Projet Final Mystic Quest

Abdallah OUSAID – Sandoche BARRES

Figure 1 : Illustration du jeu Final Fantasy (map)

Figure 2 : Illustration du jeu Final Fantasy (Combat)

Objectif

1. Présentation générale

L'objectif de ce projet est de réaliser un clone du célèbre jeu de rôle Final Fantasy avec les commandes et les règles de bases. Un exemple est présenté sur les figures 1 et 2. Le but du jeu est de récupérer 4 cristaux que protègent les boss dans chaque donjon.

2. Règles du jeu

Final Fantasy est un RPG (Rôle Playing Game) où l'on contrôle un personnage qui navigue sur une map en 2D. On peut naviguer dans trois types d'environnements qui sont : la map du monde, les villes et les donjons. Dans la map du monde et les donjons on rencontre des ennemis qui sont visibles sur la carte ce qui nous donne une liberté d'engager ou non le combat. Dans les villes on récupère principalement des objets, des équipements et de la vie. Dès qu'on rentre dans une phase de combat, on change de plan visuel. L'équipe du joueur se déploie en une équipe de deux ou trois personnages contrôlés par ce dernier. Ce jeu se déroule en deux phases :

Phase de combat :

Chaque personnage va effectuer une action à tours de rôles (attaquer, magie, se soigner, etc). Dès que tous les personnages ont fini de prendre leurs décisions et qu'elles seront validées par le joueur, elles vont s'effectuer à tour de rôle ainsi que celles de l'ennemi pris par l'intelligence artificielle. A chaque combat gagné, l'équipe gagne en expériences et son butin. Enfin on sort de la phase de combats après dès que l'on a détruit tous les monstres. Les personnages et les ennemies ont une vie que l'on nomme HP (Health Power) caractérisé par un numéro entre 0 et 9999 selon le niveau des personnages et ennemies. Quand le personnage a 0 de HP la partie se termine.
En plus de la vie, chaque personnage et ennemies possèdent un niveau d'expérience et des statistiques qui leurs sont propres. Après chaque combat gagné, l'expérience permet d'augmenter ce niveau et ces statistiques alors que ceux des ennemies sont déjà fixés.

Phase de navigation :

Le joueur se balade dans la carte du monde et choisit soit de rentrer dans une ville, soit de rentrer dans un donjon ou soit de rentrer dans une serre de combat pour augmenter ses niveaux et gagner de l'expérience et des objets.

Univers du jeu : Le jeu se déroule dans un univers médiéval-fantastique composé de donjon et de village.

Hero et compagnon : On incarne un chevalier qui manie plusieurs types d'arme et qui a les statistiques initiales les plus élevées. Il rencontrer au cours de sa quête des compagnons qui ne peuvent manipuler qu'un seul type d'arme. Ils lui prêteront main forte dans sa quête des cristaux.

Ennemis et Boss : Les ennemis sont un ensemble de monstre mystique qui ont les mêmes attributs que les personnages du jeu sauf qu'ils sont commandés par l'intelligence artificielle. Le boss garde chaque donjon et détient un cristal.

Description et conception des états

2.1 Description des états :

Notre état du jeu est constitué de deux univers : nous avons le monde et le plan de combat. Le monde se compose d'un ensemble d'élément fixe (des donjons, les villes et les serres de combats).

Les donjons : sont les éléments principaux du jeu car ils contiennent les cristaux nécessaire pour le finir. Le donjon est constitué d'éléments fixes (coffres d'objets, ennemis/boss et le labyrinthe) et d'éléments mobiles (joueur).

Villes : sont les éléments du jeu ou l'on peut régénérer de la vie, acheter des objets et des équipements. Ils constituent comme les donjons des petits mondes en elles-mêmes.

Serres de combats : c'est un passage direct de la map du monde à la phase de combats sans que cela influence l'évolution du jeu. Ce lieu permet de faire gagner de l'expérience et des objets grâce aux combats menés avec le libre arbitre du joueur

Chacun de ces trois éléments possèdent :

- Coordonnées (x ; y) dans la map.
- Un identifiant de type d'éléments : ce nombre indique la nature de l'élément.

2.2 Eléments des états :

Chaque lieu du jeu (monde, donjons, villes et serres de combats) sont des mini monde en eux-mêmes. Ils se composent d'éléments fixes qui sont :

Les ennemis: se trouvent seulement dans les donjons et serres de combats sous la forme d'un GIF animé.

Les coffres d'objets: se trouvent un peu partout dans les villes et les donjons. Ces coffres peuvent contenir des équipements, fioles de vie, fioles de magies et des objets.

Les obstacles: il s'agit des éléments infranchissables par le joueur. Il peut s'agir par exemple des arbres, montagnes, murs.

Les murs: ce sont les éléments qui délimitent les espaces du jeu (donjons, villes et carte de monde) et qui permettent de contenir tous les éléments du jeu dans un espace restreint.

Les espaces de départs: qui définissent les positions initiaux pour le joueur dans différents lieux du jeu.

Éléments mobiles:

L'élément mobile du jeu, qui correspond uniquement au personnage, possède une direction (aucune, gauche, droite, haut ou bas), une position et une clef de localisation qui permet de définir la localisation géographique du personnage.

L'élément mobile « personnage » est dirigé par le joueur et celui-ci commande la propriété de direction de ce personnage. Il possède deux status :

- Mode navigateur : où le personnage se déplace dans la carte ou les autres lieux de navigation.
- Mode combat : où le personnage peut déployer ses armes et compétences.
- Mort : lorsque le personnage a 0 de point de vie.

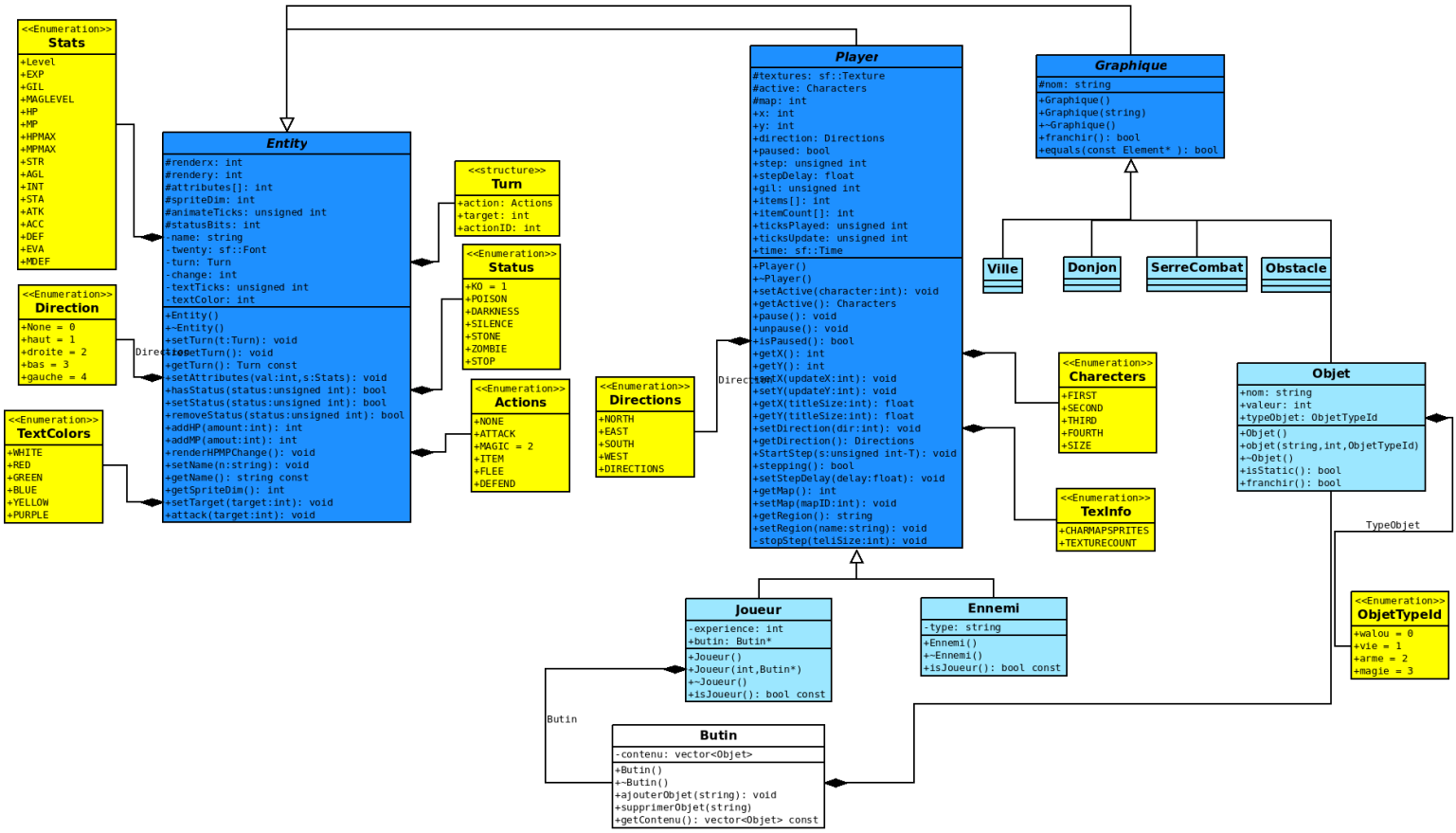
État général:

Le personnage possède un niveau d'expérience qui évolue selon le nombre de combat gagné et récupère des cristaux à chaque combat gagné contre le boss qui se trouve à la fin d'un donjon. Dès que le joueur a récupéré quatre cristaux le jeu se termine.

1. Conception logiciel :

Diagramme UML

Le diagramme UML suivant résume l'organisation des classes de notre jeu. Cela va nous permettre de modéliser les différentes classes et donc nous faciliter la conception et détecter facilement les éventuels bugs. Cette réalisation risque de changer lors de la détection de bugs en phase de réalisation.

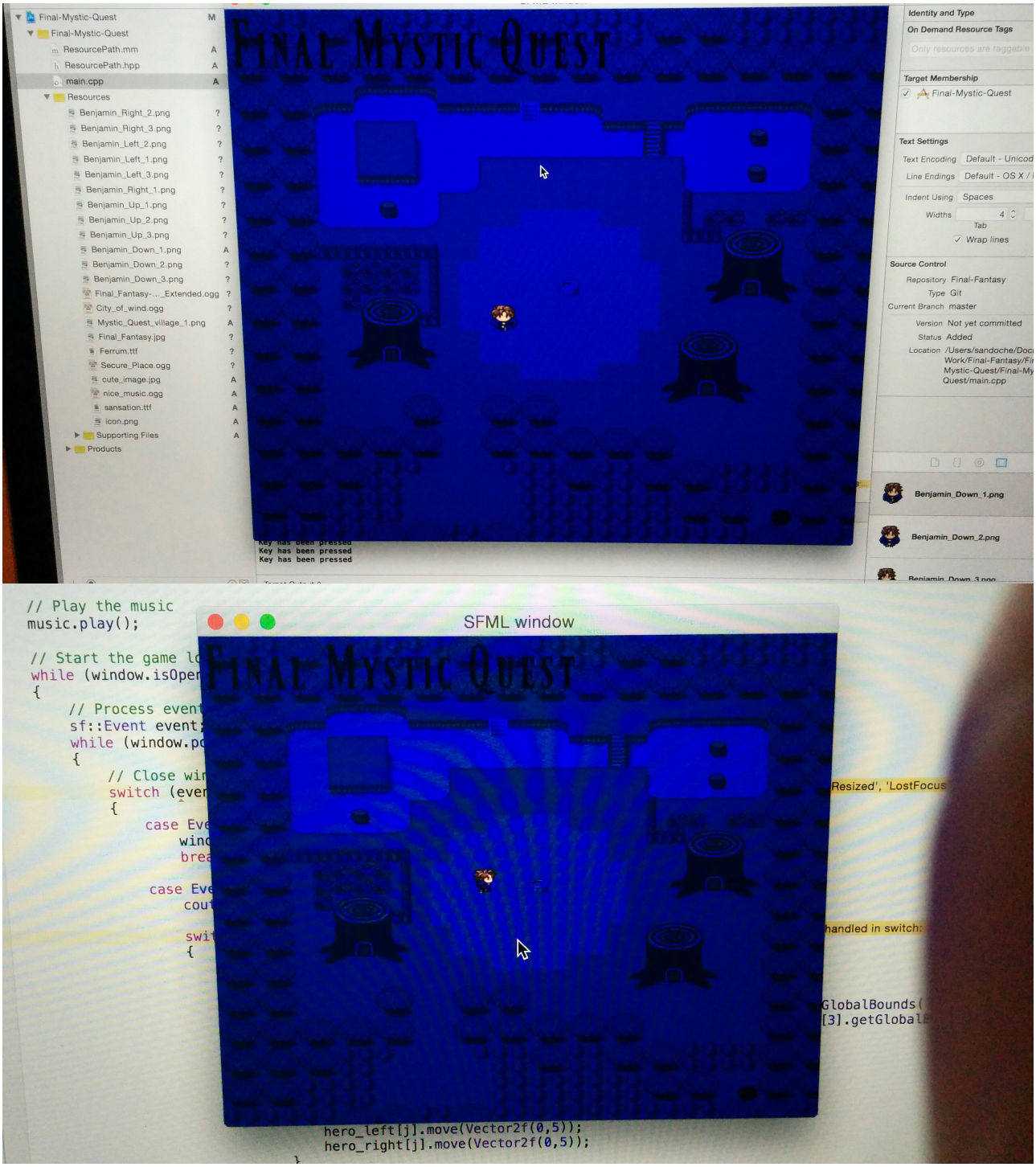


4. Rendu : Stratégie et Conception :

4.1 Stratégie de rendu d'un état

Pour le rendu d'un état, nous n'avons pas choisi de stratégie particulière. Comme dis plus haut, le jeu se décompose en deux phases : une phase exploration et une phase de combat. Pour le rendu d'un état, nous avons choisi d'afficher pour le moment le jeu dans sa phase d'exploration. Pour cela, nous avons affiché un village dans sa globalité avec le sprite du héros qui se déplace dans le village. Nous avons donc mis en premier plan le sprite du village et celui du héros. Aucun élément de collision n'a été mis en évidence. Par contre les différents états de déplacements du héros ont été mis en place. Pour cela nous avons chargé chaque états du héros dans une texture différente et ensuite afficher les sprites. Les touches directionnelles du clavier servent à afficher ces différents états. Par exemple en appuyant la touche directionnelle gauche, le héros regarde vers la gauche. Pour permettre une détection en temps réelle des différents changements d'états du personnage, une structure avec des switch est utilisée. La méthode utilisée pour afficher un tel changement d'état n'est pas du tout optimisé et elle est même assez laborieuse car une fois qu'un autre event intervient, le sprite disparaît. Il faut donc trouver une autre méthode beaucoup plus efficace pour permettre un affichage en continu. Mais le plus dure à été fait car c'est essentiellement le sprite du personnage que l'on va manipuler dans la suite.

Voici le sprite du village avec le héros qui se déplace:



5. Règles de changement d'états et moteur de jeu :

Pour le moteur de jeu, son rôle est de gérer en toute abstraction des autres classes. En effet, il permet de réunir toutes les règles et les fonctionnalités du jeu indépendamment du rendu qui peut changer de technologie (de SFML à OpenGL par exemple) ou des commandes qui peuvent venir d'une intelligence artificielle ou d'un client Android ou autre. Pour cela, on a réalisé lors du dernier jalon les classes qui permettent d'avoir un rendu satisfaisant. Donc la réalisation de ce jalon est d'autant plus simple car le moteur de rendu est déjà mis en place donc il faut juste créer les classes qui vont représenter le moteur de jeu. En effet, on a amélioré le rendu en utilisant un tableau de vertex. Cette solution permet de diminuer l'utilisation du CPU et GPU car l'utilisation de plusieurs textures sur un même rendu utilise beaucoup de capacité CPU pour pas grand-chose. En plus de cela, cette solution permet d'améliorer le rendu car les events de la souris influence les events du clavier chose non souhaité pour avoir un meilleur affichage et sans bugs. On arrive donc à différencier les deux events et avoir pour le cout un meilleur rendu.

L'optimisation du moteur de jeu est encore sujet d'amélioration car l'abstraction des différents éléments du moteur ne sont pas encore au point. Car cela va nous permettre d'améliorer les fonctionnalités du moteur sans toucher à toutes les structures du jeu. L'abstraction va nous permettre donc de faire la maintenance et l'amélioration avec une grande facilité mais aussi efficace.

6. Intelligence Artificielle:

L'intelligence artificielle du jeu n'intervient que durant la phase de combat. Comme dit plus haut, dans le jeu Final Fantasy lorsque l'on se déplace dans la map de la phase de navigation, le personnage rencontre aléatoirement des enemies avec une fréquence que l'on doit choisir. Dès que l'on rencontre cet ennemie, le jeu bascule dans un autre état. Durant cet état, le joueur doit prendre des décisions parmi "Attack, Spell, Item et Flee", doit choisir une cible parmi les enemies présents. Dès que chaque personnage a pris des décisions, une phase d'action se déroule. Durant cette phase, l'ennemie contrôlé par l'intelligence artificielle doit aussi prendre des décisions parmi "Attack, Spell, Item et Flee" et choisi aléatoirement le personnage à visé. Il s'agit d'une intelligence artificielle basique où il suffira tout simplement d'implémenter la phase d'actions de notre personnage de manière aléatoirement. Les seules différences qui existera parmi les différents ennemis du jeu résidera dans les attributs et la liste des attaques disponibles. Par exemple un boss du jeu pourra exécuter des magies beaucoup plus puissante qu'un ennemi régulier et pourra faire plus de dégâts lors d'une attaque. Il reste donc à implémenter la phase combat du personnage et une faire une version automatique pour l'ennemie.

6.1 Intelligence minimale

Après avoir réfléchie à ce que pouvais être une intelligence artificielle simple, nous avons dans un premier temps ignoré la phase de combat pour des soucis de temps. Pour réaliser un prototype de cette IA simple, nous avons créer un deuxième "**Player"** pour pouvoir faire des test de collisions. En effet pour réaliser une telle intelligence artificielle, il faut tout d'abord pouvoir gérer un minimum de collision. Et dans notre cas pour le réaliser, il s'agit essentiellement de mettre en place une collision avec un "**TileMap**". Nous avons donc implementer certaines méthodes pour la class **Map** qui sont certes, pour le moment, assez peu optimisé mais elles sont suffisamment claires pour pouvoir faire des tests avec. Nous avons implémentée une première méthode appelée "**int Map::blockedTileID(int x)"**. Cette méthode permet de renvoyer le numéro de la grille d'image. En effet sur le un "TileMap" nous avons chaque image à une certaine position. Cette méthode permet de renvoyer la position de l'image dans la grille.
Et l'autre méthode importante que nous avons implémentée est "**int Map::TileNumberConversion(int x,int y)** " qui permet de donner le numéro de la grille d'image en fonction des coordonnées du personnage. Le but de ces deux méthodes est de pouvoir détecter les collisions avec les "murs" et de ne pas passer "à travers". A partir de là, il ne nous reste qu'à réaliser une sorte de personnage automatisé qui avance tous seul dans la **Map** et qui lorsqu'il détecte un obstacle, choisit une des directions qui lui sont proposées. Pour le moment nous avons réussi à mettre à faire avancer le deuxième **Player** lorsque le premier rentre en contacte de celui-ci. Avec ce début prometteur, il nous sera plus facile de réaliser une véritable intelligence artificielle.
Cependant un problème subsiste toujours. Lorsque le personnage rentre en collision avec une des images de la grille souhaitée, le personnage n'arrive plus à bouger. Il suffira de faire un simple petit changement dans l'une des méthodes citées ci-dessus pour pouvoir y remédier.

7. Modularisation:

7.1 Répartition sur différents threads

Notre objectif dans cette partie est de placer le moteur de jeu sur un thread, puis le moteur de rendu sur un autre thread. Mais bien évidemment, nous n'avons pas réussi à obtenir exactement ce qui était demandée. Mais notre solution est très proche de ce qui est demandée. Nous avons bel et bien réussi à séparer les différents parties du code sur chaque thread mais on a utilisé un thread supplémentaire. En effet, notre moteur de jeu à été divisé en deux partie: une sur les collisions et l'intelligence artificielle et l'autre sur le mouvement du personnage principale. Normalement, nous étions supposé un seul et même bloc mais comme nous avons eu du mal à optimiser l'intelligence artificielle et les collisions, cette partie demeure non optimisée. Même si nous avons séparé chaque module en thread, nous l'avons pas fait de façon très optimisé. Pour résumé, nous avons trois threads:

- **moteur_jeu_1** qui est le thread s'occupant d'exécuter la méthode **"void character_movement(void)"**.
- **moteur_jeu_2** qui est le thread s'occupant d'exécuter la méthode **"void collision_render(void)"**.
- **rendu** qui est le thread s'occupant d'exécuter la méthode "void render(void)".

La méthode **"void character_movement(void)"**est celle qui s'occupe d'une partie du moteur du jeu (celle s'occupant du mouvement du personnage principal), la méthode **"void collision_render(void)"** celle s'occupant de l'autre partie du moteur du jeu (celle s'occupant de l'intelligence artificielle et de la collision) et finalement la méthode **"void render(void)"** qui s'occupe simplement du moteur de rendu.

La façon dont nous avons utilisé ces différents threads n'est pas du tout optimisée comme dit plus haut car lors du lancement de notre jeu, la mémoire consommé est très élevé (elle atteint une consommation de 1GO sur mon Mac..). Ce problème vient sûrement du fait que nous avons utilisé les threads à l'intérieur de deux boucle "**while"** dans la fonction main. Il faut donc pour la prochaine fois trouver un moyen d'uniformiser le moteur de jeu en une seule partie puis ensuite d'éviter cette surconsommation de mémoire.

7.2 Modularisation, API Web

Dans cette partie, on s'intéresse à la modularisation au niveau machine. Après avoir répartie les différentes méthodes sur différents threads, on souhaite les répartir sur différentes parties. Ce faisant on pourra séparer le moteur du jeu sur un serveur et le moteurs graphiques sur la partie client et ainsi avoir un fonctionnement totalement indépendant entre ces deux parties du jeu. Le fait d'avoir un programme multi-threadés va nous faciliter la tâche pour la mise en place d'une communication réseau. En effet pendant que les données de jeu soient reçues sur un thread, on peut continuer à mettre à jour et afficher l'état du jeu ce qui un gain niveau performance.

Pour mettre en oeuvre cette partie réseau, nous allons avoir besoin de 2 projets distincts:

- Un projet "**serveur**": pour créer le programme qui va faire tourner en boucle le moteur de jeu
- Un projet "**client**": pour créer le programme qui va se charger du moteur graphique

Dans la vision de notre projet, la partie serveur va contenir tout le code nécessaire pour pouvoir autoriser la connexion/déconnexion d'un client (l'IP, les ports nécessaires…) sur le réseau, la liste des clients connectés au réseau, les méthodes permettant de faire transiter des donnés du jeu entre lui-même et le client et surtout le thread contenant le moteur de jeu. Dans notre cas, il s'agira de **void character_movement(void), void collision_render(void)**. Dans la partie client, on va naturellement définir toutes les méthodes qui vont permettre à celui-ci de se connecter chez le serveur, de lui envoyer des demandes de donnés, et surtout le thread gérant le moteur graphique. Dans notre cas il s'agira de **void render(void)**.

Notre idée de base était de faire intervenir le protocole **TCP**. Mais ce protocole, en dépit des contrôles sécurisés (les paquets envoyés arrivent à destination dans le bon ordre), est beaucoup trop lent en terme d'envoie de donnés. Pour notre jeu, ce protocole n'est pas adapté car cela entrainera trop de latence au niveau affichage et réponse des commandes. On va dans un premier temps privilégier le protocole **UDP** qui est beaucoup plus rapide même si il y un effort à faire niveau des contrôles des paquets envoyés. Mais ce choix est temporaire. En fonction du temps restant pour obtenir un résultat, on pourrait décider d'utiliser un autre protocole. Ce choix constitue une base pour notre travail qui évoluera éventuellement.

7.3 Client/Serveur

Dans cette partie nous nous intéressons plus précisément sur le code de la modularisation niveau machine. Précédemment, nous souhaitions mettre en place un réseau client/serveur dans notre projet. Par manque de temps, nous avons pas pu réaliser ce que nous souhaitions initialement mais nous avons mis en place un simple mécanisme de base client/serveur qui représente la base réelle de notre travail. Ici on s'est contenté d'utiliser le protocole **TCP** pour mettre en place ce mécanisme. Au début on voulait mettre en place le protocole **UDP** mais au fur et à mesure de notre travail, on s'était rendu compte que pour un jeu en tour par tour comme le notre, on pouvait se contenter de **TCP** (sans oublier que le protocole **UDP** est plus difficile à mettre en oeuvre).

Notre mécanisme consiste à dessiner 2 rectangles, l'un de couleur bleu et l'autre de couleur rouge. Le rectangle rouge correspond au serveur et le bleu au client. Nous avons mis en place un ensemble de commande qui permet de déplacer les deux rectangles. Nous lançons le client et le serveur sur deux fenêtres différentes. Lorsque nous déplaçons l'un des carrés, le déplacement s'effectue également sur l'autre écran comme si les deux écrans étaient liés. Cela est du grâce à certaine ligne de code bien précise. Précisons également que nous avons utilisé la bibliothèque **Network** de **SFML** pour réaliser notre mécanisme.

Les parties de code qui différencient la partie serveur du client sont les suivantes:

```
if(connectionType=='s') **********Partie serveur**********
{
  sf::TcpListener listener;
  listener.listen(2000);
  listener.accept(socket);
}


else **********Partie client**********
{
 socket.connect(ip,2000);
}
```

On note ici que l'on utilise uniquement le port **2000** car on teste le programme sur le même poste. Ensuite la partie du code qui permet ce mouvement simultané sur les deux fenêtres est:

```
Packet packet;
if(prevPosition !=rect1.getPosition())
{
  packet << rect1.getPosition().x << rect1.getPosition().y;
  socket.send(packet);
 }

 socket.receive(packet);
```

```
if(packet >> p2Positon.x >> p2Positon.y)
{
  rect2.setPosition(p2Positon);
 }
```

C'est l'objet de type **Packet** qui permet le transport de donnée entre le client et le serveur.

Dans cet exemple, nous avons ainsi pu établir une véritable connection entre le client et le serveur. Même si il est basique, il nous a permit de visualiser en direct ce concept pas toujours évident à saisir et à se représenter. Le but réelle de ce mécanisme aurait été de transporter les données d'un thread pour pouvoir l'exécuter chez le client ou le serveur. On aurait ainsi pu réaliser ce que nous souhaitions obtenir à la fin. Mais là temps ne nous le permet pas. Peut être que dans un futur projet, nous pourrons l'implémenter…….

Au final nous ne sommes pas arrivé à boucler le projet dans sa globalité et nous sommes évidement très loin du résultat escompté mais un ensemble de point positif en ressort. Notons tout d'abord que les principaux mécanismes demandés fonctionnent. Nous avons bien un personnage qui se déplace sur une map avec des éléments de collisions. Les divers fonctions ont très bien été séparés dans le projet ce qui nous a permit d'obtenir un travail séparé en module. Même si chaque parties n'étaient pas aboutie, le résultat essentiels apparaissaient à cause fois. Si le temps nous l'aurait permis, on aurait évidemment pu optimiser certaine partie comme la partie sur les threads qui consomment encore une fois trop de mémoire (probablement du à des fuites de mémoires…). Nous avons néanmoins la base solide du projet final.