Rapport du projet informatique

Roiron Yohann - Groupe 2

27 mars 2015

Table des matières

I Analyse du travail effectu[Pleaseinsertintopreamble] 2

1 Présentation du sujet 3
1.1 Principe général . . . 3
1.2 L'interface graphique . . . 3

2 Analyse de la solution envisagée et mise-à-jour 5
2.1 Découpage en modules . . . 5
2.2 L'interface graphique (client) . . . 6
2.3 Le réseau . . . 8
2.3.1 Le principe général . . . 8
2.3.2 La partie client . . . 8
2.3.3 La partie serveur . . . 8
2.4 Le cœur algorithmique . . . 9

3 Échéancier 11

II Évolutions vis-à-vis de l'avant-projet 12

III Explications des choix techniques 14
3.1 Le réseau . . . 15
3.2 La gestion des barres . . . 15
3.3 La gestion du jeu . . . 16

IV Le code partiel du programme 18

Première partie

Analyse du travail effectué

Chapitre 1

Présentation du sujet

1.1 Principe général

Le sujet choisi a pour intitulé **Babyfoot en réseau**. Il s'agit de concevoir un système complet de jeu en réseau. Le système serait donc séparé en deux parties, un serveur et un client. Il faut donc réaliser à la fois le système réseau, l'interface graphique, un petit moteur physique pour les collisions et imaginer un gameplay qui rende le jeu agréable.

1.2 L'interface graphique

Par choix, j'ai décidé de faire une interface graphique la plus légère possible, il n'y a donc pas de menus, car je n'ai pas proposé d'options de gamePlay, préférant me concentrer sur la partie réseau et intéraction avec l'utilisateur, bien plus intéressante selon moi. Ces fenêtres seront réalisées avec la bibliothèque inclue dans le package standard **Swing**. J'ai aussi choisi d'utiliser des JFrame et donc de programmer une application à part entière et non une appliquette, car je ne voyais pas l'intérêt d'utiliser un navigateur pour jouer.

Page d'acceuil Une première fenêtre s'ouvre au lancement du programme, elle se veut relativement simple, et c'est à l'intérrieur que tout se passe, c'est l'unique fenêtre du programme. Un PopUp d'aide permet d'expliquer simplement les règles du jeu, et de rappeler les touches à utiliser.

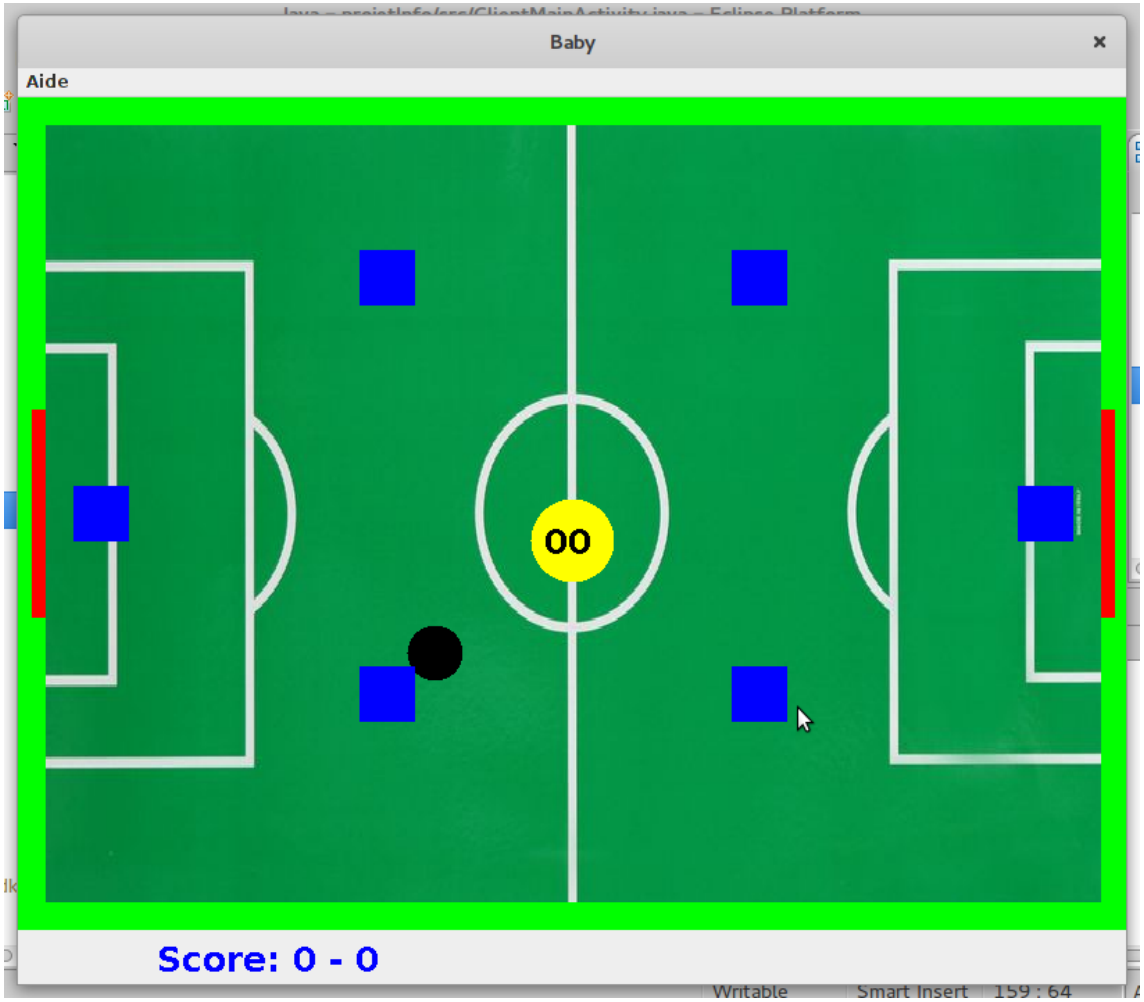

FIGURE 1.1 – Image d'acceuil du programme, à la fois simple et efficace.

Commencer une partie Le jeu a été fait au plus simple, pour commencer une partie il suffit de lancer le programme en lui passant en parramètre, l'adresse IP du serveur. Il faut simplement que le serveur soit déjà lancé. Une fois que les deux jours sont connectés, la partie commence automatiquement au bout de 10 secondes, permettant aux joueurs de se préparer, et de tester leurs commandes.

Déroulement d'une partie Le gameplay est entièrement manuel et voit le joueur maître de ses possibilités. Chaque Joueur contrôle deux lignes de joueurs, en angle et en hauteur, via une combinaison de touches. Chaque joueur, joue indépendament sur sur son programme et essai donc de marquer des buts. En cas de But, la balle repart au centre et on continue à jouer, un petit compteur permet aux joueurs de se préparer.

Dès que le score max est atteint, on affiche les scors. Et on déclare le vainqueur via des PopUp.

Options Le jeu se veut simplifié au maximum, pour ne pas avoir de paramètres complèxes à gérer, et ainsi pouvoir jouer directement. Il n'y a donc pas d'options, même si certains paramètres ont été codé pour être modifié simplement, nottament, le nombre de buts qu'il faut inscrire pour gagner. La seule chose dont il faut se préocuper, c'est la précense d'un serveur, sinon le jeu refuse de démarrer.

Chapitre 2

Analyse de la solution envisagée et mise-à-jour

2.1 Découpage en modules

J'ai donc découpé le code en deux grandes parties :

— Partie client

— Partie serveur

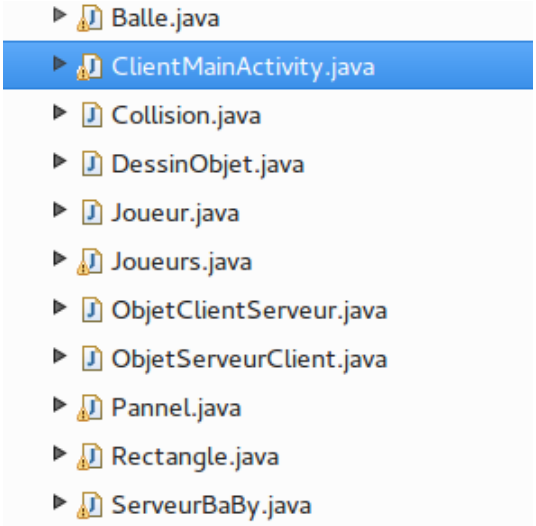

FIGURE 2.1 – Structure des fichiers du projet.

Les deux parties (client/serveur) seront elles-mêmes séparées en plusieurs modules.

— L'interface graphique (rangée dans Pannel.java) qui sera utilisé sur le serveur seulement pour le débuggage mais qui consiste en une énorme part du client, et une des plus importantes.

— La partie réseau de l'application (rangée dans les objets), qui forme les requêtes dans les deux sens.

— Le cœur algorithmique de l'application (rangée dans Collision) contenant les classes gérant œur du serveur qui se dans font l'ensemble des calculs du jeu.

Mais j'ai fait le choix, qu'une classe représente un objet ou une fonction, ainsi la classe balle, gère à la fois, la position de la balle, sa vitesse et son dessin, c'est donc cette classe qui est utilisé pour faire les collisions (c'est-à-dire passé en paramètre de la classe collision).

2.2 L'interface graphique (client)

ClientMainActivity.java Gère la fenêtre qui englobe tout le reste. On utilisera en fait des JPanel pour modifier le contenu de cette fenêtre, dont celui définit dans Pannel. J'empêcherai dans un premier temps de modifier la taille de la fenêtre pour éviter d'avoir des problèmes de dessin du terrain de babyfoot à gérer. Cette classe contient le **main** du programme. C'est elle qui lance le jeu et appelle le Jpannel Cette classe a deux autres fonctions : La première c'est de cadencer le dessin du programme client, de scruter l'appui des touches, à 100hz, et de renvoyer tous cela au serveur Via l'objet *ObjetClientServeur.java* Cette classe, contient aussi une classe interne, héritant de Runnable, qui a pour objectif d'écouter le serveur à 200hz (c'est-à-dire que si il n'y a rien à écouter, on attend 5ms avant de recommencer) et de modifier en conséquence les attributs de la classe mère.

Panel.java Classe relativement simple, elle contient une liste d'objets dessinObjet, qu'elle initialise, puis dessine à l'appel de la fonction Main, (elle est aussi utilisable sur le Serveur pour le débuggage).

DessinObjet.java Classe abstraite définissant les méthodes communes à tous les objets destinés à être déssiné.

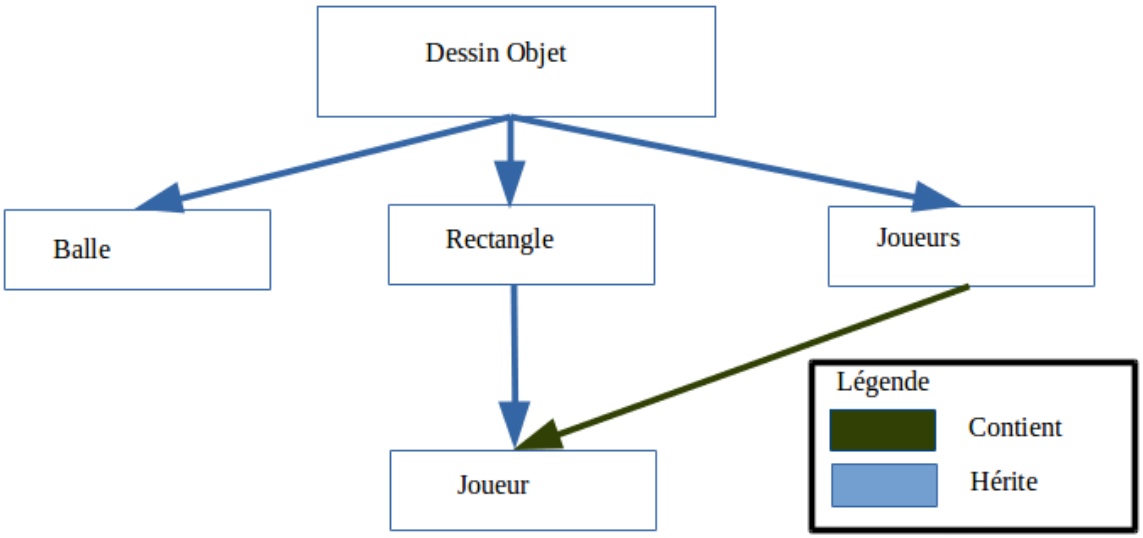

FIGURE 2.2 – Schéma des Structures d'héritage du projet.

Panel.java Classe relativement simple, elle contient une liste d'objets dessinObjet, qu'elle initialise, puis dessine à l'appel de la fonction Main, (elle est aussi utilisable sur le Serveur pour le débuggage).

Panel.java Classe relativement simple, elle contient une liste d'objets dessinObjet, qu'elle initialise, puis dessine à l'appel de la fonction Main, (elle est aussi utilisable sur le Serveur pour le débuggage).

Panel.java Classe relativement simple, elle contient une liste d'objets dessinObjet, qu'elle initialise, puis dessine à l'appel de la fonction Main, (elle est aussi utilisable sur le Serveur pour le débuggage).

FIGURE 2.3 – Image des options de l'écran de configuration avec la présence d'un démonstrateur à gauche pour tester la sensibilité.

2.3 Le réseau

2.3.1 Le principe général

Des messages sont échangés entre le client et le serveur et ce de façon répétée. Il faut donc utiliser des Thread pour gérer l'émission et la réception de données depuis le serveur et un Thread pour gérer la réception côté client. Le serveur se chargera ensuite d'enregistrer les données statiques devant être conservées.

2.3.2 La partie client

ClientBabyfoot.java Il s'agit du main du client qui initialise les fenêtres et les différentes classes utilisées.

> private static Player player ; private static Chat chat ; private static Client client ;

Client.java Cette classe gère la connexion au serveur : il y a en tout trois connexions (une pour le tchat, une pour les joueurs et une pour les données des matchs) afin de pouvoir envoyer plusieurs requêtes simultanément du même client. Toutes les requêtes envoyées sont sous la forme de chaîne de caractères avec l'utilisation d'un caractère spécial, qui est contenu dans une constante de la classe Utils du Core comme séparateur.

> private static Socket socketChat ; private static Socket socketPlayer ; private static Socket socketMatch ; private static Socket socketGame ; private ChatClient cc ; private PlayerClient pc ; private MatchClient mc ; private GameClient gc ; private Thread tChat ; private Thread tPlayer ; private Thread tMatch ; private Thread tGame ;

PlayerClient.java Gère les différentes actions possibles par et sur les joueurs et les requêtes envoyées au serveur pour pouvoir satisfaire le système (connexion d'un joueur, déconnecter un joueur, ajouter un match, etc.). Elle contient aussi un Thread qui écoute l'entrée de la socket.

MatchClient.java Procède de la même façon que PlayerClient pour les matchs, envoie les requêtes au serveur et récupère les données sur la position de la balle, des barres des différents joueurs, etc.

ChatClient.java S'occupe de gérer la partie client du tchat : une fois instanciée par le ChatPanel, elle permet d'envoyer des données au serveur et de récupérer la liste des messages disponibles dans le salon. Elle permet aussi de changer de salon, d'afficher la liste des salons et celle des joueurs connectés sur le jeu.

2.3.3 La partie serveur

FIGURE 2.4 – Structure des fichiers du serveur.

AbstractServer.java Classe abstraite qui gère quelques méthodes de base que doivent toutes présenter les classes gérant les serveurs.

ServerBabyfoot.java Gère les différentes requêtes et les redirige vers les autres entités du serveur (chat, jeu, etc.). Contient un système de Thread pour pouvoir gérer simultanément ces différentes actions, un Thread général qui gère les requêtes. C'est aussi le main qui lance le serveur.

> ServerSocket socketserver = null ; Socket socket = null ; BufferedReader in ; PrintWriter out ; String login ; public static ChatServer tchat ; public static PlayerServer tplayer ; public static MatchServer tmatch ;

PlayerServer.java Gère la connexion au serveur d'un des joueurs. L'enregistre dans une liste de joueurs connectés. Peut aussi renvoyer la liste des joueurs connectés actuellement et leur état.

MatchServer.java Gère tous les échanges de données sur les matchs. S'occupe ensuite de fournir les informations nécessaires au client sur l'état actuel de la position de la balle par exemple. Nécessitera peut-être, pour des questions de réactivité, d'être multi-threadé.

ChatServer.java S'occupe de gérer la partie serveur du tchat : écoute un port, récupère les messages envoyés par des joueurs ainsi que le salon où ils se trouvaient à ce moment là. Les enregistre dans un fichier ou bien dans une base de données (à voir).

2.4 Le cœur algorithmique

Player.java Le bloc de base du cœur algorithmique est le joueur. Il est instancié par le serveur et a plusieurs attributs : son login, son état actuel (s'il est en train de jouer ou non). Si le joueur est en match, la classe contient aussi les barres qu'il a le droit de déplacer.

> private Match match ; private String login ;

Utils.java Est une classe abstraite qui gère les fonctions utilitaires comme la mise-en-forme de la date, la mise-en-forme des requêtes, les fonctions de hash utilisées pour vérifier la validité des requêtes, etc. Commune au client et au serveur. Cette classe contient surtout les constantes du programme ainsi que les types créés (enum) pour repérer le côté du joueur, les barres, le statut du match, le type de collision, etc.

> public static enum Types ONEVSONE, TWOVSTWO, ONEVSTWO ; public static enum States WAITING, FULL, PLAYING, FINISHED ; public static enum Sides DOWN, UP ; public static final int MATCH_END = 100 ; public static enum Rod GARDIEN , DEFENSE, MILIEU, ATTAQUE ; public static enum RodStatus NORMAL, SHOOTING, HOLDING ; public static enum CollisionType SIDES, UPANDDOWN ; public static final String SEPARATOR = " ;" ; public static final int GOAL_SIZE = 2*100 ; public static final int LINE_STRENGTH = 4 ; public

static final int GAP_EDGE = 2*20 ; public static final int IMAGE_PLAYER_Y = 38 ; public static final int IMAGE_PLAYER_X = 30 ; public static final int MOVE_STEP = 10 ; public static final int BALL_RADIUS = 15 ; public static final int HEIGHT = 700 ; public static final int WIDTH = 900 ; public static final int MAX_INITIAL_SPEED = 4 ; public static final int GARDIEN_POSITION = GAP_EDGE+30 ; public static final int DEFENSE_POSITION = GAP_EDGE+30+100 ; public static final int MILIEU_POSITION = (WIDTH-LINE_STRENGTH)/2-70 ; public static final int ATTAQUE_POSITION = WIDTH-Utils.LINE_STRENGTH-Utils.GAP_EDGE-230 ;

Match.java Contient les données principales pour une partie, notamment l'avancement de la partie, l'état des scores, les joueurs y participant, etc. Sera appelée par le serveur. Cette classe contient aussi toutes les informations sur les données factuelles d'une partie, à savoir la position de la balle, la position des barres, etc. Elle se charge d'effectuer l'ensemble des calculs de déplacements de la balle, comptabilise les buts, met en place les pauses.

private int leftScore ; private int rightScore ; private Types type ; private States state ; private Player player1 ; private Player player2 ; private Player player3 ; private Player player4 ; private float ballX ; private float ballY ; private float ballSpeedX ; private float ballSpeedY ; private Collisions collisions ; private boolean noSlow = true ; private boolean pause = false ; private final int STEP_X = 2 ; private final int STEP_Y = 2 ; private int status = 0 ;

Collisions.java Gère tout ce qui touche aux collisions, avec les joueurs ou les bords. C'est là que se situe toute la difficulté de calcul. Elle contient donc des attributs pour la gestion de la balle ainsi qu'une Hashtable à deux niveaux stockant les dernières collisions. L'intérêt de la HashTable est ici clairement qu'elle permet de stocker les temps en fonction de la barre et du côté où est situé le joueur.

private float ballX ; private float ballY ; private float ballSpeedX ; private float ballSpeedY ; private Hashtable<Sides, Hashtable<Rod, Long>>lastCollision ;

Database.java Aucune base de données n'a finalement été utilisée. Le serveur stocke donc ses données dans la RAM puis les supprime lors de sa fermeture. Dans un souci de pérennité et de performances, il serait intéressant de les stocker dans des fichiers textes que l'on puisse ainsi conserver des archives.

Chapitre 3

Échéancier

Ce qui était prévu

1. **Fin Décembre** : rédiger l'avant-projet.
2. **Début/Mi Janvier** : obtenir la validation et les annotations sur la structure choisie.
3. **Fin Janvier** : réaliser la partie graphique du programme et avoir regardé les grandes lignes du développement serveur / bases de données. Avoir mis au point les éléments de base du gameplay (interaction joueur/machine).
4. **Fin Février** : développer le serveur et la gestion des différents types de requêtes. Mettre au point le tchat et la gestion des joueurs avec la base de données.
5. **Mi Mars** : dresser les liens entre serveur et jeu. Tester.

Ce qui a été fait

1. **Fin Décembre** : rédiger l'avant-projet.
2. **Fin décembre/Début Janvier** : développer la partie réseau et l'architecture de base des requêtes.
3. **Fin Janvier** : réalisation de l'interface graphique et des premiers calculs pour la mise-en-place de matchs.
4. **Février** : ajout des collisions et corrections des principaux bugs liés à la gestion des joueurs, matchs et chat.
5. **Mars** : débuguer le reste du programme et ajouter les fonctionnalités supplémentaires nécessaires à un bon fonctionnement.

Deuxième partie

Évolutions vis-à-vis de l'avant-projet

FIGURE 3.1 – Capture d'écran du jeu lui-même et de son aspect.

Le projet a été globalement mené à son terme bien qu'il subsiste de nombreux points à améliorer. La programmation de ce projet relativement ambitieux (quelques 5000 lignes de code jusque là) m'a toutefois permis d'appréhender la mise-en-place d'un réseau en java et le dessin d'une interface graphique. Il aurait pu être intéressant, devant l'envergure que peut prendre un tel projet, de chercher à implanter une stratégie *MVC*, Modèle Vue Controleur, sur le client afin de faciliter l'édition de l'interface graphique et la gestion des données reçues du serveur. C'est ce qui a été partiellement fait : les fichiers Gui sont stockés dans un dossier, les classes gérant les modèles sont faites dans Core et les classes récupérant les données sont dans Network. Néanmoins, la partie logique est en bonne partie gérée dans les vues et non dans les contrôleurs, ce qui gâche toute la logique du MVC.

Au niveau du développement lui-même L'*architecture*, bien que réfléchie lors de l'avant-projet, n'est peut-être pas suffisamment stable pour supporter une plus grande charge des clients. Les requêtes peuvent se télescoper, arriver au mauvais client, être redirigée vers le mauvais serveur et des accès simultanés peuvent mal modifier les mêmes informations. L'*interface graphique* doit pouvoir être redessinée, d'abord par souci esthétique et ensuite afin d'intégrer la possibilité pour l'utilisateur de la redimensionner, pour l'adapter à un écran de tablette par exemple.

Pour ce qui est de la structure J'ai rajouté quelques classes au projet initial mais toutes n'ont pas été mentionnées ci-dessus car certaines ne sont parfois que des classes implantant l'interface Runnable et initiant des Thread. Globalement, la structure est restée la même, en séparant toutefois bien client et serveur néanmoins, afin de laisser la possibilité que le serveur soit lancé sur une machine indépendante tandis que le client est distribué.

Pour ce qui est du code J'ai essayé d'utiliser le plus souvent possible des variables de type *enumerate* afin de rendre plus facile la lecture du code du programme. Il y a relativement peu de commentaires sur l'ensemble du projet car il y a très peu de points très techniques méritant une explication. Le plus difficile est de comprendre l'agencement des classes et leur hiérarchie dans la chaîne aboutissant au jeu.

Choix techniques J'ai bien sûr utilisé **Swing** pour réaliser l'interface graphique : d'abord pour la simple raison que le double-buffering est automatiquement utilisé par Swing, ensuite parce que la structure d'une fenêtre est beaucoup plus souple en ce qui concerne la réalisation de quelque chose d'un peu plus complet. En ce qui concerne le système réseau, j'ai tout simplement utilisé des sockets ainsi qu'elles sont décrites dans le polycopié du cours d'informatique.

Troisième partie

Explications des choix techniques

3.1 Le réseau

Le réseau est au coeur de mon projet et j'ai donc développé plusieurs Thread en parallèle pour pouvoir gérer plus facilement l'envoi de messages par mon application. Il a fallu mettre au point une nomenclature des messages afin d'obtenir successivement le type de message envoyé et le domaine qui est atteint, donc la classe effectuant le traitement de la requête, puis le joueur concerné (c'est le login qui est utilisé ici bien qu'un système d'ID aurait pu être mis en place) et enfin les données propres à la requête.

De plus, le serveur recevant un grand nombre de requêtes, j'ai séparé le serveur en plusieurs entités afin de faciliter le traitement. On trouve ainsi une classe dédiée entièrement à la gestion du Chat et des requêtes échangées. Il y a aussi un serveur se chargeant de gérer les joueurs et enfin, le plus gros des trois, un serveur s'occupant des requêtes de jeux. Il est assez sollicité puisqu'une requête est envoyée aux 2, 3, 4 clients toutes les 20 ms environ (un peu plus parfois). Ces requêtes peuvent être assez longues puisqu'elles contiennent de nombreuses informations : la position des barres, leur statut (normal, levée, tirant) ce qui représente 16 coordonnées puis les huit barres.

L'intervalle de temps entre l'envoi de la requête et la réception est aussi légérement variable et est difficilement quantifiable puisque cela dépend de la charge de l'ordinateur, de la longueur du message et de l'utilisation du client au moment de la réception du message. Il m'a donc fallu faire patienter le client ou le faire utiliser des données « périmées ».

FIGURE 3.2 – Sorte de *switch* du projet, permettant de séparer les requêtes selon les différents « serveurs ».

Afin d'envoyer un message au serveur, depuis l'interface Gui, par exemple lorsqu'une demande de pause est faite, le schéma est simple. L'interface appelle, via la classe Main, une des classes du Core du client, Player.java par exemple. Cette classe se charge ensuite d'appeler la bonne fonction de la partie Network afin d'envoyer les informations au serveur. Elle récupère par la valeur retournée par la méthode la réponse du serveur et la renvoie à l'interface, en l'ayant traitée au préalable si nécessaire.

3.2 La gestion des barres

Un point a été intéressant du point de vue algorithmique, c'est la gestion des barres. Suivant le type de match, un joueur a accès à certaines barres. Par exemple, quand il est seul contre deux joueurs, il a accès à toutes les barres. Par contre, s'il n'est pas seul, le calcul est fait automatiquement selon l'ordre d'arrivée des joueurs dans la partie. Ce qui a été intéressant, c'est de trouver une façon de stocker ces informations. Il fallait ici obtenir une méthode simple pour définir « l'utilisation »des barres, lesquelles sont occupées et lesquelles sont libres. Ce problème est très similaire à un problème d'autorisations et de droits. En effet, a-t-on le droit ou non de bouger les barres ? Comment stocker cette donnée ?

Pour cela, ma technique a été très simple, c'est d'utiliser la notation pour binaire et associer à chaque bit une autorisation particulière. Je code ici les autorisations sur quatre bits, chacun représentant deux barres, d'un côté ou d'un autre. Ainsi, les deux premiers bits (valeurs correspondant à 2^0 et 2^1) correspondent aux barres du bas et les deux autres à celles du haut. Lorsque

le statut des barres est actualisé, *i.e.* qu'un joueur entre dans la partie, cet entier est testé pour situer dans quel cas se situe le jeu et où des barres sont disponibles puis l'entier d'utilisation est mis à jour en ajoutant 1, 2, 3, 4, etc. pour signifier quelles barres va prendre le joueur. De même lorsqu'un joueur quitte la partie, l'entier voit sa valeur soustraite par celle représentant le joueur. On a ainsi une façon simple d'accéder à la possibilité pour un joueur de toucher à une barre puisqu'il s'agit d'utiliser les opérateurs binaires pour tester une autorisation. Ce nombre est stocké par le serveur et est transmis très facilement aux clients qui stockent eux les autorisations dans une Hashtable.

Il eut été possible de s'affranchir de toutes ces considérations en réalisant un écran supplémentaire avant le début de la partie où le joueur hôte de la partie aurait tout simplement assigné telle ou telle barre à tel ou tel joueur. La gestion se serait alors faite avec une Hashtable contenant des noms de joueurs par exemple. Il aurait été toutefois intéressant de stocker la même information de « l'utilisation »des barres afin de vérifier que toutes les barres ont bien été assignées à quelqu'un.

La sensibilité d'une barre est aussi très difficile à régler. J'utilise le déplacement relatif de la souris, c'est-à-dire les mouvements « verticaux » de la souris pour calculer les déplacements des barres. Néanmoins c'est assez difficile car pour une raison que j'ignore, ces déplacements sont asymétriques : la sensibilité est plus grande du haut vers le bas et moins du bas vers le haut. Il faut donc trouver une équation de déplacement qui permette à la fois d'être précis et réactif. L'équation de la différence entre la position après calcul et avant calcul est donnée par la formule ci-dessous :

```
mov = (int)Math.ceil( ( Math.abs( lastKeyY-y )* Utils.getSensibility() / ( 1.+Utils.getSensibility()
) ) ) ;
```

FIGURE 3.3 – Déplacements de la souris servant à mouvoir les barres.

3.3 La gestion du jeu

Pour calculer la position de la balle au cours du temps, j'ai utilisé un Thread qui s'actualise toutes les 10ms et qui effectue à chaque tour d'horloge l'actualisation de la position à partir de la vitesse et des tests de collisions. Les tests de collisions sont calculés à part dans une classe appelée Collision. Je vérifie ainsi la position de la balle par rapport aux bords et détermine si une collision a eu lieu avec une de ces limites. Une fonction est appelée aussi afin de déterminer si il n'y a pas eu de contact avec un des joueurs du terrain.

La principale difficulté algorithmique vient de cet aspect justement. Parvenir à prévoir les différents comportements de la balle par rapport aux joueurs. Les coordonnées utilisées par ces calculs diffèrent en plus parfois de celles d'affichage et il faut prendre en compte la taille de l'écran ainsi que les imprécisions dues au caractère discret du calcul. J'ai encore parfois des problèmes de collisions qui se produisent trop tôt, c'est-à-dire que le joueur humain a l'impression qu'elle se cogne contre un objet invisible, ou encore des collisions gérées trop tard auquel cas la balle rentre partiellement dans le joueur en plastique.

Une autre difficulté a été de déterminer dans quel sens va la balle après le rebond. En effet, il faut parvenir à placer la balle dans une des quatre zones déterminées par les bissectrices issues des coins du joueur. Si la balle arrive sur les faces avant ou arrière du joueur, elle doit donc voir sa vitesse horizontale changer de sens. Si elle arrive sur les faces de côté, elle doit alors voir sa vitesse verticale changer. Il faut donc calculer la pente de la droite formée par les deux points suivants, le centre de la balle et un des quatre coins du joueur que l'on cherche à étudier, puis la comparer avec la pente de la droite qui marque la séparation. Cela permet ainsi de procéder au rebond.

Le problème du caractère discret de ces calculs, c'est que les corrections ne suffisent pas à faire sortir la balle de la zone de collision. Ainsi, malgré le changement de vitesse et la correction effectuée lorsque la balle entre dans une des zones où la collision a lieu, il est possible que la balle reste dans une de ces zones auquel cas, le prochain tour de calcul risque de demander un nouveau changement de vitesse ce qui est bien sûr absurde.

Un des derniers défis techniques a été de régler la sensibilité des barres afin de permettre aux joueurs d'avoir à la fois un jeu réactif, confortable, c'est-à-dire où la souris n'a pas besoin de beaucoup bouger, et en même temps quelque chose de suffisamment précis pour pouvoir shooter dans la balle au bon endroit au bon moment.

J'ai essayé de mettre au point un amortissement de la vitesse de la balle au cours du jeu. Il est difficile de s'en occuper car, en conservant quelque chose de significatif, il arrive que la balle s'immobilise ce qui est légérement problématique.

Quatrième partie

Le code partiel du programme