Bordeaux INP
ENSC

Informatique

Raga - Justifications techniques

Sommaire

1. Introduction

Durant ce projet informatique, nous avons voulu profiter de la liberté accordée à notre groupe afin d'expérimenter de nouvelles possibilitées qu'offre le c# notamment la transmission de données via le réseau local; ainsi que la création d'interface graphique grâce au winform. De fait, ce projet a pour objectif la création d'un mini jeu, pouvant être joué à plusieurs (2 joueurs et +) en réseau.

Ce document a pour but d'expliquer la structure et l'organisation du code, chaque fonction est déjà décrite dans les fichiers .cs et ne seront sont pas ici.

1.1. Référence

Ce jeu reprend le principe du jeu Agar.io dans le quel le joueur controle une boule (représentés par des disques colorés) suivant le curseur de la souris du joueur. Son objectif et d'absorber d'autres boules plus petites (autre joueurs ou boules inertes) afin d'augmenter sa taille. Pour absorber une autre boule, il suffit au joueur d'être plus grand que celle ci et d'être positionnée au dessus de cette dernière.

Figure 1 - Image illustrant le jeu Agar.io
Les boules de grandes tailles représentent des joueurs, celles de petites tailles des boules inertes.

1.2. Projet Raga

Le jeu de ce projet est nommé Raga, et le fonctionnement est similaire à celui d'Agar.io. En effet, dans celui-ci, le joueur contrôle un carré (ou boule de norme infinie), suivant le curseur de la souris du joueur. Il peut absorber d'autres carrés (joueurs ou carrés inertes) dont la distance séparants leurs deux centres est inférieur au rayon du joueur. Il est également nécessaire que le joueur possède une taille supérieure à la cible.

A la différence du jeu initiale, outre le fait que le joueur contrôle un carré et non une boule, dans Raga le joueur doit choisir une couleur d'équipe avant de se connecter, puis rejoindra cette équipe lors du jeu. La couleur de l'équipe choisi sera celle du carré contrôlé par

le joueur. Les joueurs ne peuvent pas absorber des joueurs de la même équipe, même s'il sont plus grand qu'eux.

2. Organisation

Afin de réaliser ce jeu, nous avons décomposé le projet en deux "sous-projets" que nous appellerons "Client" et "Serveur", le premier représentant l'exemplaire qu'un joueur voulant jouer reçoit, et le dernier celui qu'un hôte reçoit pour créer une partie. De fait, nous pouvons déjà voir que ces deux sous-projets ont des objectifs différents :

- Le **serveur** sert à héberger une partie, écouter les différents clients connectés à lui et leur transmettre les instructions à réaliser pour le jeu. Nous avons tout de même rajouté la possibilité à l'hôte hébergeant le serveur de pouvoir exécuter certaines commandes durant une partie, afin de régler quelques paramètres, modérer, limiter l'accès des joueurs...
- Quand au **client**, il permet de générer un affichage graphique des données reçu du serveur, permettre d'écouter les actions du joueur, et sous certaines conditions, les envoyer au serveur. Le client gère également le son, l'organisation du classement ainsi que l'enregistrement des anciennes données rentrées pour la connexion à un serveur. (Et les réinsère dans zones de texte du questionnaire afin de ne pas avoir à re-rentrer toutes les entrées)

Ainsi, dans l'archive de ce projet, le dossier intitulé "ServeurRaga" correspond au serveur (**seul** exemplaire à remettre à un hôte voulant héberger une partie) et le dossier "ClientRaga" correspond au client (**seul** exemplaire à remettre un joueur voulant rejoindre une partie hébergée par un hôte). Les dossiers sont donc indépendants et peuvent être utilisés seuls.

En revanche, ces deux projets ont pour objectif de communiquer l'un avec l'autre lors de l'exécution. De fait, nous ne pouvons pas vous présenter les projets en parallèle sans avoir vu comment ils communiquent entre eux car certains éléments ne sont compréhensible qu'avec une vue d'ensemble.

3. Communication

3.1. Général

Comme vu précédemment, les deux projets ont pour objectif de communiquer entre eux, même s'ils ne sont pas si le même pc. Pour cela, nous utilisons le namespace *System.Net.Sockets* et plus particulièrement la class "UdpClient" afin de créer un client UDP permettant de recevoir/émettre un message à un autre UdpClient.

3.1.1. Nature d'un message

Un message envoyé peut contenir plusieurs types d'informations : des position, parfois des tailles de rayon, ou des pseudo; et une nature différente : un déplacement, une absorption... De fait, nous avons eu à trouver un système permettant d'identifier la nature de chaque message. Or nous ne pouvons qu'envoyer des tableaux de Byte (*Byte[]*). De fait, nous avons utilisé le premier élément du tableau du message envoyé/reçu pour indiquer la nature du message.

De fait, le premier élément d'un message reçu (*Byte[] messageRecu*) est un Byte indique la nature du message ainsi que l'organisation et le type de contenu du reste de message. De fait, nous pouvons extraire les informations suivantes facilement.

3.1.2. Composition d'un message

Comme nous ne pouvons qu'envoyer des tableau de Byte, nous avons mis en place une méthode permettant de transmettre plusieurs information en un même message. Pour cela, il suffit de convertir les informations dans un string, chacune séparée par une virgule. (Il est ainsi à noter qu'un joueur ne peut pas se connecter avec un pseudo contenant une virgule).

Ensuite, il est nécessaire de transformer ce string en Byte[]. Pour cela nous utilisons les méthodes *Encoding.UTF8.GetByte(string message)* pour convertir un string en Byte[] et *Encoding.UTF8.GetString(Byte[] message)* pour convertir un Byte[] en string.

3.1.3. Récupérer les informations dans un message

Lors de la réception d'un message, sa nature indique les informations contenues dans ce dernier (voir 3.1.4). En revanche, il nous faut alors convertir le reste du message (*Byte[]*), en sting. Nous devons ainsi enlever le premier élément du tableau de Byte, avant de le convertir en string pour le manipuler. Pour récupérer un tableau à partir du i^e élément, nous utilisons la fonction *Outils.RecupererTableau(Byte[] tableau, int indice*).

Ensuite, il nous suffit d'extraire le texte entre chaque virgule afin d'avoir les informations. Pour repérer la position d'une virgule, nous utilisons la méthode *messageString.IndexOf(char caractere, int indice)*, où dans notre cas le caractère est ',' et l'indice est 0 au début, puis l'indice suivant la précédente virgule.

Pour récupérer le texte entre les deux position de deux virgules, nous utilisons la méthode *messageString.Substring(int start, int position),* où dans notre cas start est 0 ou l'indice de la précédente virgule +1; et position est l'indice de la virgule actuelle -1.

3.1.4. Liste des natures des messages

La nature d'un message (*définie en 3.1.1*) est indiqué grâce au premier élément d'un tableau de Byte (message envoyé/reçu entre le client et le serveur). Chaque nature de message correspond à un type d'action réalisé ou à réaliser.

La nature est donc un Byte, et peut valoir 256 valeurs différentes : de 0 à 255. A chaque nombre (ou chiffre) est associé une action, ceux dont aucune action n'est associé sont considéré comme "Message incompréhensible".

La liste suivante recense toutes les natures possibles actuellement des messages et le Byte qui leur est associé. Dans le cas futur d'un ajout de fonctionnalité, il faudra définir une nouvelle nature et modèle d'extraction des données dans le Client et le Serveur (et l'indique dans ce document).

• *Le premier élément du message, indiquant sa nature est noté i dans le tableau ci dessous. Le destinataire indique qui (et seulement lui) peut recevoir ce type de message.*

• *Il est à noter que la notion de "boule" ci-dessous évoquée représente les éléments carrés inertes pouvant être absorbés par les joueurs.*

i	Intitulé	Destinataire	Description et contenu du message
0	"I am Here"	Serveur	Message envoyé du client au serveur pour indiquer sa présence (qu'il n'est pas fermé).
1	Déplacement	Client & Serveur	Indique qu'un joueur s'est déplacé. - *Le client qui **envoie** ce message se déplace* - *Le serveur **transmet** le message à tous les autres clients (sauf l'émetteur) et **enregistre** les changements de position.* - *Les clients qui **reçoivent** ce message actualisent la nouvelle position du joueur.* •Message **envoyé** par le Client : "{0},{1}," où 0 : *int* x du client, 1 : *int* y du client. •Message envoyé par le Serveur, et **reçu** par les Clients : "{0},{1},{2}," où 0 : *int* x du joueur déplacé, 1 : *int* y du joueur déplacé, 2 : *string* Pseudo du joueur déplacé.

2	Absorption	Client & Serveur	Indique qu'un joueur a grossi. *- <u>Le client qui **envoie**</u> ce message a grandi* *- Le serveur **transmet** le message à tous les autres clients (sauf l'émetteur) et **enregistre** les changements de taille.* *- <u>Les clients qui **reçoivent**</u> ce message actualisent la nouvelle taille du joueur.* •Message **envoyé** par le Client : "{0}," où 0 : *int* Nouvelle taille du client, •Message envoyé par le Serveur, et **reçu** par les Clients : "{0},{1}," où 0 : *int* Nouvelle taille du joueur ayant grossi, 1 : *string* Pseudo du joueur déplacé.
3	Une boule est morte	Client & Serveur	Indique qu'une boule est morte. *- <u>Le client qui **envoie**</u> ce message a absorbé une boule* *- Le serveur **transmet** le message à tous les autres clients (sauf l'émetteur) et **détruit** la boule.* *- <u>Les clients qui **reçoivent**</u> ce message cachent la boule du terrain.* •Message **envoyé** par le Client, envoyé par le Serveur, et **reçu** par les Clients: "{0}," où 0 : *int* L'id de la boule absorbée.
4	Un joueur est mort	Client & Serveur	Indique qu'un joueur est mort. *- <u>Le client qui **envoie**</u> ce message a tué un joueur* *- Le serveur **transmet** le message à tous les autres clients (sauf l'émetteur) et **détruit** la le joueur.* *- <u>Les clients qui **reçoivent**</u> ce message suppriment le joueur du terrain.* •Message **envoyé** par le Client, envoyé par le Serveur, et **reçu** par les Clients: "{0}," où 0 : *string* Le pseudo du joueur mort.

5	Une boule est créée	Client	Indique qu'une boule doit être créée sur le terrain. - *<u>Le serveur **envoie**</u> ce message,* - *<u>Les clients qui **reçoivent**</u> ce message créent une boule avec les informations du message sur le terrain.* •Message **envoyé** par le Serveur,et **reçu** par les Clients: "{0},{1},{2},{3}," où 0 : *int* L'id de la boule à créer 1 : *int* La position x de la boule à créer 2 : *int* La position y de la boule à créer 3 : *int* L'indice de couleur de la boule
6	Un autre joueur est créé	Client	Indique qu'un joueur doit être créée sur le terrain. - *<u>Le serveur **envoie**</u> ce message,* - *<u>Les clients qui **reçoivent**</u> ce message créent un joueur avec les informations du message sur le terrain.* •Message **envoyé** par le Serveur,et **reçu** par les Clients: "{0},{1},{2},{3},{4}," où 0 : *int* La position x du joueur à créer 1 : *int* La position y du joueur à créer 2 : *int* La taille du joueur à créer 3 : *int* L'indice de couleur du joueur 4 : *string* Le pseudo du joueur à créer
7	Création du joueur client	Client	Ce message n'est reçu qu'une seule fois, lorsqu'un client se connecte et que le serveur lui indique toutes les informations lui étant propres. - *<u>Le serveur **envoie**</u> ce message,* - *<u>Le clients qui **reçoit**</u> ce message crée un joueur et le défini comme "JoueurClient".* •Message **envoyé** par le Serveur,et **reçu** par **un** Client: "{0},{1},{2},{3},{4}," où 0 : *int* La position x du joueur à créer 1 : *int* La position y du joueur à créer 2 : *int* La taille du joueur à créer 3 : *int* L'indice de couleur du joueur 4 : *string* Le pseudo du joueur à créer

253	Freeze/Unfreeze	Client	Indique que le joueur doit se freeze/s'unfreeze. - *Le serveur **envoie** ce message,* - *Les clients qui **reçoivent** ce message actualisent la variable 'isFreeze' ou geler ou dégeler le joueur Client.* *[Attention] Ce message ne n'a pas à être converti en string pour être compris : le second élément du Byte[] suffit.* •Message **envoyé** par le Serveur,et **reçu** par les Clients: [253][0 ou 1] où 0 : Le joueur doit se freeze. 1 : Le joueur doit se s'unfreeze.
254	Kill	Client	Indique que le client doit se fermer. - *Le serveur **envoie** ce message,* - *Le clients qui **reçoit** ce message se ferme.* *[Attention] Ce message ne n'a pas à être converti en string pour être compris : il n'a pas de contenu.* •Message **envoyé** par le Serveur,et **reçu** par les Clients: [254]
255	Demande Connexion	Serveur	Indique qu'un joueur demande de se connecter ou la réponse du serveur à cette demande. - *Le client qui **envoie** ce message veut se connecter* - *Le serveur **envoie** le message à ce même client avec sa réponse (oui / non).* - *Le client qui **reçoit** ce message affiche le client de jeu ou un message d'erreur.* •Message **envoyé** par le Client : [255][Byte c][“{0}"] où c : *Byte* L'indice de couleur souhaité, 0 : *string* Le pseudo souhaité. •Message envoyé par le Serveur, et **reçu** par le Client : [255][0 ou 1 ou 2] où 0 : La connexion est établie. 1 : Échec, le serveur est verrouillé, 2 : Échec, le pseudo est déjà prit.

3.2. Client UDP

Comme vu précédemment, lors de l'exécution des deux programmes, un "*UDPClient*" est créé (un chacun), puis il est stocké dans une variable *static public* afin de pouvoir envoyer des message partout dans le projet.

- Le client UDP du projet *ServeurRaga* crée un UDPClient avec un port initialement de "*12345*", pouvant être modifié via la variable static "*port*".
- Le client UDP du projet *ClientRaga* crée un UDPClient avec un port libre non modifiable ou déterminable. (Pas de besoin)

3.2.1. Client UDP du Client

De base, lors de l'exécution du projet *ClientRaga*, dans une WinForm, l'utilisateur peut rentrer l'ip et le port du serveur hôte. Le client essaie alors de se connecter à un potentiel serveur créé sur cette ip et écoutant ce port, S'il ne reçoit pas de réponse, le client considère ce potentiel serveur comme faux, et affiche un message d'erreur. S'il reçoit une réponse, elle peut être défavorable (le serveur est verrouillé, le pseudo est déjà utilisé) et le client peut tenter de se re-connecter en changeant les paramètres rentrées. Cette réponse peut également être favorable, et dans ce cas le client UDP du *ClientRaga* se connecte à celui du serveur, possédant les informations.

3.2.2. Client UDP du Serveur

De base, lors de l'exécution du projet *ServeurRaga*, après la création de l'UDPClient, une boucle d'écoute (*précisé par la suite dans 4.*) permet d'écouter tout message reçu par le client udp et d'agir en conséquence. De fait, tant que le programme est actif, l'UDPClient est à l'écoute d'un nouveau message, puis le traite. C'est le cas lors d'une demande de connexion qui, lorsqu'elle est favorable (*voir 3.2.1.*) indique au serveur qu'il conserver l'ip de l'émetteur (via la création d'un JoueurS) afin de pouvoir communiquer plus tard avec lui. De fait, l'UDPClient du serveur ne se connecte avec aucun autre UDPClient, mais utilise la liste des ip des Clients.

3.2.3. Thread d'écoute et traitement des messages

Après avoir créé leurs UDPClient, les deux projet lancent des "Boucles d'écoute", dans un thread parallèle permettant de recevoir des messages sur leur client UDP sans bloquer l'exécution du reste du programme. Ces boucles sont donc dans un Thread qui est démarré lors de la création du serveur ou de la création de la fenêtre de jeu.

Ensuite, la boucle écoute (dans un while(true)) tout message reçu sur le client UDP, puis execute la fonction *TraiterMessage(Byte[] messageRecu, IPEndPoint ipExpediteur)* qui, en fonction de la nature du message (voir 3.1.4.), détecté avec un switch; execute le reste des instructions appropriées. (voir le code pour plus d'info).

3.3. Méthodes à Utiliser pour la communication

Les envois de message au serveur/aux clients sont très fréquents, de fait des méthodes ont été implémentés afin de faciliter l'envoie de message.

3.3.1. Côté Client

Le client ne peut communiquer qu'avec le serveur et utilise la méthode: *Outils.SendToServer(int code, string message)* où le code est la nature du message (*voir 3.1.4.*) et le message le messager à transmettre (il sera transformé en Byte[] par la méthode).

3.3.2. Côté Serveur

Le serveur quant à lui possède plusieurs méthodes car il peut avoir à communiquer avec 1 (par exemple pour répondre à une demande de connexion), tous (par exemple pour geler ou tuer tous les joueurs), ou tous sauf un client (par exemple après un action comme un déplacement, pour ne pas dire au client qu'il s'est déplacé lui même).

- Communication avec un **seul** client: *SendToClient(int code, string message, IPEndPoint ip)* où le code est la nature du message (*voir 3.1.4.*) et le message le messager à transmettre (il sera transformé en Byte[] par la méthode).

- Communication avec **tous** les clients: *Outils.SendAll(int code, string message,)* le message est transformé en Byte[] dans la méthode. (Il est à noter que cette méthode possède des surchage avec simplement un Byte[] en argument. Cette dernière sera utilisée pour l'envoie de message sans contenu : KillMessage, Freeze…)

- Communication avec **tous les clients sauf un**: *SendAllExecpt(int code, string message, IPEndPoint ipException)* le message est déjà transformé en Byte[ ]. (Il est à noter que cette méthode possède des surchage avec simplement un Byte[] en argument. Cette dernière sera utilisée pour l'envoie de message sans contenu : KillMessage, Freeze…)

4. Serveur

4.1. Gestion des données

4.1.1 La class BouleS

Une instance de la class BouleS contient des informations permettant de définir une boule (id, position x et y, couleur).

• L'**id** d'une boule, elle stockée avec la variable *int id*. Chaque boule a une id unique.

• La **position** d'une boule, elle stockée avec les variables x et y (deux variable de type *int*), et sont contenues entre 5 et *tailleTerrain-5 pour x* et 5 et *tailleTerrain-5* pour y. Ces derniers sont renseignés par le serveur lors de la création d'une boule.

• La **couleur** d'une boule est entré dans son élément graphique et n'est pas stockée.

Pour l'attributs static de la class, la *List<BouleS> listeBoules* permet de contenir tous les boules(connus par le client).

4.1.2 La class JoueurS

Une instance de la class JoueurS contient l'ensemble des informations permettant de définir un joueur (ip, pseudo, position x et y, couleur, taille). Chaque instance de la class Joueur possède ses setter, getter ainsi que certains méthodes telles que SetPosition ou SetRayon qui modifient les informations ainsi que l'affichage. (Voir 5.2)

• L'**ip** d'un joueur, il est stocké avec la variable *IPEndPoint ip*.

• Le **iAmHereNumber** est un entier compris entre 0 et 4 et qui corresponds à un temps d'inactivité du Client, 4 étant la plus petite durée d'inactivité et 0 la plus grande.

• Le **pseudo** d'un joueur, il est stocké avec la variable *string pseudo*.Un pseudo ne peut pas contenir les caractères : '\"', ',' et ';'. Chaque joueur a un pseudo unique.

• La **position** d'un joueur, elle stockée avec les variables x et y (deux variable de type *int*), et sont contenues entre 0 et *maxXPosition pour x* et 0 et *maxYPosition* pour y. Ces deux variables sont des constantes définies en tête de la class.

• La **taille** d'un joueur est stockée dans la variable *int rayon* et a pour limite *rayonMax*, une constante également définie en tête de class.

• La **couleur** d'un joueur est stockée dans la variable *int couleur*. Elle peut varier de 0 à 6, chaque nombre correspond à une couleur de la liste Outils.Couleurs.

Pour les attributs static de la class, la *List<JoueurS> listeJoueurs* permet de contenir tous les joueurs (connus par le client); les autres attributs static joueurClient et kJoueurClient sont respectivement l'instance de la class Joueur correspondant au joueur du Client; et la constante k de vitesse de déplacement du joueurClient. (N'est présente qu'une fois par client car utilisé uniquement par le client).

4.2. Les commandes

La thread principal écoute la console. Lorsqu'une chaîne de caractère est rentrée un switch/case détermine la nature de la commande et l'exécute. (Voir la liste des commandes dans le fichier Program.cs)

5. ClientRaga

Dans cette partie, nous allons parler de tous éléments contenus dans la solution Client, et d'expliquer leurs applications.

5.1. Gestion des Données

Afin de stocker toutes les données des joueurs, des boules… Des class ont été créées afin de structurer les données : les class pouvant être instanciées sont la class *Joueur* et la class *Boule*; les class statiques sont la class *Camera*, la class *Classement*. (La class *Outils* quant à elle ne sert qu'à contenir les méthodes utilisés à plusieurs endroit du projet)

5.1.1. La class Joueur

Une instance de la class Joueur contient l'ensemble des informations permettant de définir un joueur (pseudo, position x et y, couleur, taille). Chaque instance de la class Joueur possède ses setter, getter ainsi que certains méthodes telles que SetPosition ou SetRayon qui modifient les informations ainsi que l'affichage. (Voir 5.2)

• Le **pseudo** d'un joueur, est stocké avec la variable *string pseudo*.Un pseudo ne peut pas contenir les caractères : '\"', ',' et ';'. Chaque joueur a un pseudo unique.

• La **position** d'un joueur, elle stockée avec les variables x et y (deux variable de type *int*), et sont contenues entre 0 et *maxXPosition pour x* et 0 et *maxYPosition* pour y. Ces deux variables sont des constantes définies en tête de la class.

• La **taille** d'un joueur est stockée dans la variable *int rayon* et a pour limite *rayonMax*, une constante également définie en tête de class.

• La **couleur** d'un joueur est stockée dans la variable *int couleur*. Elle peut varier de 0 à 6, chaque nombre correspond à une couleur de la liste Outils.Couleurs.

• L'élément **graphique** Winform d'un joueur est stockée dans la variable *CustomCircle customCircle*. (*Voir 5.2.1.*)

Pour les attributs static de la class, la *List<Joueur> listeJoueurs* permet de contenir tous les joueurs (connus par le client); les autres attributs static joueurClient et kJoueurClient sont respectivement l'instance de la class Joueur correspondant au joueur du Client; et la constante k de vitesse de déplacement du joueurClient. (N'est présente qu'une fois par client car utilisé uniquement par le client).

5.1.2. La class Boule

Une instance de la class Boule contient l'ensemble des informations permettant de définir une boule (id, position x et y, couleur, taille).

• L'**id** d'une boule, elle stockée avec la variable *int id*. Chaque boule a une id unique.

• La **position** d'une boule, elle stockée avec les variables x et y (deux variable de type *int*), et sont contenues entre 5 et *tailleTerrain-5 pour x* et 5 et *tailleTerrain-5* pour y. Ces derniers sont renseignés par le serveur lors de la création d'une boule.

• La **couleur** d'une boule est entré dans son élément graphique et n'est pas stockée.

• L'élément **graphique** Winform d'une boule est stockée dans la variable *CustomCircle customCircle*. (*Voir 5.2.3.*)

Pour les attributs static de la class, la *List<Boule> listeBoules* permet de contenir tous les boules(connus par le client), *List<Boule> listeAnciennesBoules* permet de contenir tous les boules anciennement présentes sur la Winform et mangées.

Cette dernière permet de limiter la création de control dans la fenêtre Winform, ainsi lorsqu'une boule est mangée, elle est simplement cachée (-2500 , -2500) et retiré de la liste des boules connus. Lorsqu'une nouvelle boule est créée, si la liste des anciennes boules n'est pas vide, alors elle est "recréée" (modifiée graphiquement et ré-introduite dans la liste des boules).

5.1.3. La class Camera

La Camera est une class static permettant de mettre de n'afficher qu'une partie de la map (trop grande pour être affichée en entier sur l'écran). Elle possède une position X allant de *-(la largeur de la fenetre / 2)* à *-(la largeur de la fenetre / 2)+maxXPosition*, et Y allant de *-(la hauteur de la fenetre / 2)* à *-(la hauteur de la fenetre / 2)+maxYPosition*. La Camera possède toutes les fonctions d'affichage.

5.1.4. La class Classement

La Camera est une class static permettant de stocker et d'afficher le classement en haut à gauche de l'écran. Le nombre d'équipe maximal est stockée dans la variable static *int nbEquipes*.

Le tableau *int[] classementEquipes* comporte à la fois les score et les position de chaque équipe. Il est de taille *2 * nbEquipes*, possède sur ses position paires (0, 2, …) les numéros des équipes (0 : bleu, 1: rouge…) et sur ses position impaires les score (correspondant à l'équipe dont le numéros est à l'indice d'avant).

5.2. Déplacement

5.2.1. Déplacement du joueur Client

Le déplacement d'un joueur se grâce au Timer du Winform. Toutes les 40ms, il compare la position en x et y du joueur et celle de la souris. Si la distance entre eux est de plus de ⅔ du rayon du joueur, alors le joueur est déplacé vers la souris (et non à la position de la souris).

Le déplacement du joueur n'est autorisé que si le joueur reste dans la zone de jeu, c'est à dire entre 0 et *maxXPosition pour x* et 0 et *maxYPosition* pour y.

5.2.2. Affichage

L'interface de jeu est géré par le client via la solution ClientRaga. Il est composé d'une winform FenetreJeu qui reçoit les données des autres joueurs et permet grâce à un Timer (System.Windows.Forms.Timer) d'actualiser l'affichage toutes les 40ms au travers de la fonction Timer_Tick(). Ainsi toutes les 40ms il faut tester si le joueur mange une boule ou un adversaire et donc grossit ou s'il s'est fait manger. Puis s'il est toujours en vie il faut redessiner sur l'écran la nouvelle position du joueur, des adversaires visibles et des boules visibles.

5.2.3. Les class CustomCircle et Camera et Rectange

• ***La class CustomCircle***

La vue personnalisée CustomCircle permet de dessiner des formes à un endroit donné sur FenetreJeu grâce à un System.Drawing.Graphics propre à ce CustomCircle et non celui de FenetreJeu. Ainsi pour déplacer la forme il suffit de déplacer le CustomCircle qui le contient. L'affichage des joueurs et des boules sont donc chacun contenu dans des CustomCircle.

• ***La class Camera***

La class Camera est une class static qui représente la zone de la carte visible sur l'écran. Cette class permet de n'afficher que les joueurs et boules situés dans cette zone. Elle est définit par deux entiers X et Y qui correspondent aux coordonnées du centre de l'écran sur la carte.

• ***La class Rectange (oups)***

La vue personnalisée Rectange est utilisé pour dessiner les bordures de la zone de jeu. Il y a 4 instanciations de Rectange pour les quartes bord.

5.2.4. Initialisation de la Camera

Lors de la création du joueur principal (client), Camera est initialisé (Camera.InitialiseCamera()). Camera.X et Camera.Y sont initialisés tels que le centre de l'écran corresponde aux coordonnées du joueur sur la carte pour que le joueur soit affiché au milieu de l'écran.

Les boules et les joueurs adverses sont affichés lors de l'actualisation de l'affichage après réceptions d'un message provenant du serveur.

Les barriè

5.2.5. Actualisation de l'affichage et déplacement

Lors d'une actualisation de l'affichage dans la fonction Timer_Tick() une nouvelle position du joueur sur l'écran est calculée à partir de la distance entre le joueur et le curseur. Le calcul de cette nouvelle position permet d'avoir une vitesse de déplacement constante quelque soit la distance entre le curseur et le joueur.

Si cette nouvelle position se situe à moins 250px (distance calculée grâce à la fonction Camera.GetDistanceWithJoueursClient()) du centre de l'écran alors le joueur est déplacé à cette position puis la position des CustomCircle contenant les adversaires est modifié et on affiche ceux qui rentrent dans la zone de la carte couverte par l'écran (grâce à la fonction Camera.DisplayAllMoovingPlayers()). Les boules restent immobiles.

Sinon c'est la "caméra" qui bouge. Le joueur reste immobile sur l'écran et les valeurs Camera.X et Camera.Y sont modifiés et varient de la différence entre la position actuelle du joueur sur l'écran et la nouvelle position calculée, projeté sur chaque axe. Puis à partir de ces nouvelles valeurs de X et Y Camera.RedisplayAll() modifie la position des CustomCircle contenant les boules et les joueurs et affiche celles qui rentrent dans la zone de la carte couverte par l'écran.

Les bordures sont déplacés (DisplayBarrière()) par rapport à Camera.X et Camera.Y à chaque actualisation de l'interface.

5.3. Affichage du classement

5.3.1. Initialisation

Le classement est initialisé lors de la création de FenetreJeu via la fonction Fenetre_Load().

5.3.2. Actualisation

Lorsqu'un joueur grossis (réception d'un message de type 2 dans FenetreJeu.TraiterMessage / Joueur.GrossirJoueurClient) ou meurt (via réception d'un message de type 4) Classement.AjoutPointEquipe et Classement.RetraitPointEquipe sont appelé pour trier le tableau ClassementEquipe après avoir ajouté les points à l'équipe correspondante.

Le score de chaque équipe est ensuite actualisé en changeant le texte des System.Windows.Forms.Label.

6. Divers

6.1. Stockage dans un fichier Texte

Les champs de connexion (ip, port, pseudo, couleur) sont stockés sur un fichier txt au même emplacement que l'exécutable. Ainsi lors de la connexion ces champs sont pré remplis pour permettre au joueur de relancer un partie rapidement. Le score personnel maximal est aussi stocké après la mort du joueur pour être affiché dans un Label sur l'interface de jeu lors de la connexion.

Les données dans le fichier sont stockées sous le format suivant: *ip,port,pseudo,couleur,score* .

Pour modifier le fichier, il est supprimé grâce à System.IO.File.Delete puis recréer grâce à System.IO.StreamWriter. Etant donné le peu de données stockés, supprimer le fichier puis le recréer est plus simple que de le lire puis de le modifier.

6.2. Les sons

Deux sons (fichier wav) sont stockées dans le répertoire WinformTest/bin/Debug/ et sont joué grâce System.Media.SoundPlayer lorsqu'une boule est mangée par le joueur principal ou un joueur tué.