Mini-projet C++ 2023-2024

Nils Beaussé – CIR2 Nantes

1. Consigne et présentation du sujet

Ce sujet a pour but de vous présenter le mini-projet de C++/QT et de vous aider à vous lancer sur sa réalisation. Le but est de créer un petit jeu en 2D de type *« tower-defense »*, comme sur la figure ci-dessous : Le joueur y construira des tours dans le but de juguler une invasion.

Figure 1 : Les tower-defenses sont un type de jeu assez célèbre. Ci-dessus un tower defense très classique.

Méthode de travail :

- Chaque code est réalisé en binôme. Si du code est utilisé à plusieurs, merci de l'indiquer dans vos sources. Si du code provient de l'extérieur, merci d'en indiquer la source en commentaire dans le code. Dans le cas contraire, c'est du plagiat et les codes plagiés seront sanctionnés. La copie de bout de code sans les comprendre sera également sanctionnée (un test pourra être proposé sur certains codes au hasard ou sur tous).
- Le choix du binôme est libre, à vous de le déterminer, la note sera identique sauf si l'un des deux ne comprend pas certaines parties du projet, ce qui pourra être déterminé par une séance de question orale si l'enseignant à un doute.
- De fait : vous devez vous répartir le travail de manière efficace tout en comprenant ce que fait votre camarade.

Cahier des charges minimal :

- La carte doit être composée de chemin et de zone autres, la carte doit être parcourable avec les flèches du clavier
- Les chemins doivent mener vers une unique « base » qui peut être représentée par ce que vous voulez (un château par exemple, ou une simple extrémité de la carte)
- Des ennemis doivent apparaitre à l'extrémité des chemins et se diriger, en suivant les chemins, vers la base.
- Le joueur perd des points quand les ennemis arrivent à toucher la base
- Le joueur peut se défendre en construisant des tours sur les bords des chemins (pas sur les chemins) : on doit pouvoir les construire en les plaçant avec la souris
- Les tours tirent des projectiles enlevant des points de vies aux ennemis, quand un ennemi atteint 0 PV, il meurt.
- Le meilleur score et le pseudo de l'utilisateur l'ayant réalisé sont mémorisés et peuvent être affichés via un tableau des scores en jeu.
- On doit pouvoir recommencer une partie lorsque le joueur a perdu sans devoir relancer le programme
- Le choix de l'esthétique et des règles du jeu est libre

Liberté dans les règles et bonus :

- En dehors des consignes minimales ci-dessus, vous êtes libre d'ajouter ce que vous voulez au jeu, par exemple :
    - Système de pièces d'or, lâchées par les ennemis, à ramasser pour payer les nouvelles tours
    - Différents types d'ennemis
    - Différents types de tours
    - Différents types de projectiles (par exemple des projectiles types éclairs qui touchent à tous les coups, ou des projectiles type fragmentation qui envoie de nombreux projectiles sur une zone sans précision)
    - Système d'upgrade des tours
    - Différentes phases de jeux avec des ennemis de plus en plus fort
    - etc etc.
- De même, l'intégration de musique et de bruitage sera considérée comme un plus.
- Chaque caractéristique en plus par rapport à la consigne vous rapportera des bonus dans la note. Il faut savoir que faire le strict minimum bien vous rapportera entre 10 et 11 selon la réalisation et la qualité du rendu. Les notes entre 11 et 20 seront atteintes par ce que vous rajoutez en plus, donc soyez créatif !

Rendu du code :

- **Lundi 10 juin, 18h00**. Attention, tout retard sera sévèrement sanctionné par des points en moins.
- Une archive à déposer sur Moodle de votre code portant votre nom de famille en minuscules et sans espace contenant :
    - Les fichiers .h et .cpp
    - Le fichier de compilation (CMakLists.txt, Makefile, fichier .pro…)
    - Les répertoires contenant les différentes sources utiles au projet (images, son…)
    - Un fichier lisez-moi.txt expliquant ce qu'il faut faire pour compiler et lancer votre jeu et les éventuels prérequis ainsi que les règles, commande en jeu, particularité, etc.
    - Une attention particulière va être portée sur le rendu. Notamment le bon déroulement de la compilation de votre code source et de sa première exécution, ainsi que la présence de commentaires dans le code source.

Le code peut être écrit sur la plateforme et avec l'EDI de votre choix (CLion, Eclipse etc…).

2. Pas à pas pour commencer

a) Pour bien démarrer :

1. Récupérer l'archive CPP_QT_TPminiprojet.zip sur Moodle.
2. Créer un nouveau projet C++ sur Clion.
3. Copier/coller le code contenu dans l'archive dans votre projet. Attention l'organisation est un peu différente de ce que vous connaissez :
    - Un fichier CMakeLists.txt qui contient les instructions pour compiler le code source. Il contient notamment une instruction permettant de lister tous les fichiers .h et .cpp d'un répertoire.
    - Un répertoire ressources vide qui contient toutes les ressources nécessaires (images…).
    - Un répertoire src qui contient tout le code source.

Il n'est pas demandé de respecter stricto sensu cette organisation de répertoires/fichiers. Mais c'est un bon début d'organisation avant de passer à un découpage plus fin pour des projets plus conséquents.

4. Compilez et exécutez le code source. Prenez soin de vous approprier le code source, il représente la base de votre application !

b) Ajout d'objets à la scène

La scène représente la classe qui contient tous les éléments du jeu. Évidemment, plus le code est conséquent, plus cette classe doit être décomposée en sous-classes, chacune s'occupant de tâches particulières à la scène. Pour ce projet, vous pouvez rester concentrés sur cette classe et ne pas nécessairement vous préoccuper de créer d'autres classes.

Les objets ajoutés à la scène héritent tous d'une classe QGraphicsItem. Cette classe est la classe mère de classes filles spécialisées : QGraphicsTextItem, QGraphicsRectItem, QGraphicsPixmapItem…

Commencez par tester l'ajout d'un objet à votre scène : dans le constructeur de MyScene, ajouter :

```
QGraphicsRectItem* qgri = new QGraphicsRectItem(10, 100, 300, 200);
this->addItem(qgri);
```

Observer le résultat.

Ajouter maintenant :

```
QGraphicsTextItem* qgti = new QGraphicsTextItem("CIR2 Nantes");
this->addItem(qgti);
```

Vous aurez ensuite besoin d'ajouter des `QGraphicsPixmapItem` qui sont des objets contenant une image. Vous pouvez essayer.

c) Animation de la scène

Pour l'animation, il suffit d'ajouter un timer à la scène. Créez un attribut timer dans MyScene puis initialisez-le dans le constructeur :

```
timer = new QTimer(this);
connect(timer, SIGNAL(timeout()), this, SLOT(update()));
timer->start(30); //toutes les 30 millisecondes
```

Créer ensuite un slot dans votre classe qui est la méthode update(). Dans cette méthode, essayez de faire tout d'abord un affichage simple (avec un cout ou un qDebug), puis de bouger l'objet qgti défini précédemment en le faisant descendre vers l'objet qgri.

Pour cela, vous pouvez utiliser le code suivant dans la méthode update() :

```
QPointF pos = qgti->pos(); //récupération de la position de l'objet qgti
qgti->setPos(pos.rx(), pos.ry()+1); //incrémentation de la coordonnée y
```

d) Gestion de la collision

Pour pouvoir réaliser le jeu demandé, une question qui va vite se poser est de savoir quand un objet percute un autre objet ? Pour cela, Qt vous fournit un mécanisme de gestion des collisions sur les objets de type QGraphicsItem.

Dans la méthode update(), ajouter la gestion de la collision :

```
if (qgti->collidesWithItem(qgri)) {
   qDebug() << "Collision !";
}
```

Le message de collision doit s'afficher lorsque les 2 items entrent en collision. Attention, le calcul est réalisé sur les « bounding box » (boites englobantes) associées à chaque objet. Cette bounding box correspond à un rectangle qui englobe tout l'objet.

Les animations et les déplacements des items doivent se réaliser dans la méthode update() de la classe MyScene (ou dans une méthode appelée dans la méthode update). Cette méthode update() est appelée toutes les X millisecondes et permet le rafraîchissement du jeu. C'est ce qui va déterminer

notamment la vitesse de rafraîchissement (les FPS pour les gamers…). Il faut donc que les animations soient gérées dans cette méthode.

e) Gestion des touches clavier

Un dernier point à aborder est la gestion des touches au clavier. Dans la classe MyScene :

• définir dans le .h `void keyPressEvent(QKeyEvent* event);` et éventuellement `void keyReleaseEvent(QKeyEvent* event);` en protected (on va surcharger ces fonctions qui existent déjà dans la classe mère)

• les écrire dans le .cpp :

```
void MaClasse::keyPressEvent(QKeyEvent* event)
{
   if(event->key() == Qt::Key_P) { // appui sur la touche P du clavier
      …
   }
}
```

Inutile de faire un connect ou de gérer les signaux, les évènements claviers/souris sont automatiquement gérés dans la classe QGraphicsScene.

Pour mettre le jeu en pause, il suffit d'arrêter le timer.

f) Développement du jeu…

Vous avez maintenant tous les éléments en votre possession pour développer le jeu.

Concentrez-vous sur les tâches principales. Il est essentiel de savoir hiérarchiser les problèmes. Par exemple, si vous n'arrivez pas à mettre une image en fond, ce n'est pas essentiel au bon fonctionnement de votre programme. Vous reviendrez dessus plus tard, vous allez monter au fur et à mesure en compétences sur Qt, des choses vont s'éclaircir au fur et à mesure du temps passer à développer.

Ne mélangez pas tout, au risque de créer du « code spaghetti » : découpez votre code en méthodes, chaque méthode réalisant une tâche simple et claire.

Testez et débuggez votre code le plus souvent possible. Évitez de reproduire sans cesse les mêmes tests, au risque de passer à côté de certaines erreurs.

g) Aide : redimensionnement de la fenêtre et systèmes de coordonnées

Il existe plusieurs façons de faire (comme très souvent). Vous remarquerez qui si vous utilisez une petite image en fond de votre scène, elle va se répéter. Voici une solution « propre » avec en fond une image (png, jpg…).

La méthode consiste à :

• Réécrire la méthode drawBackground de votre classe héritant de QGraphicsScene pour dessiner une seule fois l'image dans la scène.

• Réécrire la méthode resizeEvent de la classe s'occupant de la vue (hérite de QGraphicsView) pour redéfinir l'affichage lors du redimensionnement.

Concrètement cela donne :

```
void MyScene::drawBackground(QPainter* painter, const QRectF &rect)
{
    Q_UNUSED(rect);
    painter->drawPixmap(QPointF(0,0), pixBackground, sceneRect());
    // pixBackgroud est un attribut de type QPixmap qui contient l'image de fond
}
```

Il faut ensuite redéfinir une classe MyView héritant de QGraphicsView et surcharger la méthode resizeEvent :

```
protected:
virtual void resizeEvent (QResizeEvent* event)
{
    this->fitInView(sceneRect());
}
```

Normalement, les redimensionnements de votre fenêtre principale donneront un comportement « normal ».

À partir du moment où vous faites cela, toutes les coordonnées utilisées vont dépendre des coordonnées de la scène : il faut donc tout développer en fonction de la taille de l'image que vous allez positionner au fond. N'hésitez pas à la redimensionner une bonne fois pour toutes.

Le reste est à faire par vous-même !

Bon courage à toutes et à tous !