**Dossier technique**

***Cours de Projet dans un Environnement Graphique***



Table des matières

[Diagramme de classe 3](#_Toc352597530)

[Diagramme d’origine : 3](#_Toc352597531)

[Diagramme actuel : 4](#_Toc352597532)

[Fonctionnement client-serveur 5](#_Toc352597533)

[Schéma d’échange client-serveur 5](#_Toc352597534)

[Explication 6](#_Toc352597535)

[Lors du lancement du serveur 6](#_Toc352597536)

[Lors de la création d’un client 6](#_Toc352597537)

[Problématiques et solutions 7](#_Toc352597538)

[Jimmy Turgeon 7](#_Toc352597539)

[Alexis Vuillaume 7](#_Toc352597540)

[Cédric Outreville 7](#_Toc352597541)

[Compromis en architecture 8](#_Toc352597542)

[Jimmy Turgeon 8](#_Toc352597543)

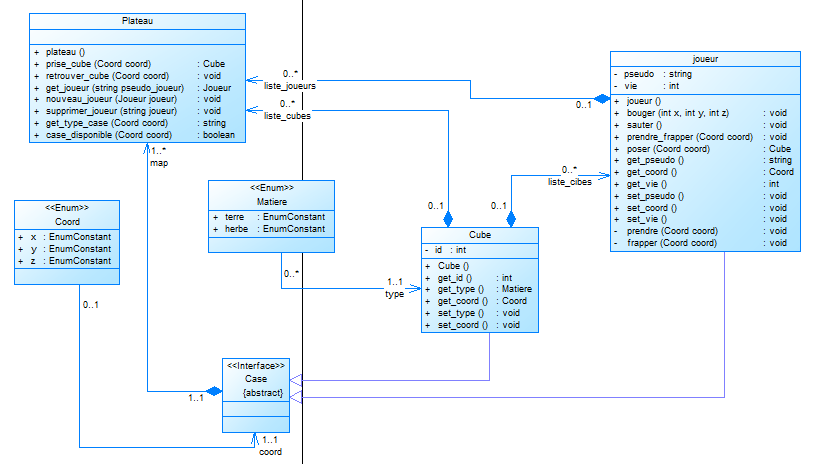
[Cédric Outreville 8](#_Toc352597544)

[Alexis Vuillaume 8](#_Toc352597545)

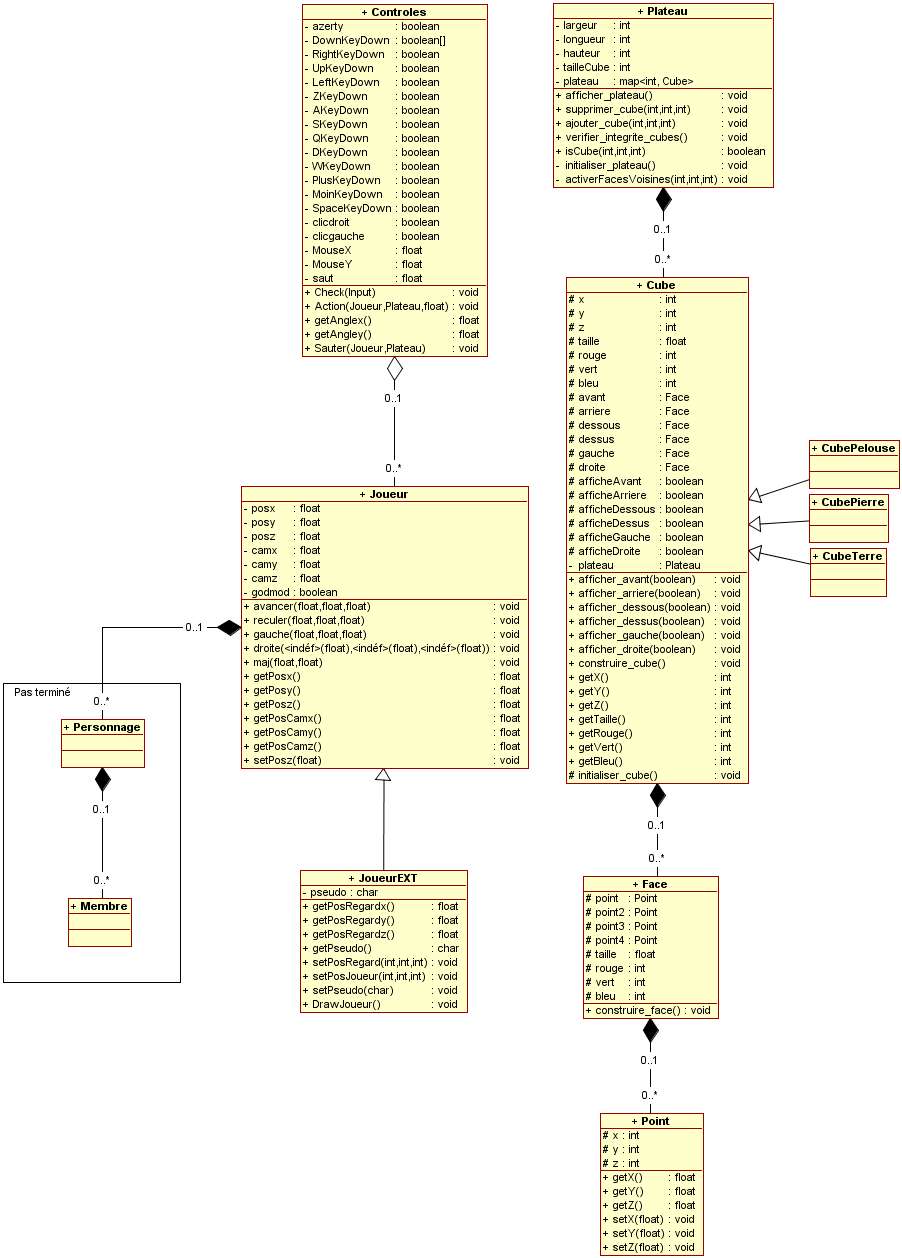
[Documentation HTML (Doxygen) 8](#_Toc352597546)

# Diagramme de classe

## Diagramme d’origine :

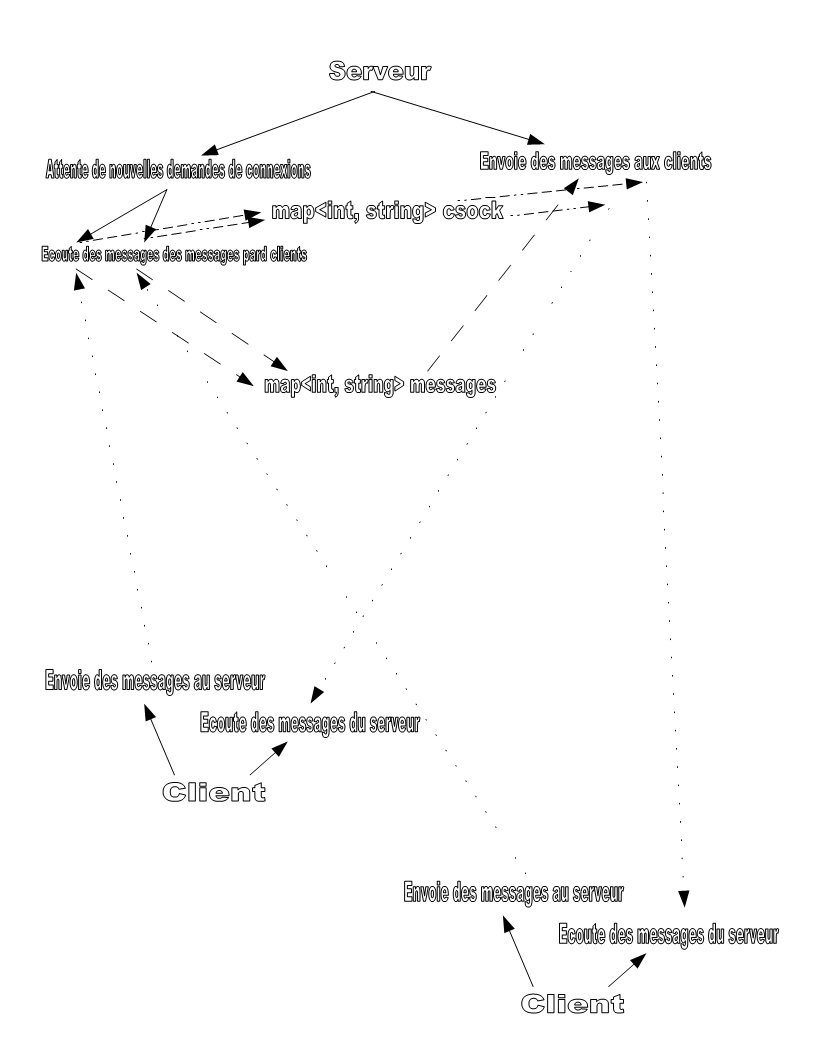


## Diagramme actuel :



# Fonctionnement client-serveur

## Schéma d’échange client-serveur



## Explication

### Lors du lancement du serveur

Lors du lancement du serveur, deux threads sont lancés.

Le premier thread attend des demandes de connexion des clients. Pour ce faire, le client va donner son nom. Ce thread créera un nouveau thread avec comme nom, celui du client pour chaque nouveau client. Si le nom existe déjà, la demande de connexion sera jetée. De plus, le nom du client et son identificateur d'échange (correspondant au socket qui lui est attribué pour les échanges) sera mis dans une liste « csock ». Le serveur identifie donc ces clients et leurs messages grâce à un string envoyé par le client.

Ces nouveaux threads créés écoutent le client qui leur a été affecté. Chaque fois qu'un message sera reçu pour ce client, le thread mettra le nouveau message dans une liste d'attente « messages » avec son nom comme identifiant du message. S'il y a déjà un message de se client, le nouveau écrasera l'ancien message.

Le second thread créé au lancement du serveur, envoi les messages à tous les clients. Dès lors que la liste « messages » n'est pas vide, le message sera envoyé à tous les clients de « csock ». Puis le message sera écrasé de manière sécurisé avec des mutex. Si un client à un problème de réception alors tant pis pour lui, aucun nouveau message ne lui sera renvoyé. Mais comme dans notre contexte, chaque client envoie 60 messages par secondes, cela ne créé pas de latence. Elle l'empêche même!

### Lors de la création d’un client

Lors de la création d'un client, il y a tentative de connexion au serveur puis si tout se passe bien, notre nom est envoyé au serveur. Ensuite, deux threads sont également lancés. Le premier ecoute, le second envoi.

Tout comme le serveur, ces threads sont associés à des listes. La réception de message insert le message reçu dans une file d'attente « messages\_recu » et l'envoi va chercher dans la liste « message\_envoi » les messages, les envois et les supprime de la liste.

Ces threads sont invisibles pour l'utilisateur de la classe « Client ». L'utilisation des méthodes envoyer() et recevoir() permettrons de donner les messages à envoyer et de récupérer les messages reçus. envoyer() va mettre dans la file « message\_envoi » et recevoir() va prendre le premier message de « messages\_recu » puis le supprimer.

# Problématiques et solutions

## Jimmy Turgeon

Mon principal problème fût l’apprentissage du langage C++, puisque les cours que nous avions eu lors de la première année de notre technique ne couvrait que les bases de la programmation et la syntaxe commençait aussi à se faire loin.

Par la suite, il a fallu réapprendre le principe inutilement compliqué des pointeurs afin de pouvoir faire de l’allocation dynamique lors de la création du plateau de jeu. Ce dernier comporte tout de même 100 000 milles cubes et l’optimisation est de rigueur.

Finalement, il y a eu le choix d’une collection pouvant répondre à nos besoins pour la création du plateau. Cette dernière devait pouvoir contenir un nombre de cube définit en x et en y, mais indéfinis pour les z. Nous avions à la base un tableau 3D de cubes, la hauteur du plateau ne pouvait être changée. Nous donc changé pour un tableau 2D de vecteurs, mais ces derniers ne peuvent contenir d’espace vide alors il aurait été nécessaire de créer des cubes vides ce qui ruinait nos efforts d’optimisation. Nous avons finalement opté pour un tableau 2D de map. Ainsi, il nous était possible de gérer les espaces vides et d’ajouter des cubes en hauteur de façon presque infinis.

## Alexis Vuillaume

Le problème technique le plus important que j’ai rencontré touchait surtout à la géométrie de l'espace. En effet, pour créer une caméra capable de reproduire une vision classique type First Person Shooter.

Voici le calcul permettant de calculer le point qui est fixé par le joueur lors du déplacement. http://uppix.fr/?di=4D9J

Ce point sert à positionner la caméra de façon à regarder constamment le point de coordonnée x, y, z.

Trouver cette formule et faire en sorte qu'elle marche n'a pas été aisé, et a requis beaucoup de test afin d'en ajuster la sensibilité pour correspondre au besoin du jeu.

## Cédric Outreville

Tout d’abord, l’utilisation d'une machine virtuelle empêche la communication sur le réseau externe, ce qui nécessite une configuration supplémentaire. Ensuite, l’utilisation de string pour les envois entre les sockets est difficile, car la structure de ces derniers n’est absolument pas propice ! Finalement, il est important de ne pas compliquer les choses quand elles sont simples, même s’il s’agit de permettre la connexion avec plusieurs clients.

# Compromis en architecture

## Jimmy Turgeon

J’ai dû changer trois fois de collections afin de gérer le plateau de jeu, mais j’ai finalement opté pour une map comme mentionné dans la partie précédente.

## Cédric Outreville

J’ai dû faire un réusinage du code pour envoyer non pas des string mais des char[] lors des échange client-serveur. J’ai aussi dû effacer un thread qui compliquait les choses et n'apportait rien d’essentiel.

## Alexis Vuillaume

J'ai dû repenser comment fonctionnais un joueur en séparant les contrôles des actions, ce qui a créé 2 classes distinct, Contrôle et Joueur

# Documentation HTML (Doxygen)

Pour ouvrir la documentation HTML, il suffit d’ouvrir le lien suivant : [Documentation](html/index.html).