# COMPTE-RENDU TP3

## HMIN317 – Moteur de jeux


### Résumé

Création d'un système client/serveur avec changement de saison.



David Lonni

Master 2 IMAGINA

# 1) Fonctionnalités

- Création d'un serveur recevant les requêtes entrantes sur le port 9999
- Les 4 fenêtres des terrains sont des clients et contiennent un attribut de type QTcpSocket leur permettant de se connecter au serveur lors de leur création.
- Le serveur envoi toutes les 10 secondes le changement de saison aux fenêtres connectées. Une saison différente par fenêtre.
- Les fenêtres de terrain changent leur couleur et leur titre en fonction de la saison courante.
- Ajout de particules de pluie et de neige.
  Ces particules sont générées à un emplacement et une altitude aléatoire au-dessus de la surface du terrain. Elles ont également une vitesse de chute aléatoire. Lorsque celles-ci entre en collision avec le terrain elles disparaissent et de nouvelles sont instanciées.
- Les fenêtres affichent ces particules lors de la saison appropriée.
- Ajout de directives #pragma omp afin d'accélérer la création des particules et leur affichage. Cela m'a également permis d'accélérer la vitesse d'affichage des terrains.

# 2) Démarche de développement

### Classe MyTcpServer

J'ai tout d'abord commencé par créer une classe MyTcpServer possédant un attribut de type → QTcpServer* server et un tableau de 4 → QTcpSocket* clients[4] où seront stockés les connexions des 4 fenêtres jouant le rôle des clients.

Le constructeur de cette classe va simplement instancier mon attribut server, vérifier que celui-ci écoute bien sur le port 9999 et connecter le signal QTcpServer::newConnection() à mon slot MyTcpServer::newConnection() que j'ai redéfini dans ma classe.

Il va également connecter le signal timeout() de mon timer avec le slot sendSeason() de ma classe MyTcpServer afin de changer régulièrement (toutes les 10 secondes) les saisons affichées dans les fenêtres clientes.

Ma méthode MyTcpServer::newConnection() récupère simplement la dernière connexion au serveur et la stocke dans mon tableau de QTcpSocket :

```
QTcpSocket *socket = server->nextPendingConnection();
clients[id] = socket;
```

Ma méthode sendSeason() va faire évoluer dans l'ordre les saisons des fenêtres clientes en leur envoyant un paquet contenant la chaine de caractère correspondant à la saison qu'elles doivent afficher.

## Classe GameWindow

Ma classe GameWindow comporte un attribut QString season lui permettant de connaître sa saison actuelle et l'afficher dans le titre de la fenêtre ainsi que sur le terrain en changeant simplement sa couleur en fonction de la saison. Elle possède également un attribut QTcpSocket *socket lui permettant de se connecter au serveur et ainsi recevoir les informations concernant le changement des saisons.

Pour cela, j'ai créé la méthode GameWindow::doConnect() qui permet de réaliser les connexions entre les différents signaux et slots (que j'ai dû redéfinir) propre au QTcpSocket.

```
connect(socket, SIGNAL(connected()),this, SLOT(connected()));
connect(socket, SIGNAL(disconnected()),this, SLOT(disconnected()));
connect(socket, SIGNAL(readyRead()),this, SLOT(readyRead()));
```

Mais cela va également permettre la connexion au serveur grâce à :

```
socket->connectToHost("127.0.0.1", 9999);

// we need to wait...
if(!socket->waitForConnected(5000))
{
    qDebug() << "Error: " << socket->errorString();
}
```

Si la connexion s'effectue correctement, le slot *connected()* sera déclenché et si le serveur s'éteint le slot *disconnected()* sera lancé.

Lorsque le serveur envoi des données aux clients via la méthode MyTcpServer::sendSeason() que l'on a vu précédemment, le slot GameWindow::*readyRead*() est déclenché, va stocker les données reçues dans l'attribut season = QString(socket->readAll()) et mettre à jour le titre de la fenêtre.

## Création de particules

Au départ, j'initialise toutes mes particules dans un tableau contenant **MAX_PARTICULES** (1000)

Afin de pouvoir optimiser la détection de collision des particules avec le terrain, celles-ci ne sont pas générées à des positions totalement aléatoires.

En réalité, elles sont créées à un emplacement aléatoire mais correspondant à la position d'un point existant sur le terrain.

```
particles part;
int rand_x, rand_y, id;

rand_x = rand() % m_image.width();
rand_y = rand() % m_image.height();
id = rand_y * m_image.width() + rand_x;

part.x = p[id].x;
part.y = p[id].y;
```

Ainsi, avec cette méthode, je peux connaître directement l'altitude du point avec sa composante Z, qui correspond à la valeur de l'image à cet emplacement, et savoir précisément à quel moment supprimer ma particule et cela avec un simple test de valeur :

*Si (position_particule.z < position_point.z) supprimer particule*

Je stocke donc la composante Z du point en question dans une variable *min_z* de ma structure de particules :

part.min_z = p[id].z;

Ma particule sera également créée à une hauteur aléatoire (entre 0.3 et 0.5) et aura une vitesse de chute également aléatoire (entre 0.001 et 0.01).

J'affiche ensuite dans le *render()* mes particules lorsque c'est l'automne (particule bleue) ou l'hiver (particule blanche) grâce à ma méthode GameWindow::displayParticles()

Elle va simplement afficher les particules de mon tableau en faisant diminuer leur altitude par leur vitesse de chute et vérifier si la particule a atteint le sol ou pas.

```
glVertex3f(tab_particles[i].x, tab_particles[i].y, tab_particles[i].z);
tab_particles[i].z -= tab_particles[i].falling_speed;

if(tab_particles[i].z < tab_particles[i].min_z){
        tab_particles[i] = newParticle();
}
```

J'ai terminé ce TP en ajoutant des directives #pragma omp afin d'accélérer le processus de création et d'affichage des particules.

## 3) Structure de données

Pour réaliser les particules de pluie/neige sur les terrains, j'ai créé une structure me permettant de connaître la position de chaque particule, leur vitesse de chute et l'altitude minimale qu'elles peuvent atteindre (avant qu'elles ne touchent le terrain)

```
struct particles
{
    float x, y ,z;
    float falling_speed;
    float min_z;
};
```

Je les stocke ensuite dans un tableau de **MAX_PARTICLES** = 1000

particles* tab_particles;

## 4) Parties bonus

Un moyen d'accumuler les particules et les transformer en surface comme une rivière ou des tas de neige aurait été de ne pas supprimer les particules une fois le terrain atteint mais de les entasser en leur appliquant une gravité en fonction de leur caractéristique.

Par exemple, la pluie aura tendance à glisser le long du flanc des montagnes et se regrouper dans des creux pour former des rivières ou des flaques. Tandis que la neige aura tendance à s'accrocher sur les parois des montagnes alors que d'autres particules s'entasseraient par-dessus pour former des tas de neige.

Ces particules auraient un système de boite englobante qui leur permettrait de gérer les collisions avec les autres particules et de ne pas être rassembler en une même position.