction de la sélection du joueur. En ce qui concerne le menu d'expédition, on retrouve la classe **Besoin_expedition** et **Perso_expédition** qui redéfinissent la même méthode mousePressEvent pour rendre opaque ou non opaque la sélection du joueur mais aussi pour pouvoir changer d'objet sélectionné dynamiquement. Par exemple si le joueur choisi le père et qu'il change d'avis et appuie sur le fils, le père sera désélectionné (c'est-à-dire rendu non opaque) et le fils sélectionné (c'est-à-dire opaque). Nous nous retrouvons finalement avec plus de 8 classes. Plus nous avons de classes, plus nous avons de possibilités de mise à jour dans nos codes sans avoir à repenser tout notre code.

Pour le jeu, nous avons décidé de créer **2 classes abstraites**. La première est appelée **Entity**, elle représente une **entité du jeu** avec une image. On trouve donc ensuite **les classes Objet et Personnage** qui **héritent de cette classe Entity**. La classe **Objet** est une classe qui représente un objet. Chaque objet a un **type** et une **quantité** qui lui est propre. On retrouve par exemple un objet de type NOURRITURE, ou encore un autre de type HACHE. La classe **Personnage** hérite aussi de la classe Entity et c'est également une **classe abstraite**. Dans le niveau de hiérarchie suivant, on retrouve les **classes Père, Mère, Fils et Fille**. L'utilité d'avoir une classe Personnage qui soit abstraite réside dans les **méthodes qui seront surchargés tous les types de personnages**. Avoir ici **3 niveaux de hiérarchie** est très **utile** car certaines des **méthodes définies dans la classe Personnage sont utilisées par tous les types de personnage**. On retrouve par exemple les méthodes manger, boire et soigner qui sont communes à tous les types de personnages.

Par ailleurs, on retrouve dans la classe Personnage nos **2 fonctions virtuelles**. La première **virtual void besoins_vitaux(int folie)** décrémente les besoins vitaux de chaque personnage. Cependant, suivant qu'il s'agisse d'un père, d'une mère, d'un fils ou d'une fille, cette perte n'est pas définie de la même façon. C'est pourquoi on utilise une fonction virtuelle. La seconde fonction virtuelle est **virtual std::vector<Objet*> partir(bool * retour, std::vector<Objet*> objet).** Cette fonction est utilisée pour le départ en expédition d'un personnage. Or comme expliqué dans la description du jeu, selon le type de personnage qui part, on ne veut pas que le résultat soit le même. C'est pourquoi on utilise encore une fois une fonction virtuelle.

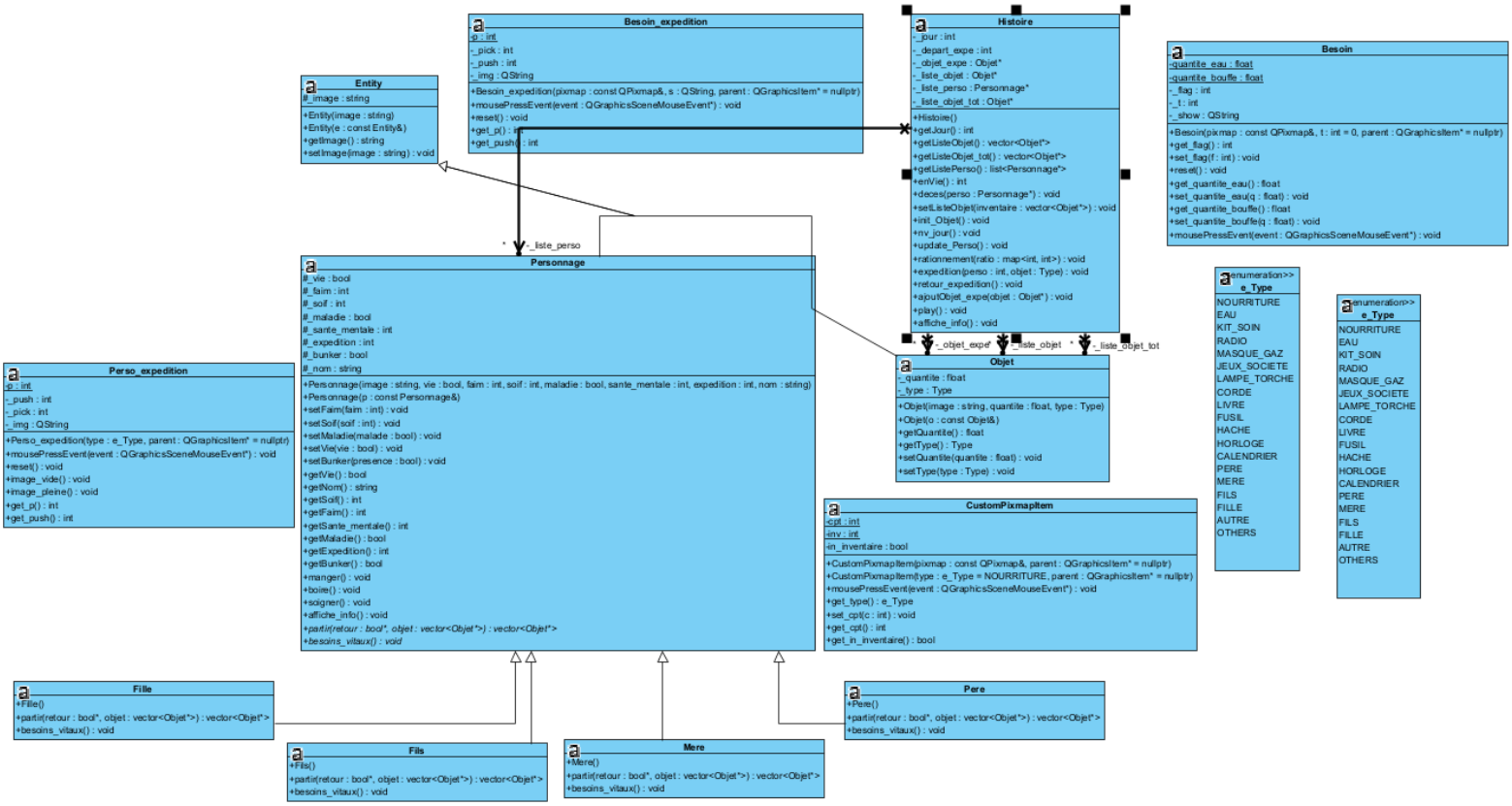
Comme la classe Histoire va **stocker tous les personnages et les objets** et mettre à jour dès qu'une action est réalisée sur l'un d'entre eux, on utilise des **vecteurs de pointeurs de Personnages et d'Objets**. On sauvegarde tous les Personnages en vie dans le vecteur de pointeur de Personnage **_liste_perso**. De ce fait, on pourra **parcourir facilement le vecteur** pour appliquer des méthodes à

certains personnages. Même chose pour la **liste d'objet** dans le bunker, on les stocke dans l'attribut **_liste_objet** de la classe Histoire. Comme autre conteneur, nous avons utilisé des **QList, des map** (etc.). L'utilisation des map est très utile notamment dans la méthode de la classe Histoire qui permet de remplir _liste_objet par rapport à la sélection de l'inventaire en tout début de jeu. On peut en effet **associer à un type d'objet, sa quantité**. Ainsi, il est facile de créer un Objet en parcourant ce map.

Enfin, la dernière contrainte qui nous était imposée était la **surcharge d'opérateur**. La surcharge d'opérateur permet de modifier le comportement d'un opérateur afin de réaliser des opérations intelligentes. Cela permet de rendre le code plus court et lisible. Dans notre cas, nous avons décidé de **surcharger l'opérateur ()** dans la classe Objet afin de **retourner le type de l'objet**. De ce fait, au lieu d'utiliser un getter, on utilise simplement l'opérateur () pour récupérer le type de l'objet. De la même manière, nous avons décidé de **surcharger l'opérateur « += » et « -= »** afin **d'augmenter et diminuer la quantité d'un objet** à la valeur que l'on souhaite. Cela facilite la lecture du code car ce dernier est plus court et concret.

4. Diagramme UML :

Vous trouverez ci-dessous le diagramme UML associé à l'application « **25 seconds to survive** ».



5. Procédure d'installation et d'exécution du code :

Pour pouvoir utiliser notre application, il faut utiliser qt5. Pour pouvoir installer qt5 sur votre Linux, il suffit d'entrer la commande : « **sudo apt-get install qt5-default** ». Nous vous conseillons d'utiliser g++ 9.3.0 comme compilateur car nous avons conçu notre code avec celui-ci. Il vous suffit ensuite de compiler votre code avec la commande : « **qmake projet_ui.pro** ». Vous remarquerez que le code

compile sans warnings. Enfin, comme tout executable, il suffit de lancer votre application avec la commande « **./projet_ui ».**

6. Fiertés :

Il y a certaines parties de notre implémentation dont nous sommes particulièrement fières. Nous sommes assez fiers des **détails au niveau de l'affichage**. Par exemple, lors de l'étape du rationnement, initialement chaque bouteille d'eau et chaque boite de conserve est peu colorée. Puis lorsque l'on clique dessus, celle-ci devient colorée puis se grise à nouveau si on reclique dessus. Même chose pour la sélection du personnage et de l'objet de l'expédition, nous avons réussi à implémenter le fait qu'un seul personnage peut être sélectionné et donc plus coloré que les autres. Ainsi lorsque l'on clique sur un autre personnage pour changer notre sélection, le personnage déjà sélectionné redevient grisé et le nouveau coloré. Nous sommes aussi très fier du rendu de notre jeu même si nous aurions voulu ajouter d'autre fonctionnalité tel qu'un scenario aléatoire dans lequel il serait possible de faire des échanges avec des personnages insolites.

7. Points à améliorer

En revanche, nous pensons que certaines choses auraient pu être mieux implémentées. Notamment au niveau du lien entre l'affichage et le déroulement de l'histoire. Nous avons conçu le code en dissociant entièrement l'affichage de l'histoire. Or ces deux entités sont intimement liées, et nous avons rencontrés de nombreux problèmes lors de l'association de ses 2 parties. Par ailleurs, en ce qui concerne les fuites mémoires, nous avons utilisé valgrind mais celui-ci nous indique de nombreuses erreurs que nous n'avons pas réussi à résoudre. D'autre part, une fois que le jeu se termine, l'écran de fin s'affiche mais il n'est pas possible de relancer une nouvelle partie. Nous avons quand même beaucoup appris sur la conception d'un petit jeu en C++ en utilisant Qt.