RAPPORT MULTISTORM

14/02/2016

Projet C++

ADRIEN MARET / ARNAUD PASSELAIGUE
JULIEN CATTEAU / CLEMENT MABILEAU

Dépôt Github

Guide de lancement

Avant de démarrer le projet, veillez à installer les 4 paquets suivants :

sudo apt-get install clang qtmultimedia5-dev libaudio2 pulseaudio-utils

Il faut ensuite compiler le projet :

qmake -makefile && make

Puis lancer un serveur :

./multistorm srv

Et un client :

./multistorm

(La gestion d'un serveur et d'un client en même temps demande la dernière version de QT qui n'était pas sur les VMS, cependant le code est commenté dans le main)

Un de vos amis peut vous rejoindre si il est sur le même sous-réseau que vous, simplement en saisissant votre IP à la place de "localhost" à côté du bouton "Join Game". Le Wifi est à éviter et l'utilisation d'un switch ou d'une connexion ethernet cablé préférable.

Répartition des taches

Adrien : Gestion client/serveur, architecture général, gestion multijoueur

Clément, Julien, Arnaud : Gestion partagés des déplacements, des rotations des images et des formes, des hitboxs et collisions, des tirs et projectiles, ainsi que les explosions, le bouclié, l'alignement des éléments entre eux...

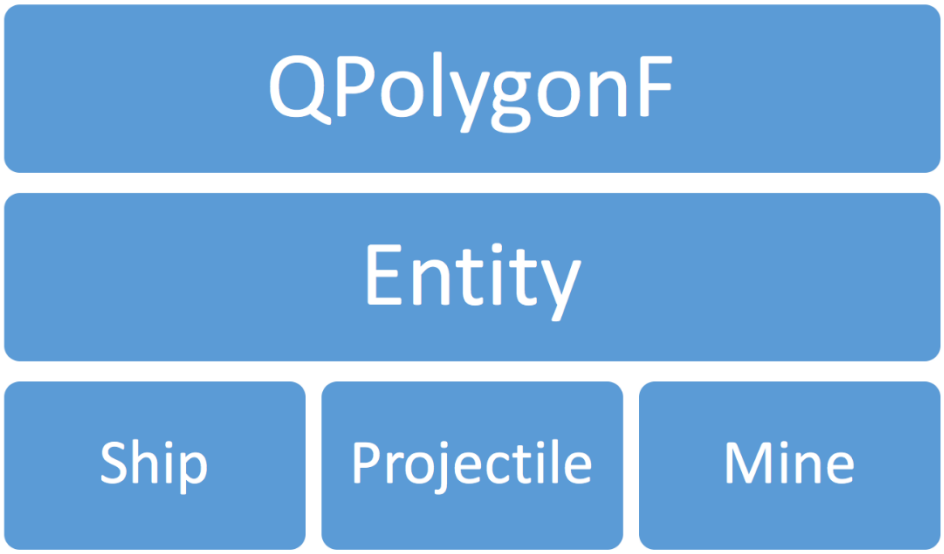
Deux branches principales de développement ont étés utilisés sur Git, pour 6 branches au total depuis le début du développement.

La répartition des tâches et des tickets à été simplifié à l'aide de Trello

https://trello.com/b/XmMAga4R/minestorm

Entities

Toutes les entités du jeu héritent de la même super classe : Entity qui hérite elle même de QPolygonF (http://doc.qt.io/qt-4.8/qpolygonf.html)



Il existe 3 types d'entités : Ship, Mine & Projectile. Elles sont différenciées pas leurs Type.

Entity

La classe « Entity » comprend toutes les variables, méthodes et enum commun aux entités du minestorm.

Lorsqu'une entité meurt(collision d'un tir avec une mine, tir sortant de l'écran, un joueur déconnecté ou perd toute ses vies), on son état à Dead. Un parcours sur toutes les entités est fait entre deux itération de la fonction principale Core::Step(), afin de les supprimer hors de la boucle.

Elle comprend les méthodes suivantes :

- Rotation
- Mouvement
- Gestion de la vitesse de déplacement

Ship

La classe « Ship » comprends les méthodes, variables et enum du vaisseau d'un joueur.

Elle comprend les méthodes suivantes :

- Ajout/Suppression de la vie
- Ajout/Suppression du bouclier
- Calcul du centre
- Initialisation du PolygonF

Projectile

La classe « Projectile » comprends les méthodes et variables d'un missile.

Elle comprends les méthodes suivantes :

- Calcul du centre
- Jouer du son
- Initialisation du PolygonF

Mine

La classe « Mine » comprends les méthodes, variables et enum d'une mine.

Elle comprends les méthodes suivantes :

- Activer/Desactiver

Player

La classe « Player » contient toutes les méthodes et variables d'un joueur.

Un « Player » est construit avec un :

- Id
- Number
- Point du spawn

Un « Player » peut aussi avoir un pseudo. A saisir sur l'interface graphique.

Images

Les images du jeu sont chargées au début du jeu. Elles sont stockées, par type, dans un QVector<QSharedPointer<QImage>> dans la classe « Image ».

Pour accéder à une image il suffit d'appeler la méthode getImage(Element::Type, qreal) de la classe « Image » et de lui envoyer le bon Element ::Type ainsi qu'un angle.

MainWindow

La classe « MainWindow » crée un widget contenant un controller et un gameboard pour le jeu.

Elle set le BackGround du jeu. Le titre de la fenêtre et la taille de la police d'écriture.

Element

Element est une entité mais côté Client.

Un Element est construit par la classe « MessageObject » puis ajouter dans une liste QSharedPointer<QVector<Element>> &elements.

La classe « Element » contient les variables suivantes :

- Type _type
- QPolygon _polygon
- qreal _angle
- QPoint _center
- QPoint _imageCenter
- bool _shield
- bool _playSound

Un élement implémente IDrawable afin d'avoir une méthode draw() que l'on pourra utiliser dans le Display.

Collision

Toutes les collisions du multistorm sont gérées via la classe collision.

A chaque nouvelle step du jeu avant de faire bouger une mine ou un vaisseau on test si il y a collisions avec une autre entités. On appelle ensuite une fonction qui s'occupe de supprimer les entités mortes.

Déroulement du test de collision sur un vaisseau :

- On boucle sur toutes les entités (mines, tir, vaisseau).
- S'il y a collision on enlève une vie au vaisseau (sauf exceptions) puis en fonction de l'entité rencontré on effectue une action spécifique:
    - Avec un tir : On ajoute des points au tireur et on change l'état du tir sur "DEAD".
    - Avec une mine : On change l'état de la mine sur "DEAD".
    - Avec un vaisseau : Rien de plus.
- Chaque fois que le vaisseau perd une vie il est réinitialisé à ca position de départ et devient invincible pendant 4 secondes.

Déroulement du test de collision sur une mine:

- On boucle sur toutes les entités "tirs"
- S'il y a collision on change l'état de la mine et du tir à "DEAD". Puis on ajoute des points au tireur.

Fonctions

- CheckCollision(Entity, Entity) : Détecte une collision entre deux entités.
- DetectShipCollision(Ship) : Itère sur une liste d'entités et appelle CheckCollision(Ship, Entité).
- DetectMineCollision(Mine) : Itère sur une liste de tir et appelle CheckCollision(Mine, Tir).

MessageObject

Message object deserialize les messages reçu par la classe « MessageBase » .

Elle reçois un « String » émis par le serveur et crée les entités du jeu côté client.

Réseau

Le jeux se base sur une architecture Client / Server afin de gérer le mode multijoueur.

Il faut d'abord lancer un serveur qui gérera l'ensemble du jeux (clients, déplacement, collisions..) puis les joueurs se connectent dessus avec un client.

Le serveur s'occupe de chaque client dans un thread séparé matérialisé par la classe Worker. Ce dernier encapsule une BaseSocket qui contient les fonctions de base pour envoyer et recevoir des messages.

Le serveur possède 2 méthodes pour envoyer des message, broadcast pour envoyer à tous les clients et unicast pour envoyer à un client précis identifié par son identifiant (= socket descriptor). Il possède également un slot pour recevoir les messages en provenance des clients.

Le client lui contient une BaseSocket et utilise ses fonctions pour communiquer avec le serveur.

Les messages echangés entre les clients et le serveur sont sous forme de texte. Ils contiennent des informations sur les éléments à afficher ou encore les informations sur les joueurs. Ces messages héritent de MessageBase et possèdent 2 constructeurs : un pour sérialiser les données et un autre qui les dé sérialises.

Core

Cette classe "est" le jeu en lui-même. Elle contient un serveur pour communiquer avec les clients, la liste des joueurs actuellement connectés et la liste de toutes les entités présentes.

Au lancement, le serveur commence à écouter sur le port 4242 et attend que le 1er joueur se connecte avant de lancer une partie.

Lors de la connection d'un joueur, si il reste de la place, il est ajouté à la liste des joueurs avec ses informations (pseudo, numéro, vaisseau) puis on ajoute son vaisseau à la liste des entités. Si il ne reste plus de place, le client est placé en spectateur, il pourra regarder la partie mais pas participer.

Si un joueur perd ou se déconnecte, une place est libérée pour le prochain client à se connecter.

Un QTimer règle le nombre de cycles par seconde du jeux (25 par défaut), chaque 40ms la fonction step() est appellé afin de faire avancer le jeu (Déplacements, collisions, envoi des éléments aux clients).

Cela nous permet également de nous baser sur cette fonction pour créer des timer pour d'autres objets. En créant un simple compteur dans la classe Mine que l'on décrémente à chaque step(), il est possible de les faire éclore à un nombre de seconde précis.

Le Core a également la gestion de la redistribution des événements envoyés depuis les clients. (Touches pressés, envoi du pseudo..) Les événements arrivent dans la fonction messageDispatcher() avec un idClient (=socket descriptor) et le message en texte. Ils sont ensuite désérialisés pour récupérer les informations puis transmis au joueur adéquat afin d'exécuter une action. (Tirer, accélérer..)

Display

La classe Display s'occupe de l'affichage des éléments transmis par le serveur et de l'envoi des évènements au serveur.

Elle possède un Client qui se connecte au serveur afin de pouvoir communiquer, la classe Images pour gérer les différentes images, d'une liste de PlayersInfos pour garder en mémoire les informations sur les joueurs (pseudo, score, vies) et un compteur de FPS.

Les évents sont reçu depuis la classe GameBoard qui représente l'air de jeu. Ils sont ensuite sérialisés et transmis au serveur.

A chaque fois que le serveur transmet la liste des éléments au Display, celui-ci les map au sein d'une liste d'élément, puis itères sur ces derniers dans un switch dessinant les différents éléments graphiques et images selon le type de l'élémént reçu, son angle, sa taille et sa position. D'autre informations sont également passé comme le fait de jouer un son, ou la présence d'un bouclié sur les différents vaisseaux.
Le nombre d'images par secondes est donc dépendant du nombre de message reçu par seconde.