Rapport de projet Bataille Navale

Auteurs :
Charles Éponville,
Nicolas Gille,
Vincent Metton,
Grégoire Pommier,
Sylvestre Vimard

Responsable :
M. Yannick Guesnet

27 avril 2016

Table des matières

Remerciements

Nous remercions :
M.GUESNET pour avoir proposer le sujet, nous avoir soutenu et encourager tout au long du projet,
M.ANDARY pour avoir répondu à toutes nos question à propos de la manipulation des Threads en java,
M.BEDON pour avoir répondu à toutes nos questions quand au développement Java, serveur côté Java et l'utilisation des Threads,
M.VASSEUR pour avoir répondu à toutes nos réponses liées à la partie réseau du projet,
ainsi qu'à tous ce qui nous ont aidés et que nous ne pouvons pas remercier.

Introduction

De nos jours, il est possible de jouer à des jeux de société même avec des personnes qui ne sont pas dans la même pièce grâce aux différents protocoles réseaux. De plus en plus de jeux sont donc implémentés et utilisent ces technologies pour permettre de jouer en réseau, c'est à dire, de pouvoir jouer entre deux machines distantes.

De cette constatation, le besoin est venu d'implémenter un classique des jeux de société : la bataille navale (ou "touché coulé") qui est un jeu se jouant à deux joueurs dont le but est de couler tous les bateaux de l'adversaire, disposés sur un plateau de jeu. La bataille navale pouvant se jouer avec plusieurs règles, nous avons opté pour "le tour par tour" : chaque joueur joue une et une seule fois puis donne la main à son adversaire.

Ce projet a pour objectif de disposer de deux modes de jeu : un en local permettant de jouer contre une intelligence artificielle et l'autre permettant de jouer contre adversaire physique en réseau.

Dans ce rapport, nous allons présenter les différents points de l'application, notamment en décrivant le modèle du jeu, ensuite la partie réseau, puis l'interface homme machine permettant à l'utilisateur de jouer dans des conditions correctes, enfin nous exposerons dans une dernière partie les différentes connaissances que ce projet a apporté, les difficultés rencontrées ainsi que l'évolution possible de ce projet.

1 Modèle de jeu

1.1 Le plateau de jeu

Dans le cadre de ce projet, nous avons choisi de donner aux joueurs la possibilité de jouer différemment à la bataille navale, notamment de jouer en trois dimensions.

Pour cela, nous avons réfléchi puis développé un modèle capable de s'adapter aux besoins des joueurs. En effet, celui-ci peut décider de toutes les tailles, longueur, largeur ou encore profondeur, ainsi qu'un nombre de dimensions quelconques.

Le plateau de jeu permet donc de jouer en une dimension (soit sur la longueur ou soit sur la largeur), en deux dimensions (longueur et largeur), en trois dimensions nous ajoutons la profondeur pour enfin pouvoir jouer en plus de trois dimensions.

1.1.1 Définition récursive des Boards

Nous avons uns structure donnée qui se définit récursivement de la manière suivante :

— Une case d'un Board est un Board à zéro dimensions.

— Un Board à N dimensions, dont les tailles dans chaque dimension sont (TN, TN-1, ... , T1) où les Ti sont les tailles, est composé de TN Boards à N - 1 dimensions, dont les tailles dans chaque dimension sont (TN-1, ... , T1).

Notez bien que les Boards à N - 1 dimensions ont exactement les même tailles pour chaque dimension. Ainsi, un Board à deux dimensions sera toujours représentable comme un rectangle, un Board à trois dimensions peut-être visualisé comme un pavé droit et ceux des dimensions supérieures suivent le même principe, bien qu'ils soient plus difficile à représenter.

On peut aussi voir le Board comme un arbre particulier avec les propriétés suivantes :

— La distance (plus court chemin de la racine à une feuille) et la hauteur (plus long chemin de la racine à une feuille) de l'arbre sont égales.

— Tous les nœuds d'un même niveau ont le même nombre de descendants.

Cette structure récursive nous permet d'instancier un Board de n'importe quel nombre de dimensions en lui passant tout simplement un tableau d'entier classique contenant les tailles dans chaque dimension.

1.1.2 Remplissage des Boards

Pour remplir un tableau classique en programmation, on utilise généralement autant de boucles imbriquées qu'il y a de dimensions au tableau. Dans le cas du Board, ce n'est pas possible à notre connaissance, puisque le nombre de dimensions est inconnu à la compilation et qu'il faudrait un nombre variable de boucles imbriquées dans le code.

Nous avons donc créé un itérateur de Boards à nombre de dimensions zéro, implémenté dans la classe interne de Board, DimZeroIterator qui donne un par un les Boards à zéro dimensions qui sont les cases du Board.

Cet itérateur est également construit de manière récursive. Il contient les DimZeroIterator correspondants aux Boards fils du Board qu'il itère et les méthodes next() et hasNext() appelent leurs homologues sur l'itérateur fils en cours de parcours.

Si on voit le Board comme un arbre, cet itérateur donne les cases dans l'ordre où un parcours en profondeur donnerait les feuilles.

1.1.3 Accès aux cases

L'accès en lecture et en écriture d'une case en particulier se fait grâce aux objets Coordinates. Ces objets sont techniquement des tableaux d'entiers non mutables, dont on peut tester l'égalité avec la méthode equals() redéfinie. L'accès à une case de ce tableau se fait avec la méthode get(int index), à la façon des listes, mais il est itérable comme un tableau classique.

Lors d'un appel à getItem() ou setItem() sur un Board, chaque composante de la Coordinates fournie en paramètre sert à déterminer sur quel Board de dimension inférieure il faut effectuer l'appel récursif.

Les objets Coordinates sont également utilisés pour instancier les Boards en fournissant le nombre de dimensions et la taille pour chaque dimension.

1.1.4 Détails de l'implantation

Les algorithmes d'accès aux cases de Board sont récursifs terminaux. Structures des objets :

```
structure Board :
    champ Object : item;
    champ Liste de Board : boards;
fin structure

structure Coordinates :
    champ Tableau d'entiers : coord;
    champ Entier : length;
fin structure
```

1.1.4.1 Algorithmes de Coordinates

```
algo Coordinates creerCoordinates(Tableau d'entier t) :
    coord <- copie de t;
length <- longueur de t;
fin algo

algo Entier get(Coordinates c, Entier index) :
    renvoyer c.coord[index];
fin algo
```

1.1.4.2 Algorithmes de Board

```
algo Board creerBoard(Coordinates c) :
    renvoyer creerBoard(c, 0);
fin algo

algo Board creerBoard(Coordinates c, entier index) :
    Board b;
    b.item <- null;
    si index = c.length alors
        renvoyer b;
    finsi
    b.boards <- nouvelle Liste de Board;
    pour entier k de 0 à c.get(index) faire
        b.boards[k] <- creerBoard(c, index + 1);
    fin pour
    renvoyer b;
```

```
fin algo

algo Object getItem(Board b, Coordinates c) :
    renvoyer getItem(b, c, 0);
fin algo


algo Object getItem(Board b, Coordinates c, Entier index) :
    si index = c.length alors
        renvoyer b.item;
    finsi
    renvoyer getItem(b.boards.get(c.get(index)), c, index + 1);
fin algo

algo voidsetItem(Board b, Coordinates c, Object item) :
    setItem(b, c, 0, item);
fin algo

algo void setItem(Board b, Coordinates c, Entier index, Object item) :
    si index = c.length alors
        b.item <- item;
    finsi
    setItem(b.boards.get(c.get(index)), c, index + 1, item);
fin algo
```

1.2 Les bateaux en dimensions variables

Les navires sont représentés par la classe Ship. Ils ont une longueur, un nom et contiennent les coordonnées des cases du Board qu'ils occupent lorsqu'ils sont placés.

Dans une bataille navale classique, chaque navire est disposé verticalement ou horizontalement. Dans notre modèle à nombre variable de dimensions, il faut généraliser cette règle pour qu'elle fonctionne dans tous les cas.

1.2.1 Définition d'un bon placement

Pour être bien placés, les Ship doivent respecter les conditions suivantes :

— Bon alignement : les cases occupées par un Ship doivent avoir des coordonnées dont toutes les composantes sauf une sont égales, et les composantes qui ne le sont pas doivent se suivre.
Exemple : Dans un Board en quatre dimensions, un porte-avion (de longueur 5) occupe les cases (0,2,1,3), (0,2,2,3), (0,2,3,3), (0,2,4,3),

(0,2,5,3). Ce placement respecte la condition de bon alignement car les premières, deuxièmes et quatrièmes composantes des coordonnés sont respectivement égales entre-elles, et que les troisièmes composantes se suivent.

— Non dépassement : les cases occupées par un Ship sont celles contenues dans un Board.
Exemple : Dans un Board de dimensions (5,5,5,5), le porte-avion placé précédemment dépasse par sa dernière coordonnée, dont la troisième composante n'est pas comprise entre 0 et 4.

— Bonne longueur : un Ship doit occuper exactement le même nombre de cases que sa longueur.

— Non-Chevauchement : une case ne peut être occupée que par un seul Ship.

1.2.2 Tir sur les Ships

Les tirs sur les Ships se font via le Board sur lequel ils sont disposés. Pour cela, le type générique Board est paramétré soit par le type Case, soit par le type State.

Les Board<Case> de chaque joueur sont ceux sur lesquels ils placent leurs navires, et les Board<State> représentent la grille de tir des joueurs, là où ils peuvent voir où ils ont tirés et pour quel résultat.

State est une énumération représentant le statut de tir de chaque case d'un Board<State>. Il existe quatre états possibles :
— NOT_AIMED : cette case n'a pas été visée par un tir.
— MISSED : il y a eu un tir sur cette case, mais il n'y avait pas de Ship adverse sur la case correspondante du Board<Case> du joueur adverse.
— HIT : Il y a eu un tir sur cette case, et un Ship adverse a été touché.
— SUNK : Même chose que HIT, avec en plus l'information que le Ship touché a été coulé par ce tir.

Les objets Case ont pour rôle de stocker deux informations : la présence d'un Ship et si la case a déjà reçu un tir. Ces deux informations suffisent à faire la distinction entre les états NOT_AIMED, MISSED et HIT/SUNK.

Un objet Case sert également à relayer un tir jusqu'au Ship qui l'occupe, s'il y en a un, et à s'assurer qu'on ne tire pas deux fois au même endroit.

Cette condition rend non-nécessaire le stockage dans les Ship des endroits précis où ils ont été touchés : un Ship qui a été touché un nombre de fois égal à sa longueur a forcément été touché sur toutes les cases qu'il occupe, et on sait ainsi qu'il a été coulé.

1.3 Les intelligences artificielles

Il nous à été demandé de pouvoir jouer à la bataille navale en local. De ce fait, nous avons due développer une Intelligence Artificiel (IA), car jouer à deux sur un même ordinateur présente de nombreux risques de triches.
Notre IA se compose de 3 niveaux de difficultés différents, allant de facile à difficile.
Nous allons donc vous expliquez le fonctionnement de chacune de ces 3 IA.

1.3.1 IA Facile

L'IA facile fut la première intelligence artificielle programmée. Son cheminement est simple, elle consiste simplement à choisir aléatoirement une case, en vérifiant bien entendu que celle-ci n'est pas déjà était tirée au préalable, puis de tirer dessus, qu'importe l'avancement de la partie.
Le choix de l'aléatoire est fait de la façon suivante :

```
Fonction EasyAdvisor :
Entrée : Board un tableau de state contenant les resultats de tous les tirs.
         Dim le tableau des tailles des dimension de Board.
Sortie : Coord une Coordonnée d'une case non touchée.
Début
    Faire {
        Entier distance = Alea(0, taille(Board)) ;
        Entier longueur = taille(Board) ;
        Pour (n allant de 0 à taille(Dim)) {
            longueur = longueur / Dim[taille(dim) - 1 - n) ;
            Ajout(Coord, taille(dim) - 1 - n, distance / longueur) ;
            Distant = distance % longueur
        }
    } tant que (Board[Coord] != NON_CIBLÉ) ;
    Retourner Coord ;
Fin
```

1.3.2 IA Moyen

L'IA moyen, deuxième IA développée, fonctionne de manière similaire à l'IA facile, à ceci près qu'une fois un bateau touché l'IA passe en mode "chasse".
Le mode chasse indique à l'IA de tirer sur les quatres directions(nord, sud, est et ouest) entourant la case touchée et, si une nouvelle case est touchée, repart de cette dernière pour ainsi continuer jusqu'à avoir coulé le bateau.
Elle repart alors dans sa routine de ciblage.
L'algorithme est présenté ci-dessous :

```
Procedure MediumAdvisor :
    Entrée : Board un tableau de state contenant les résultats
      de tous les tirs.
             Dim le tableau des tailles des dimension de Board.
             Coe une liste de case.
    Sortie : Coord une Coordonnée d'une case non touchée.

    Début
        Si estVide(coe) {
            Coodinate c = EasyAdvisor(Board, Dim) ;
            SI (Tir(enemy, c) != MANQUÉ) {
                AjouterCasesAdjacentes(coe, coord) ;
            }
            Retourner Coord ;
        }
        Sinon {
            Si (Board[Coe[0]] == NON_CIBLÉ) {
                Coord = coe[0] ;
                Retirer(coe, 0) ;
                Retourner Coord ;
            }
            MediumAdvisor(Board, Dim, Coe);
        }
    Fin
```

1.3.3 IA Difficile

L'IA difficile, troisième IA développée, fonctionne de manière similaire à l'IA normale, c'est à dire qu'elle effectue sa routine de "ciblage" jusqu'à toucher un bateau, puis passe en mode "chasse" une fois un bateau touché. La différence avec l'IA normale est que cette IA ne tire pas de case adjacente

l'une à l'autre. En effet, suivant la taille minimal du bateau ennemi existant au début de partie, afin qu'elle ne tire que toute les tailles du bateau minimal. Par exemple, dans le cas ou le plus petit bateau est un batea de 2, l'IA va donc écarter ses tirs de 2 cases d'écart à chaque fois.

```
Procedure MediumAdvisor :
Entrée : Board un tableau de state contenant les resultats de tous les ti
         Dim le tableau des tailles des dimension de Board.
         Coe une liste de case.
         Minsize la taille du plus petit bateau present sur la grille
Sortie : Coord une Coordonnée d'une case non touchée.

Si estVide(coe) {
    Faire {
        Entier distance = Alea(0, taille(Board)) ;
        Entier longueur = taille(Board) ;
        Entier mod = 0 ;
        Pour (n allant de 0 à taille(Dim)) {
            Mod = 0 ;
            longueur = longueur / Dim[taille(dim) - 1 - n) ;
            Ajout(Coord, taille(dim) - 1 - n, distance / longueur) ;
            Distant = distance % longueur
            Mod = (mod + distance /longueur) % minsize;
        }
    } tant que (mod != 0 || Board[Coord] != NOTAIMED) ;

    SI (Tir(enemy, coord) != MISSED) {
        AjouterCasesAdjacentes(coe, coord) ;
    }
    Retourner Coord ;
}
Sinon {
    Si (Board[Coe[0]] == NOTAIMED {
        Coord = coe[0] ;
        Retirer(coe, 0) ;Retourner Coord ;
    }
}
MediumAdvisor(Board, Dim, Coe) ;
Fin Procedure
```

2 Le modèle réseau

2.1 Protocoles

2.1.1 Protocole Réseau

Le protocole réseau [1] utilisé pour que deux joueurs puissent se connecter et jouer en réseau est le protocole TCP [2] qui est un protocole en mode connecté mais également fiable quant à la connexion entre deux machines et pour la réception des paquets envoyés sur le réseau.

Nous avons préféré le protocole TCP au protocole UDP ([3] car, premièrement, son implémentation est bien plus aisée. En effet, nous n'avons pas à traiter certains cas d'erreurs, notamment dans le cas d'un paquet qui soit mal formé ou imcomplet car TCP assure que celui-ci sera totalement prêt avant de l'envoyer.
Deuxièmement, malgré le fait que le protocole UDP soit plus rapide dans l'envoi des données, utiliser le protocole TCP dans notre cadre n'a pas d'impact significatif puisque, comme il sera exposé un peu plus tard, chaque joueur doit attendre la réception de données avant de continuer son traitement. Ainsi, il n'est pas, en pratique, possible de recevoir des données qui ne sont pas attendues à des moments précis.

2.1.2 Protocole de l'application

L'application doit suivre un certain protocole pour que les deux joueurs puissent communiquer correctement et sans problèmes majeurs. Il a donc été décidé de suivre un type de protocole classique qui est celui du question/réponse. Néanmoins, nous n'avons pas choisi d'implémenter de serveur d'hébergement centralisant toutes les parties et dirigeant toutes les actions des clients, nous avons opté pour un échange de type pair à pair, c'est à dire que les deux joueurs doivent jouer les deux rôles : le processus serveur et le processus client, chacun à des temps déterminés et non mutables.
Nous expliquerons dans cette partie la connexion entre les deux joueurs ainsi que comment se passe l'échange de données.

1. Tous les termes demandant explications sont disponibles dans le lexique
2. Transmission Control Protocol : https ://tools.ietf.org/html/rfc793
3. User Datagram Protocol : https ://tools.ietf.org/html/rfc768

2.1.2.1 Connexion des joueurs

Comme dit plus haut, chaque joueur doit jouer deux rôles bien distincts, celui du processus serveur comme celui du processus client. Dans ce cas-ci, il est donc bon de noter l'ordre des différentes opérations effectuées entre un joueur qui va accueillir la partie et celui qui va en être l'invité, nous allons donc expliciter quelles actions font parties du processus client et lesquelles font parties du processus serveur.

Les deux joueurs vont, en premier lieu, créer et lancer leur serveur respectif, en spécifiant l'adresse de la machine en format IPv4, un port d'écoute (nous vérifions qu'il soit compris entre 1024 et 65535), ainsi qu'un nombre de connexions autorisées qu'on nommera "backlog".

Ensuite, le joueur hôte met son serveur en attente d'une demande de connexion via sa socket d'écoute (il est donc considéré en tant que processus serveur) et, de son côté, le joueur invité (considéré comme le processus client à ce moment) va se connecter à l'adresse du joueur hôte. Cela aura pour effet, côté processus serveur, de créer une socket de service par laquelle le joueur passera pour communiquer avec le joueur invité.

À cette étape, il reste à faire la connexion inverse, c'est à dire que le joueur hôte (qui est maintenant le processus client) doit faire une demande de connexion à l'adresse du serveur du joueur invité (devenu le processus serveur) et, une fois la connexion acceptée par la socket d'écoute du processus serveur, une nouvelle socket de service est créée pour que le joueur invité puisse communiqué avec le joueur hôte.

Suite à ces différentes étapes, la connexion entre le joueur hôte et le joueur invité est établie, les échanges de données peuvent donc être effectuée.

2.1.2.2 Échanges des données

Mais avant de pouvoir commencer la partie et à échanger des données, chaque joueur doit placer ses bateaux puis attend que l'adversaire finisse de placer les siens. Pour nos besoins, nous n'avons pas décidé de laisser l'application choisir au hasard qui commence une partie : ce sera toujours le joueur hôte.

De ce fait, le joueur hôte devient donc le processus client qui doit alors choisir puis envoyer les coordonnées de son tir, ces dernières étant réceptionnées par le processus serveur, i.e le joueur invité, qui est en

attente de ces coordonnées et rien d'autre. En effet, nous verrons plus tard comment ces données sont formatées pour que notre protocole fonctionne correctement (voir 2.1.2.3).

La prochaine étape consiste, pour le joueur invité maintenant devenu le processus client, d'envoyer le résultat du tir qui est un état de la case visée. Cet état peut être de trois natures : "touché", "coulé" ou bien "manqué". Le joueur hôte, quant à lui, attend l'arrivée de ce résultat en tant que processus serveur.

Enfin, les joueurs mettent à jour leur vue respective pour finalement changer le tour de jeu, c'est à dire que le joueur hôte passe la main au joueur invité.

Ce schéma, se répète quand c'est au tour du joueur invité de jouer, il suffit simplement d'inverser les rôles pour obtenir un tour de jeu complet et ce jusqu'à la victoire d'un des joueurs qui fermeront correctement ensuite proprement les différentes sockets mises en jeu.

2.1.2.3 Format des données

Dans le cadre de notre protocole, il a fallu décidé d'un certain format que les données doivent respectées pour pouvoir être acceptées et ainsi traitées. De ce fait, tout paquet est formé de cette manière afin de faciliter l'extraction des données utiles au protocole :

"Objet :Valeur"

Ces deux composantes définissent donc le format des données envoyées. Nous allons donc maintenant définir quels sont les types d'objets possible ainsi que leurs valeurs associées.

1. "Coordinates :X,X" ⇒ Ce format permet d'envoyer les coordonnées d'un tir, la valeur est une suite de nombres séparés par des virgules représentant la coordonnée sur un axe donné. Par exemple, si nous recevons "Coordinates :1,1", cela signifie que nous voulons tirer sur la case situé à 1 sur l'axe des X et en 1 sur l'axe des Y.
2. "State :X" ⇒ Ce format permet d'envoyer l'état de la case qui a été visé. Les valeurs possibles ont déjà été définies dans la partie 2.1.2.2. Par exemple, en recevant "State :HIT", cela signifie que la case visé à été touché.
3. "String :X" ⇒ Un objet String est une chaîne de caractères, la valeur pouvant être n'importe quelle chaîne. Grâce à ce format, nous pouvons

envoyer des messages simples, comme le fait d'avoir gagné la partie ou bien un message d'erreur que nous affichons à l'utilisateur.

2.2 Gestion des erreurs

La gestion des erreurs est faites sur plusieurs niveaux d'abstraction dans notre projet. En effet, le premier niveau d'abstraction est le fait d'utiliser le protocole TCP/IP, car celui-ci, en opposition au protocole UDP, en plus de recevoir le paquet fait une vérification afin de valider le paquet avant de le transmettre. De ce fait, une partie des vérifications est déjà effectué par ce biais.

Le second niveau d'abstraction est lié au langage que nous avons choisi. En effet, Java implémente les protocoles TCP et UDP. Hors, Java effectue aussi des vérifications quand à la validité des données et permet facilement de vérifier si le paquet reçu est valide ou non, par le biais des Exceptions présente dans l'API de base.

Au final, nous avons implanté notre propre protocole de validité des données, afin d'être sur de ce que nous recevons. Comme vous avez pu le lire précédemment, chacune de nos données envoyées possède un format propre a notre protocole et si, lors d'un envoie de donnée, nous n'avons pas le bon type de données, alors nous gérons le problème en indiquant qu'un problème est survenue lors de l'envoie des données, ce qui n'est pas normal et qui indique un problème au niveau du programme. De ce fait, nous fermons donc proprement l'application afin d'éviter tout problèmes lors d'une exécution prolongée du logiciel.

3 Interface Homme Machine

4 Autres

4.1 Technologies utilisés

Les technologies que nous avons utilisé durant le projet sont les suivantes.
La première technologie utilisée fut le langage Java. En effet, nous avons choisit de prendre le Java, car notre client nous l'a conseillé, en particulier pour ce qui est la gestion de l'IHM. De plus, étant un langage que nous maîtrisons tous plus ou moins, nous avions ainsi davantage de facilité à le manier et donc à développer le projet voulu.
La seconde technologie fut l'utilisation de GIT, un logiciel de gestion de version de code source. En effet, via GIT, nous pouvions aisément travailler sur des partie du code différentes sans interférer avec les autres. De plus, cela nous permettait, en cas de soucis, de récupérer le code ultérieur pour repartir dessus si des améliorations, ou des modifications apportées au code courant modifièrent trop le fonctionnement de l'application, ou pire, créèrent des bugs non réparable.
La troisième technologie que certains d'entre nous apprirent fut LaTeX, car part le biais de LaTeX, nous avons pu rendre un rapport de projet propre, et surtout facilement modifiable en cas de besoin.

4.2 Idées d'améliorations

Voici une liste d'amélioration possible qui pourrait être implanter sur le projet afin de le rendre encore plus complet.
La première amélioration serait d'ajouter une IA très difficile, qui fonctionnerait de la même façon que l'IA difficile, avec cependant une variante qui consisterait à adapter les coups qu'elle tire en fonction des bateaux touchées et des cases restantes sur la grille.
La seconde amélioration possible serait le fait qu'un thread utilisateur soit lancer lorsque nos sockets de service sont en attendent. En effet, lorsque celles-ci sont en attente, EDT (Event Dispatch Thread) le thread gérant toute l'IHM se retrouve aussi bloquer due a l'attente de réception de données.
Une troisième amélioration potentielle serait d'améliorer l'IHM afin que celle-ci affiche un beau menu lors de l'ouverture du logiciel, afin que les utilisateurs ait vraiment la sensation de jouer à un jeu.
Nous pourrions aussi ajouter le Drag'n'Drop, appelé aussi "Glissé-Déposé", pour placer plus facilement nos bateaux sur la grille, lors du début d'une partie. En effet, nous plaçons actuellement nos bateaux par le biais d'un clique plaçant l'avant du bateau, puis l'arrière du bateau par un second clique, et le fait de le faire par Drag'n'Drop serais plus intuitifs pour certaines personnes.

Il serait aussi possible d'offrir la liberté d'ajouter plus de bateaux, ou de modifier ceux existant, par exemple en modifiant leur taille, ou encore en jouant avec 7 bateaux au lieu de 5 initialement dans les règles officiels de la bataille navale.
Une autre amélioration développable serait de pouvoir changer les règles ou d'ajouter des règles durant une partie. Par exemple, nous pourrions utiliser ajouter un chronomètre lors d'un tour de jeu, afin que chaque joueur doivent jouer rapidement, sous peine de voir son tour être passé sans avoir eu le temps de jouer. Nous pourrions aussi ajouter un chat pour que les joueurs puissent communiquer entre eux, afin qu'ils puissent par exemple discuter entre eux, ou encore se charrier gentillement lors d'un coulé par exemple.
Comme nous l'avons vu, ce projet ne possède guère de limite si ce n'est notre imagination.

4.3 Problèmes rencontrés

De nombreux problèmes ont été rencontrés au cours du projet, les plus important d'entre eux sont le fait que nous avons manquée de temps pour le projet, ainsi que le manque de salle informatique pour travailler efficacement, notamment le vendredi après midi, ou aucune salle informatique n'était libre pour nous permettre de bosser tous ensemble.
Un autre problème que nous avons rencontré est le fait que nous n'avions pas de redirection de port, ne permettant pas de tester facilement la partie réseau du projet, car il faut absolument que chacun possède son point d'accès, pour pouvoir le modifier, ce qui n'était pas forcément le cas.
Nous avons rencontré aussi des problèmes organisationnelles et logistiques vis à vis du travail de chacun au début, en particulier sur le début du projet.
Au niveau du développement, nous avons rencontré un problème au niveau de l'uniformisation du code. En effet, nous avions convenu initialement de coder en anglais, langue universelle de l'informatique. Hors, plusieurs fois nous nous somme rencontrer a des noms de variables ou encore de méthodes écrite en français, faisant ainsi une grosse différence entre chaque développeur présent sur le projet.
Toujours au niveau du code, nous avons eu de grosse difficultés à manipuler les threads, en particulier le lien entre EDT et les thread utilisateurs, ce qui explique que cette gestion n'ait pas été réalisé et implanter dans le code, malgré de nombreuses heures passées dessus.
Pour terminer, une dernière difficulté s'est aussi présenté à nous, en effet, l'appréhension et la visualisation du modèle permettant des grilles à dimension variable et inconnu a été assez difficile a comprendre, car au delà de la troisième dimension, nous étions régulièrement perdu.

4.4 Compétences acquises

Au cours du projet, nous avons acquit nombres de connaissances tels que par exemple l'initiation a GIT, un logiciel de versionning de code, permettant ainsi un travail efficace en groupe, car chacun peut facilement travailler sur un module du projet sans risquer de "casser" le code fonctionnel.
Nous avons aussi acquit de nombreuses connaissances sur le réseau, en particulier sur le protocole TCP/IP qui est un standard d'internet, ainsi que le fonctionnement d'un protocole question/réponse et pour finir l'utilisation des sockets par le biais du langage Java.
Pour finir, nous avons acquit la capacité de gérer et de se répartir le travail efficacement lorsqu'on travail en groupe.

Conclusion

Pour conclure, avec LaTeX on obtient un rendu impeccable mais il faut s'investir pour le prendre en main.

A Lexique

— Protocole réseau : Un protocole est une méthode standard qui permet la communication entre des processus (s'exécutant éventuellement sur différentes machines), c'est-à-dire un ensemble de règles et de procédures à respecter pour émettre et recevoir des données sur un réseau. Il en existe plusieurs selon ce que l'on attend de la communication. Certains protocoles seront par exemple spécialisés dans l'échange de fichiers (le FTP), d'autres pourront servir à gérer simplement l'état de la transmission et des erreurs (c'est le cas du protocole ICMP), ...

— TCP : Acronyme de Transmission Control Protocol, le protocole TCP/IP est le protocole standard utilsié sur internet, pour la liaison entre deux ordinateurs. Le protocoel TCP vérifie la validité des paquets après leur réception afin d'être sur de la validité de celle-ci. Le protocole TCP est située sur la couche 4 (couche de transport) du modèle OSI.

— UDP : Acronyme User Datagram Protocol, le protocole UDP est un des protocoles standards utilisé sur internet. La différence avec TCP est que les paquets sont reçus sous forme de datagramme qui doit être vérifier pour valider la qualité du paquet reçu. Le protocole UDP est très utilisé, notamment, dans le cadre du jeu en ligne, ou encore le streaming, car la perte de paquet influe peu sur la quantité reçus. Le protocole UDP est située sur la couche 4 (couche de transport) du modèle OSI, au même titre que le protocole TCP.

— Socket : Une socket est une interface de connexion bidirectionnelle permettant l'échange de données entre deux processus (distants ou non).

— Socket de Berkeley : Les sockets de Berkeley, sont un ensemble normalisés de fonctions de communications lancé par l'université de Berkeley au début des années 1980. De nos jours, elle est la norme utilisé par quasiment l'ensemble des langages de développement (C, Java, Python, ...).

— Pair à pair : Le pair à pair (ou peer-to-peer en anglais), est un modèle de réseau permettant à deux machines de discuter d'égale à égale. Dans les faits, cela s'explique par le fait qu'une machine se connecte a une autre machine et inversement afin que celles-ci puissent s'échanger des informations sans passer par un serveur distant.

B Webographie

— Fonctionnement des intelligences artificielles : `http://www.datagenetics.com/blog/december32011/index.html`

— API Java : `https://docs.oracle.com/javase/7/docs/api/`

— Règles de la bataille navale : `http://www.regles-de-jeux.com/regle-de-la-bataille-navale/`

C Figures