

Réalisation d'un jeu vidéo et de son serveur portable

Fabio Dubath

Daniel Roulin



Supervisé par M. Daniel Kessler
Collège André-Chavanne

2022

Contenu

1	Introduction	3
1.1	Les outils	3
1.2	Le jeu	3
2	Structure	4
2.1	Partie matérielle (Hardware)	4
2.2	Partie logicielle (Software)	6
2.2.1	Serveur	6
2.2.2	Client	10
2.3	Protocole	13
2.3.1	TCP OU UDP ?	13
2.3.2	Le client (n') est (pas) roi	13
2.3.3	Types de données	13
3	Quelques chiffres	14
3.1	Lignes de code en fonction du temps	14
3.2	Contributions	15
4	Démarche	16
4.1	L'idée	16
4.2	Raspberry Pi	16
4.2.1	Découverte de la carte et génération de Wifi	16
4.3	Tutoriel de Tom Weiland	17
4.4	Chronologie	17
4.4.1	9 Mars 2022	17
4.4.2	17 Mars 2022	17
4.4.3	18 Mars 2022	17
4.4.4	22 Mars 2022	18
4.4.5	22 au 28 Mars 2022	18
4.4.6	28 Mars au 1 Mai 2022	18
4.4.7	1 au 14 Mai 2022	18
4.4.8	11 au 29 Août 2022	19
4.4.9	30 Août au 19 Septembre	20
4.4.10	19 Septembre à début Novembre	20
5	Problèmes rencontrés	21
5.1	FPS du serveur	21
5.2	Scene parsing	22
5.2.1	Format	22
5.2.2	Parsing	22
5.2.3	Fonctions ObjectToCollider et ObjectToVector2	24
5.2.4	Améliorations	24
5.3	Tri du classement	25
5.4	Sélection des points d'apparition	27
5.5	Trajectoires des balles	28
5.6	Style artistique	28

6 Conclusion	29
7 Remerciements	29
A Paquets	32
A.1 ServerPackets (envoyés du serveur au client)	32
A.2 Client Packets (envoyés du client au serveur)	34

1 Introduction

Les jeux multijoueurs peuvent être classés en deux catégories: ceux qui se jouent sur le même appareil, sans connexion à internet, et ceux qui se jouent en ligne sur des appareils séparés. Cependant, il existe des situations où aucune de ces deux options ne sont viables. C'est pourquoi il nous est venu l'idée d'explorer une troisième configuration, celle de pouvoir jouer ensemble sur plusieurs appareils sans connexion à internet. Nous avons donc imaginé un boîtier, créant son propre réseau Wifi et permettant aux personnes environnantes de jouer simultanément, ainsi qu'un jeu adapté à cette nouvelle plateforme. Ce rapport documente la réalisation de ce projet.

1.1 Les outils

L'outil de base pour la création de notre jeu vidéo est le moteur de jeu Unity. Un moteur de jeu est une application contenant du code déjà en place qui enlève aux développeurs beaucoup de tâches fastidieuses comme le rendu des images sur l'écran ou encore l'exportation du jeu sur les différents systèmes d'exploitation. Unity a été publié en 2005 par Unity Technologies et permet de faire des jeux en 2D ou 3D, avec la possibilité d'exporter ces derniers sur une multitude de plateformes. Concernant le code, Unity utilise le langage C#, qui est un langage de programmation orientée objet créé par Microsoft. Il est dérivé du C++ et est très proche du Java, dont il reprend la syntaxe générale ainsi que les concepts. Nous nous sommes aussi aidés de GitHub, qui est un service web d'hébergement et de gestion de développement de logiciels, utilisant le logiciel de gestion de versions Git [10] ainsi que de l'outil d'organisation Trello [17] appartenant à Atlassian et basé sur le style Kanban.

1.2 Le jeu

Le concept du jeu est très simple. Les joueurs, dans une arène, ont accès à des armes et peuvent s'éliminer en se tirant dessus, ils réapparaissent une fois éliminés. Quand un joueur élimine un autre, il gagne un point et celui qui a le plus de points à la fin du temps imparti a gagné. Le concept s'inspire du style "battle royale", très populaire ces dernières années.

Lorsque l'on démarre le jeu, on se retrouve sur le menu principal qui donne accès au menu de configuration, au bouton pour rejoindre le serveur et au bouton pour quitter le programme.



(a) Menu de début de jeu



(b) Menu de configuration

Le menu de configuration permet à l'utilisateur de choisir l'adresse IP et le port du serveur sur lequel il veut se connecter, le programme indiquant si l'adresse insérée est valide.

Une fois connecté, le joueur se retrouve dans la salle d'attente où il peut se déplacer et configurer son nom d'utilisateur le temps que tout le monde soit là. Quand les joueurs décident de démarrer la partie, ils se positionnent tous sur le carré "GO".



(a) Salle d'attente

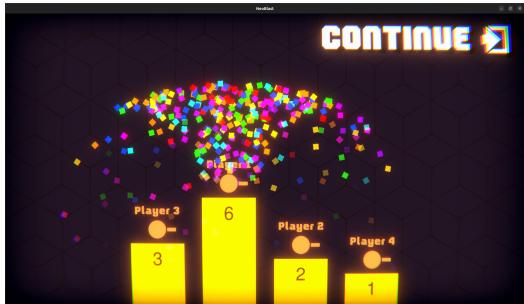


(b) Jeu

Durant la partie, le joueur a accès au menu de pause qui lui permet de revenir au menu principal, mais qui ne met pas en pause la partie. Quand la partie est terminée, tous les joueurs sont emmenés au podium qui montre leur nombre d'éliminations. Puis, ils sont renvoyés vers la salle d'attente où une nouvelle partie peut être commencée.



(a) Menu de pause



(b) Podium

2 Structure

Ce projet peut être séparé en deux parties : une partie matérielle (hardware) et une partie logicielle (software). En raison de problèmes d'approvisionnement qui ont suivi la récente pandémie et suite à la complexité inattendue de la partie logicielle, la partie matérielle a été moins développée que ce que nous souhaitions.

2.1 Partie matérielle (Hardware)

Cette partie consiste en une Raspberry Pi 3 modèle B+ avec une carte microSD, d'un petit câble et d'une batterie portable de 5000 mAh fournissant 5 V à 2.1 A. La Raspberry provient d'une connaissance de Fabio, qui a gracieusement accepté de nous la prêter pour notre projet. La figure 4 montre deux photos de la Raspberry Pi et deux photos du dispositif complet.



Figure 4: Images du dispositif

2.2 Partie logicielle (Software)

La partie logicielle est constituée du code nécessaire au fonctionnement du jeu. Afin de conserver un historique des versions ainsi que pour pouvoir collaborer, nous avons utilisé le programme Git [10]. Notre projet est hébergé à cette adresse : <https://github.com/DanielRoulin/Waypoint>

Le plus simple pour étudier le projet consiste à télécharger le code source avec la commande suivante:

```
git clone https://github.com/DanielRoulin/Waypoint.git
```

Un dossier nommé *Waypoint* contenant le code source sera normalement créé.

Waypoint est le nom que nous avons donné à ce projet, NeoBlast est le nom du jeu et Theel est le nom d'utilisateur de Fabio sur GitHub.

Voici une vue d'ensemble du projet:

```
~/Waypoint$ tree -a -L 1
Waypoint
├── Client
├── Server
├── Design
├── Statistics
├── Rapport
├── LICENSE
└── README.md
.
└── .git

4 directories
```

Tout d'abord, nous avons les deux dossiers les plus importants : *Client* et *Server*. C'est là que se situe la plus grande partie de notre travail, séparée entre le code exécuté sur le client (téléphone ou ordinateur) et le code exécuté sur le serveur (Raspberry Pi ou VPS [1]). Le dossier *Rapport* contient ce PDF ainsi que le document L^AT_EX utilisé pour le créer. Ensuite, le dossier *Design* contient nos réflexions concernant la conception du boîtier et un logo, tandis que le dossier *.git* stocke toutes les informations nécessaires au contrôle de version. Le dossier *Statistics* contient les scripts générant les graphiques de la section 3. Le fichier *LICENSE* contient la licence de ce projet. Nous avons choisi la *GNU General Public License v3.0*¹, qui autorise l'utilisation commerciale, la modification et la distribution de notre code, à condition que nous soyons cité et que le projet utilise la même licence. Finalement, le fichier *README.md* contient un très court résumé du projet.

2.2.1 Serveur

Ce dossier a la structure d'une application .NET classique. Il est donc nécessaire d'installer ce programme pour compiler notre projet. Pour installer .NET sur Ubuntu, par exemple, il suffit d'exécuter cette commande:

```
sudo apt-get update && sudo apt-get install -y dotnet6
```

Des instructions plus détaillées sont disponibles à cette adresse:

<https://learn.microsoft.com/en-us/dotnet/core/install/linux>

Pour lancer le projet, le plus simple consiste à l'ouvrir avec l'éditeur *VSCode*, car le dossier *.vscode* contient la configuration nécessaire. Sinon, pour simplement lancer le projet, il faut exécuter la commande suivante dans le dossier *Server*:

¹<https://www.gnu.org/licenses/gpl-3.0.en.html>

```
dotnet run
```

Et pour compiler le projet pour Linux (toujours dans le dossier *Server*):

```
build GameServer.csproj -r linux-x64 -o bin/Linux/
```

ou pour compiler le projet pour la Raspberry Pi:

```
build GameServer.csproj -r linux-arm -o bin/Raspberry/
```

Après avoir compilé le projet, il ne faut pas oublier de copier le dossier *Scenes* là où il est exécuté.

Note : Si le message suivant apparaît, c'est parce que nous utilisons une version dépassée de .NET, car le tutoriel que nous avons suivi utilise cette dernière, et que .NET est rétrocompatible. (voir la section 4 pour plus de détails)

```
warning NETSDK1138: The target framework 'netcoreapp3.0' is out of support and will not  
→ receive security updates in the future.
```

Voici les différents fichiers du serveur :

```
tree -a --dirsfirst Server
Server/
├── .vscode
│   ├── launch.json
│   ├── settings.json
│   └── tasks.json
├── bin
│   ├── Debug
│   │   └── ...
│   ├── Linux
│   │   └── ...
│   └── Raspberry
│       └── ...
└── Scenes
    ├── Maps
    │   ├── Map1.unity
    │   ├── Map2.unity
    │   ├── Map3.unity
    │   ├── Map4.unity
    │   └── Map5.unity
    └── WaitingRoom.unity
    ├── GameServer.csproj
    ├── .gitignore
    ├── Client.cs
    ├── Colliders.cs
    ├── Constants.cs
    ├── GameLogic.cs
    ├── Item.cs
    ├── Packet.cs
    ├── Player.cs
    ├── Program.cs
    ├── Projectile.cs
    ├── Scene.cs
    ├── Server.cs
    ├── ServerHandle.cs
    ├── ServerSend.cs
    ├── ThreadManager.cs
    └── Utilities.cs
22 directories, 753 files
```

- Le dossier `.vscode` contient la configuration de notre éditeur: *VSCODE*.
- Le fichier `tasks.json` contient les instructions pour compiler le projet.
- Le fichier `launch.json` contient celle pour le déboguer.
- Le fichier `settings.json` contient nos différents réglages de l'éditeur.
- Le dossier `bin`, qui n'est pas inclus sur github, contient tous nos builds [5] pour les différentes plateformes, en l'occurrence Linux et Raspberry.
- Le dossier `obj` est créé lors de la compilation et contient les fichiers créés pendant l'étape intermédiaire entre la compilation et le linting.
- Le dossier `Scenes`, qui contient une copie des terrains du client, au format `.unity`. Ce format s'appelle UnityYAML [11] et, comme son nom l'indique, il s'agit du format des scènes Unity, basé sur le format YAML. Il contient la salle d'attente, `WaitingRoom.unity` et toutes les cartes du jeu : `Map1.unity`, `Map2.unity`, ...
- `GameServer.csproj` contient les réglages du projet. En voici une copie :

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <RollForward>Major</RollForward>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="System.Numerics.Vectors" Version="4.5.0" />
    <PackageReference Include="YamlDotNet" Version="11.2.1" />
  </ItemGroup>
</Project>
```

On peut voir que l'on demande au compilateur de créer un fichier exécutable, en utilisant la version 3.0 de `.NET` si possible mais en acceptant n'importe quelle version au-dessus. De plus, nous importons le package `System.Numerics.Vectors`, qui nous permet d'utiliser des vecteurs ainsi que le package `YamlDotNet` [12] qui nous permet de parser [13] du *YAML*.

- `.gitignore` contient une liste des fichiers ou dossiers que Git [10] doit ignorer. En l'occurrence, on ne souhaite ignorer que les résultat de la compilation:

```
bin
obj
```

Les fichiers restants sont les scripts, qui sont le cœur du serveur. Voici un tableau résumant leurs fonctions respectives :

Fichier	Fonction
<code>Program.cs</code> ¹	Premier script appelé lors de l'exécution, <code>Program.cs</code> est responsable du démarrage du serveur et de la <i>game loop</i> dans deux différents <i>threads</i> . Il actualise le jeu à intervalles réguliers.

<code>ThreadManager.cs</code> ¹	Ce script gère simplement la communication entre le <i>thread</i> du serveur et la <i>game loop</i> (<i>main thread</i>).
<code>Server.cs</code> ¹	<code>Server.cs</code> contient la classe la plus importante du projet, la classe <i>Server</i> . Cette classe stocke toutes les informations de la partie, telle que sa durée, la liste des joueurs, des objets et des projectiles, etc. Elle est responsable de l'initialisation du serveur et de la connexion des nouveaux joueurs.
<code>Client.cs</code> ¹	Ce fichier contient la classe <i>Client</i> , qui est instancié pour chaque joueur connecté. Cependant, elle n'est responsable que de la gestion de la connexion, les fonctions liées au joueur se situent dans la classe <i>Player</i> . Chaque <i>Client</i> a un <i>Player</i> associé et est chargé de l'initialiser et de le réinitialiser quand le client se déconnecte.
<code>Packet.cs</code> ¹	Cette classe contient la logique nécessaire à l'encodage et au décodage des paquets, qui sont formés d'octets. Il contient aussi la liste et le nom des différents paquets, qui doit impérativement être la même sur le client et le serveur.
<code>ServerSend.cs</code>	Cette classe contient les fonctions responsables de la construction et de l'encodage des paquets envoyés par le serveur aux clients.
<code>ServerHandle.cs</code>	Cette classe contient les fonctions responsables du décodage et de l'interprétation des paquets envoyés par les clients au serveur.
<code>GameLogic.cs</code>	C'est ici que se situe la logique du jeu. Ce script contient la fonction <i>Update</i> , qui est appelée à chaque tic et s'occupe de mettre à jour les joueurs, les projectiles et les objets, ainsi que les fonctions permettant de démarrer et d'arrêter le jeu.
<code>Scene.cs</code>	Ce fichier est responsable de l'interprétation des scènes, qui se situe dans le dossier <i>Scenes</i> . Il est responsable de trouver certains objets dans les scènes, tel que les obstacles, les éléments déclencheur d'action (<i>trigger</i>) ainsi que les points d'apparition (<i>spawn points</i>).
<code>Player.cs</code>	Cette classe est instanciée pour chaque joueur, et contient toutes leurs informations, telles que leur position, leur rotation, leur nom, leur armes, etc. Elle est aussi responsable de leurs interactions, telles que récupérer des armes ou tirer des projectiles.
<code>Item.cs</code>	Cette classe est instanciée pour chaque objet du jeu, et contient toutes leurs informations, telles que leur position et type.
<code>Projectile.cs</code>	Cette classe est instanciée pour chaque projectile du jeu, et contient toutes leurs informations, telles que leur position, rotation et type. Elle est aussi responsable de changer leur position, en suivant différentes trajectoires en fonction de leur type.

Collider.cs	Ce fichier est responsable de détecter les collisions du jeu. Il contient deux classes: <i>RectCollider</i> et <i>CircleCollider</i> , représentant respectivement des rectangles et des cercles. Chaque entité du jeu possède un de ces <i>collider</i> . Chacune des classes contient une fonction permettant de vérifier qu'elle n'intersecte pas avec un autre collider.
Utilities.cs	Cette classe contient diverses fonctions pratiques, liées notamment à la génération de nombres aléatoires, de position aléatoire spécifiques et au logging
Constants.cs	Cette classe contient toutes les constantes du jeu, telles que le nombre de tic par seconde, la taille des terrains, les caractéristiques des différentes armes et les noms par défaut des joueurs.

¹*Nous n'avons que très peu modifié ces fichiers, ils proviennent du tutoriel de Tom Weiland. Voir la section 4.*

2.2.2 Client

Le deuxième grand dossier de ce projet est le dossier *Client*. Ce dernier contient l'entièreté du code source du client, qui est un projet Unity. Le lien ci-dessous explique comment installer la plateforme, ce qui est nécessaire pour ouvrir et compiler le projet:

<https://unity3d.com/get-unity/download>

Notre projet utilise la version 2020.3.2f1, mais fonctionne sur d'autres versions supérieures. En voici une vue d'ensemble:

```
tree -a --dirsfirst Client
Client
├── Assets
│   ├── Animations
│   │   └── ...
│   ├── Fonts
│   │   └── ...
│   ├── Graphic
│   │   └── ...
│   ├── Heathen Engineering
│   │   └── ...
│   ├── Hexanim
│   │   └── ...
│   ├── Materials
│   │   └── ...
│   ├── Prefabs
│   │   └── ...
│   ├── Scenes
│   │   ├── Maps
│   │   │   ├── Map1.unity
│   │   │   ├── Map2.unity
│   │   │   ├── Map3.unity
│   │   │   ├── Map4.unity
│   │   │   └── Map5.unity
│   │   └── Empty.unity
```

```

    └── End Screen.unity
    └── WaitingRoom.unity
Scripts
├── CamerasC.cs
├── Client.cs
├── ClientHandle.cs
├── ClientSend.cs
├── DontDestroy.cs
├── EndScreenBar.cs
├── EndScreenManager.cs
├── GameManager.cs
├── Item.cs
├── Leaderboard.cs
├── LeaderboardEntry.cs
├── Menu.cs
├── Packet.cs
├── PlayerController.cs
├── PlayerManager.cs
├── Projectile.cs
└── ThreadManager.cs
ProjectSettings
└── ...
UserSettings
└── EditorUserSettings.asset
.vscode
└── settings.json
.gitignore

```

55 directories, 1635 files

Note: Les fichiers .meta ne sont pas affichés. Ils sont créés par Unity et ne contiennent que des métadonnées concernant les fichiers du même nom.

- Le dossier *.vscode* contient, comme sur le serveur, la configuration de notre éditeur.
- Le fichier *settings.json* contient une liste de fichier à ne pas montrer dans l'éditeur afin de simplifier l'affichage.
- Les dossiers *ProjectSettings* et *UserSettings* contiennent respectivement les réglages du projet et ceux de l'utilisateur.
- Les dossiers *Animations*, *Fonts* et *Materials* contiennent des animations, des polices d'écriture ainsi que des matériaux (texture).
- Le dossier *Prefab* contient tous les *prefabs* du projet. Un *prefab*, dans Unity, est un objet pouvant être instancié plusieurs fois dans une scène. Par exemple, les joueurs et les projectiles sont des *prefabs*.
- Le dossier *Scenes* contient tous les terrains du jeu et doit être exactement le même que sur le serveur.
- Le dossier *Scripts* contient tous les scripts du projet.

Voici un tableau résumant leurs fonctions respectives:

Fichier	Fonction
ThreadManager.cs ¹	Ce script gère simplement la communication entre le <i>thread</i> du client et la <i>game loop (main thread)</i> .
Client.cs ¹	Cette classe est responsable de la connexion avec le serveur. Elle gère le transfert de paquets via TCP et UDP.
Packet.cs ¹	Ce fichier est le même que sur le serveur.
ClientSend.cs	Cette classe contient les fonctions responsables de la construction et de l'encodage des paquets envoyés par le client au serveur, tels que les boutons qu'il presse, sa rotation ou son nom d'utilisateur.
ClientHandle.cs	Cette classe contient les fonctions responsables du décodage et de l'interprétation des paquets envoyés par le serveur au client, tel que la position des autres entités ou le nom des autres joueurs.
GameManager.cs	Cette classe est la plus importante du client, car elle est responsable du déroulement du jeu. Par exemple, elle s'occupe de charger les différentes cartes, de démarrer et terminer les parties et de faire apparaître les joueurs, les projectiles et les objets sur la scène.
Menu.cs	Cette classe gère tous les menus (UI) du jeu.
PlayerManager.cs	Cette classe est associée à chaque joueur apparaissant sur la scène et est responsable de leur apparence et comportement.
PlayerController.cs	Cette classe s'occupe d'envoyer au serveur le nom du joueur local ainsi que les différents boutons qu'il presse.
Item.cs	Cette classe est instanciée pour chaque objet du jeu, et contient toutes leurs informations, telles que leur position et type.
Projectile.cs	Cette classe est associée à chaque projectile et s'occupe de les déplacer et de les détruire.
CameraSC.cs	Ce script est rattaché à la caméra et est responsable de son mouvement, pour qu'elle suive le joueur de façon fluide.
Leaderboard.cs	Cette classe gère le tableau des scores qui apparaît en haut à droite durant une partie. Il se charge notamment de trier les joueurs par leurs scores.
LeaderboardEntry.cs	Cette classe est associée à chaque entrée du tableau des scores, stocke leur position et leur texte, et est responsable de l'animation de leur position.
EndScreenManager.cs	Ce fichier contient la classe responsable de l'écran apparaissant à la fin d'une partie
EndScreenBar.cs	Cette classe est responsable du podium qui apparaît à la fin de la partie.
DontDestroy.cs	Ce script est très petit et ne fait qu'indiquer au compilateur de ne pas détruire certains éléments quand une nouvelle carte est chargée.

¹*Nous n'avons que très peu modifié ces fichiers, ils proviennent du tutoriel de Tom Weiland. Voir la section 4.*

2.3 Protocole

Pour communiquer entre eux, le client et le serveur utilisent deux protocoles de base: TCP [15] et UDP [16]. Ils transfèrent des séquences d'octets, appelées paquets (*packets*). La signification d'un paquet dépend de son `packetId` mais aussi de l'état du client qui le reçoit. Par défaut, le port 26950 est utilisé.

Voici une vue d'ensemble du format d'un paquet:

Nom	Type	Notes
<code>length</code>	int	Longueur du message. Est égal à la longueur du <code>packetId</code> + la longueur des données du paquet.
<code>packetId</code>	int	Type du paquet, voir les sections suivantes.
<code>data</code>	bytes	Liste d'octets dépendant du <code>packetId</code> .

Le même format est utilisé si la connexion est en TCP ou en UDP.

2.3.1 TCP OU UDP ?

Le jeu utilise deux types de connexion: TCP et UDP. TCP à l'avantage de garantir l'arrivée d'un paquet si un client peut le recevoir, ainsi que de conserver l'ordre des paquets. UDP est plus rapide que TCP, mais ne vérifie pas la réception ni l'ordre des paquets. C'est pourquoi TCP est privilégié pour les messages importants, envoyés qu'une seule fois, tandis qu'UDP est utilisé pour les messages envoyés à chaque tic. En pratique, seul le temps, la position et la rotation utilisent UDP. Un autre inconvénient d'UDP est que certains réseaux, notamment celui de notre collège, semblent bloquer ce protocole. C'est pourquoi nous aurions souhaité ajouter à notre jeu un mode "TCP only", ce que nous n'avons malheureusement pas eu le temps de faire.

2.3.2 Le client (n') est (pas) roi

Lorsque l'on développe un jeu en ligne, il est nécessaire de choisir la liberté du client. Par exemple, le client pourrait simplement envoyer sa position sous forme de vecteur au serveur. Cependant, cela implique au serveur de faire confiance au client pour qu'il n'envoie pas des positions interdites ou erronées, car ce dernier pourrait ainsi se téléporter. C'est pourquoi notre jeu, comme la plupart des jeux en ligne, demande au client d'uniquement envoyer ses clics de boutons, puis le serveur envoie la position du joueur comme celle de n'importe quelle autre joueur, sous forme de vecteur. De plus, si plusieurs paquets de mouvement sont envoyés en un seul tic, le serveur n'applique que le dernier reçu. Cela permet de contrôler la vitesse des joueurs. La seule exception à cette règle est la rotation du joueur, que le client contrôle entièrement afin d'éviter un délai nettement visible.

2.3.3 Types de données

Voici un tableau résumant les 6 types de données utilisés par le protocole:

Nom	Taille (octets)	Encode	Notes
byte	1	Un octet, nombre entier entre 0 et 255	Unsigned 8-bit integer
int	4	Un nombre entier entre -2147483648 et 2147483647	Signed 32-bit integer, two's complement
float	4	Un nombre à virgule flottante	A single-precision 32-bit IEEE 754 floating point number
bool	1	Vrai ou faux	True is encoded as 0x01, false as 0x00.
string (n)	4 + n	Une séquence de caractère du tableau ASCII (7 bits)	ASCII (7 bits) string prefixed with its size in bytes as an int
vector2	2 * 4	Un vecteur 2 dimensionnel	X as a float, followed by Y as a float

Ces représentations proviennent simplement de la fonction `BitConverter.GetBytes` du *namespace* [19] `System`.

Une liste détaillée des paquet se situe dans l'annexe A.

3 Quelques chiffres

3.1 Lignes de code en fonction du temps

Le graphique 5 montre le nombre de lignes contenu dans les fichiers .cs (fichiers de code C#) en fonction du temps. Il a pu être fait en remontant dans l'historique des versions via Git. On constate la réorganisation du projet mi-mars 2022, suivi de l'ajout du code du tutoriel, puis l'adaptation du projet à nos besoins en fin du mois de mars. Les changements brutaux sont dus à des conflits de fusion de notre code (*merge conflicts*) qui ont été résolus avec plus ou moins de succès. La grosse vallée fin mai représente un changement de nos menus (UI), d'où la suppression et l'ajout rapide de code. On peut aussi voir une période de calme pendant les vacances, de juin à fin août. L'agitation à la fin du mois de mai correspond de nouveau à des conflits de fusion, car nous travaillions en même temps aux mêmes endroits. Finalement, les changements du mois d'octobre représentent notre débogage, c'est pourquoi peu d'ajouts de code peuvent être observés.

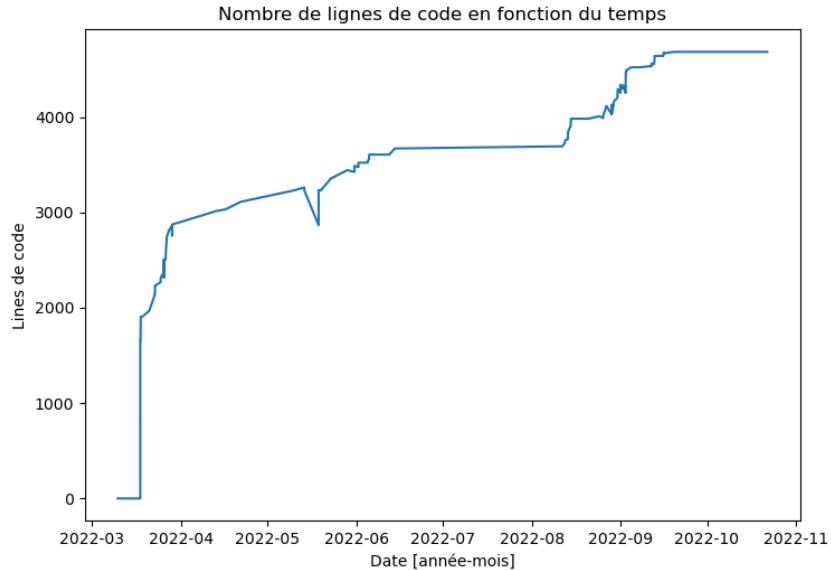
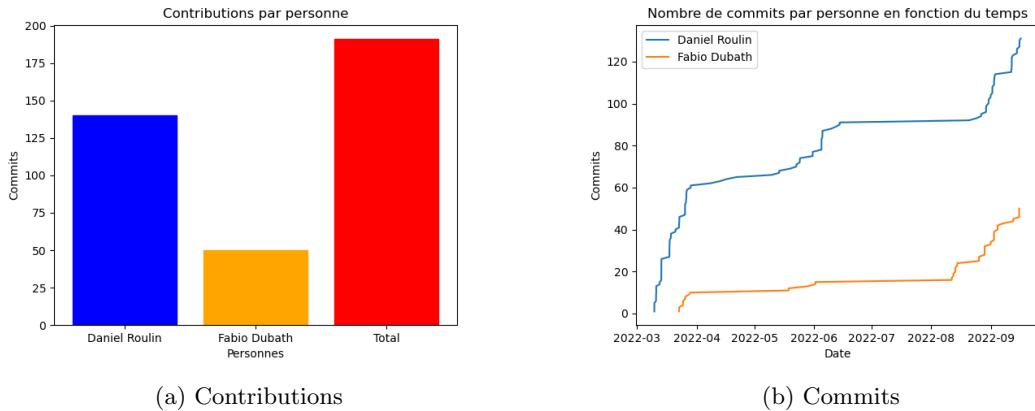


Figure 5: Lignes de code en fonction du temps

3.2 Contributions



Le graphique 6a montre le nombre de *commits* par personne. À première vue, on pourrait penser que Fabio a moins contribué aux projets. Cependant, les commits ne sont pas représentatifs du travail total fourni. En effet, nous avons tous les deux différentes façons de contribuer. Par exemple, Daniel préfère créer un *commit* à chaque changement, tandis que Fabio le fait quand il a fini toute une partie. De plus, Fabio s'est aussi concentré sur la partie artistique, qui n'est pas mesurée par les commits.

Le graphique 6b montre l'évolution du nombre de *commits* en fonction du temps. On y voit nos

deux différents styles, ainsi que les différentes périodes du projet. Le décalage entre les deux courbes est aussi dû au fait que le *repo* était à l'origine utilisé uniquement pour le serveur, avant d'avoir fusionné avec le reste du projet (voir la section 4). C'est pourquoi les premiers *commits* ont été faits seulement par Daniel et commencent plus tôt.

4 Démarche

Ce chapitre parle du processus de création du projet, de la première idée au produit fini.

4.1 L'idée

Tout a commencé en cours de philo, durant une phase soudaine d'ennui profond. Nous nous sommes mis à discuter. C'est là que nous avons réalisé qu'il n'était alors pas possible de jouer à des jeux en ligne sans être connecté à internet, et qu'il était possible de changer cela. Il se trouve que nous avons eu cette réflexion à peu près au moment de choisir le sujet de notre travail de maturité, et que nous avions à nous deux l'essentiel des connaissances nécessaires pour le démarrer. Cependant, simplement fabriquer un boîtier fonctionnant à la fois comme modem Wifi et comme serveur n'est pas aussi complexe que ça, c'est pourquoi nous avons aussi décidé de développer un jeu pouvant marcher avec ce dispositif. C'est d'ailleurs la partie du projet qui nous a pris le plus de temps. Nous nous sommes donc inscrits pour un travail pratique à deux, choisissant M. Kessler, professeur d'application des mathématiques et membre de la commission de validation des TM, pour nous accompagner.

4.2 Raspberry Pi

Nous avons donc commencé par chercher une Raspberry Pi [7], car c'est la base de la partie matérielle. Cependant, nous avons été rapidement confrontés à un problème: elles n'étaient plus en stock nulle part et les meilleures estimations prévoyaient une livraison vers fin 2022. En effet, la récente pandémie de Covid-19 a causé une pénurie mondiale de circuits imprimés, qui sont des composants indispensables à la fabrication de puces électroniques et donc de Raspberry Pi. De plus, la demande a, elle aussi, fortement augmenté en 2021. C'est pourquoi la *Raspberry Pi Fondation* a eu de gros problèmes de stock et a décidé de prioriser les clients professionnels qui dépendent de Raspberry Pi pour faire fonctionner leurs entreprises.²

Heureusement, Fabio connaissait grâce à un camp d'informatique Gérard Ineichen, vice-président de Caritas Jeunesse Genève et surtout passionné d'informatique, qui a accepté de nous prêter une Raspberry en attendant.

C'est ainsi que le 9 mars 2022, Fabio est allé chercher une Raspberry Pi 3 Model B+ chez Gérard et que nous avons pu commencer à travailler.

4.2.1 Découverte de la carte et génération de Wifi

Après l'avoir reçue, il a fallu configurer la Raspberry. Pour commencer, il nous fallait un périphérique de stockage pour stocker le système d'exploitation. Nous nous sommes donc dirigés vers la Fnac de Balexert à Genève, afin d'acheter une carte micro SD de 16 Go, visible dans la figure 7. Il a fallu ensuite installer un système d'exploitation sur la carte SD. Nous avons donc suivi les instructions de

²<https://www.raspberrypi.com/news/production-and-supply-chain-update/>

la documentation officielle³, et avons installé *Raspberry Pi Imager* sur l'ordinateur de Daniel. Puis, nous avons sélectionné le système d'exploitation *Raspberry Pi OS Lite (32-bit)*, car notre modèle à une architecture en *arm32* [8] et que nous n'avions pas besoin d'un environnement de bureau [9]. Ensuite, nous avons installé et paramétré le programme *ssh* [6], afin de pouvoir accéder à la carte à distance. Finalement, l'avons configurée pour qu'elle puisse émettre son propre réseau Wifi, en suivant à nouveau les instructions officielles⁴. Par ailleurs, Daniel avait constaté une faute de frappe sur la documentation. Mais comme le site était Open Source [4], il a simplement fait une pull request, qui a été acceptée quelques jours après.⁵

Pour finir, la carte créait un réseau Wifi. Il ne restait plus "que" à faire le jeu.

4.3 Tutoriel de Tom Weiland

Nous avions tous les deux déjà programmé des jeux, mais nous n'avions aucune expérience des jeux en ligne. Nous avons donc étudié les différentes façons d'en créer un avec Unity, le moteur de jeu que nous souhaitions utiliser. Il existait de nombreuses solutions offertes directement par la plateforme, mais la plupart ne semblaient pas assez complexes pour nos besoins et leurs critiques étaient plutôt négatives. De plus, la plupart utilisaient *Unity Server*, mais ce programme ne pouvait pas se compiler pour l'architecture de la Raspberry (arm32 [8]). Il était aussi très lourd et ne nous paraissait pas nécessaire pour notre utilisation.

C'est ainsi que nous avons découvert le youtubeur Tom Weiland et son fabuleux tutoriel [3] sur le Networking en C#. Son projet était basé sur .NET [2], qui utilise le même langage que Unity.

Cependant, nous avons quand même dû l'adapter à nos besoins. En effet, son client était en 3D, tandis que nous voulions faire un jeu en 2D. De plus, il a fallu changer l'interface utilisateur, les graphismes, ainsi que, le fonctionnement du jeu lui-même. La figure 8 montre ce tutoriel.



Figure 7: Carte micro SD à côté d'une pièce de 1 franc pour l'échelle

4.4 Chronologie

4.4.1 9 Mars 2022

Comme dit précédemment, c'est à cette date que nous avons reçu la Raspberry Pi

4.4.2 17 Mars 2022

C'est à ce moment-là que nous avons réorganisé le projet, qui était initialement consacré à la création du Wifi, pour qu'il corresponde à la structure du tutoriel.

4.4.3 18 Mars 2022

Création de notre *Trello* [17]. La figure 9 montre une capture d'écran de son état à la fin du projet.

³<https://www.raspberrypi.com/documentation/computers/getting-started.html#installing-the-operating-system>

⁴<https://www.raspberrypi.com/documentation/computers/configuration.html#setting-up-a-routed-wireless-access-point>

⁵<https://github.com/raspberrypi/documentation/commit/e7e2d7ffcba31cf6ad233fc4a70dbfe623a820f4>

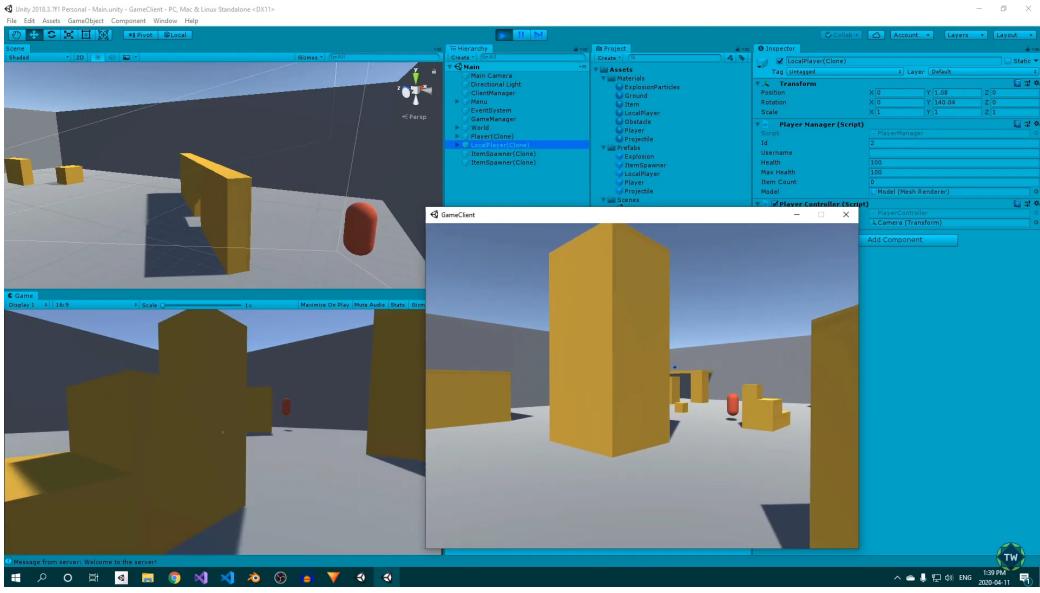


Figure 8: Jeu de Tom Weiland.

4.4.4 22 Mars 2022

À cette date, Fabio a pu rejoindre le projet, car en raison d'un manque de temps et de divers problèmes techniques, il n'a pas pu le faire avant. Il ajouta donc les personnages qu'il avait dessinés, visibles à la figure 10, avec le *commit* "[Fabio] joined the game"⁶. (Les personnages n'ont pas été intégrés dans la version finale, voir la section 5.6 pour plus d'informations). Nous avons aussi expérimenté notre premier *merge conflict*, car nous utilisions différentes versions d'Unity. C'est aussi à ce moment-là que nous avons fini de suivre le tutoriel et que nous avons commencé à travailler sur notre jeu, notamment en créant un script pour importer les scènes Unity sur notre serveur.

4.4.5 22 au 28 Mars 2022

Entre ces deux dates, nous avons ajouté les animations des joueurs et les objets qui deviendront plus tard les armes. Fabio a commencé à travailler sur les trajectoires des projectiles. De plus, Daniel a ajouté le *parsing* des scènes sur le serveur.

4.4.6 28 Mars au 1 Mai 2022

Durant cette période, nous avons principalement travaillé sur les mécaniques de base, telles que la réapparition des joueurs, la salle d'attente et les conditions de début et de fin d'une partie. Nous avons aussi commencé à améliorer l'interface utilisateur.

4.4.7 1 au 14 Mai 2022

Durant cette période, nous avons ajouté les collisions au serveur, créé toutes les *maps* actuelles, visibles sur la figure 11, avec le *tileset* [18] de Fabio. Nous avons continué à travailler sur l'interface

⁶<https://github.com/DanielRoulin/Waypoint/commit/1bb1d6869941d451d384b7a802145543b2b81c72>

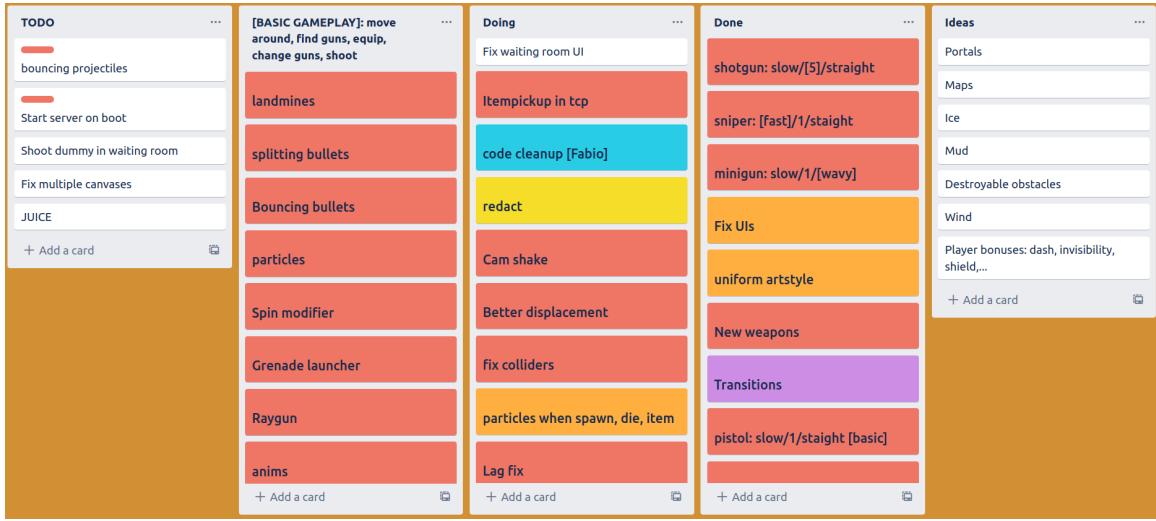


Figure 9: Capture d'écran de notre *Trello* à la fin du projet.

du client et essayé d'automatiser la compilation du client et du serveur avec *Github Action*⁷, ce qui n'a malheureusement pas abouti, car ce système était trop complexe et n'était pas adapté à nos besoins. Nous avons aussi corrigé des bugs mineurs que nous avions trouvés.

4.4.8 11 au 29 Août 2022

Après deux mois de pause pendant les vacances d'été, nous avons continué à travailler sur le projet. Pour commencer, Fabio a ajouté beaucoup d'armes et d'objets et a créé un système modulaire permettant de créer des armes en fonction de différents paramètres. Puis, Daniel a séparé les différentes *maps* dans leur propre scène (fichier). Cela a aussi nécessité de récrire le code du serveur,

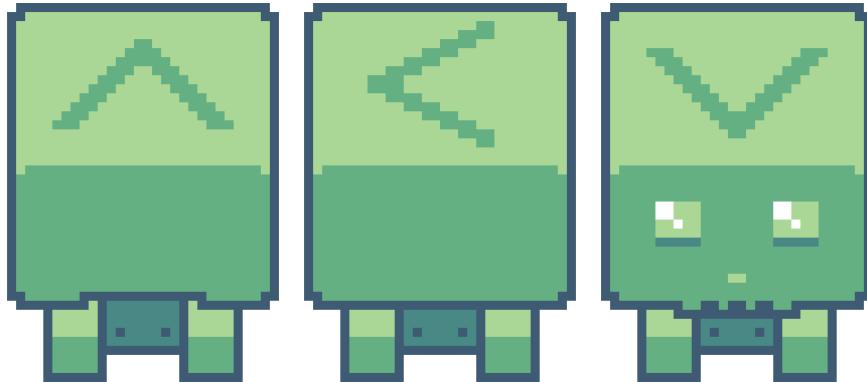


Figure 10: Dessins de joueurs par Fabio

⁷Github Action est un service proposé par Github permettant d'exécuter du code sur leur serveur à chaque *commits*.



Figure 11: Différentes cartes de jeux

et a été la cause de notre plus gros *merge conflict*, comme nous pouvons le voir à la Figure 12.

4.4.9 30 Août au 19 Septembre

Durant cette dernière période, nous n'avons pas ajouté de fonctionnalités majeures, mais nous nous sommes concentrés sur la correction de bugs, l'amélioration des menus, avec notamment l'ajout du menu *option* et *pause*, ainsi que des changements plutôt de l'ordre graphique.

4.4.10 19 Septembre à début Novembre

En cette période, nous avons rédigé notre rapport, nettoyé le code et les fichiers du projet et nous avons fait les statistiques de la section 3.

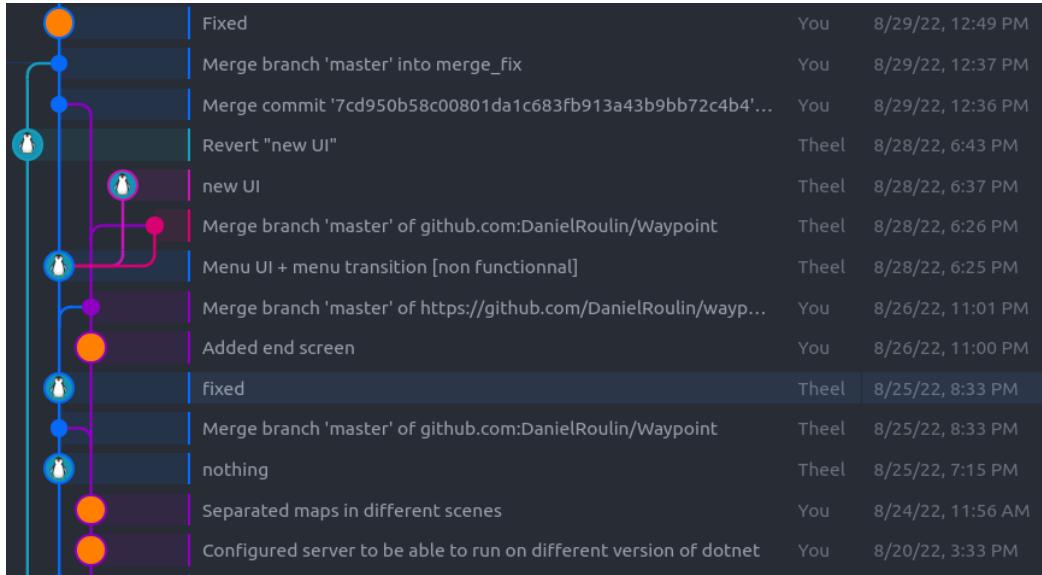


Figure 12: Diagramme montrant les différentes branches du projet pendant le conflit

5 Problèmes rencontrés

5.1 FPS du serveur

Le premier problème rencontré a été un problème d'optimisation. En effet, le serveur est un programme relativement simple qui peut être facilement exécuté par l'ordinateur. Mais cette facilité faisait que le programme était exécuté plusieurs centaines de fois par seconde, ce qui prenait trop de performances (mémoire, processeur, etc) et empêchait une bonne synchronisation entre le serveur et les clients. La solution était simple, il a suffi de limiter le nombre d'exécutions par seconde à un nombre raisonnable avec le code ci-dessous qui "mets en pause" l'exécution du programme pour qu'il n'y ait qu'une exécution par 30^{ème} de seconde:

```
while (isRunning)
{
    while (_nextLoop < DateTime.Now)
    {
        GameLogic.Update();
        _nextLoop = _nextLoop.AddMilliseconds(Constants.MS_PER_TICK);
        if (_nextLoop > DateTime.Now)
        {
            Thread.Sleep(_nextLoop - DateTime.Now);
        }
    }
}
```

Extrait 1: Code pour la limitation du nombre d'exécutions par seconde

5.2 Scene parsing

Quand nous avons commencé à créer des cartes pour les joueurs, nous nous sommes rapidement heurtés à un problème: comment synchroniser les terrains du serveur avec ceux du client ? En effet, il est nécessaire que les deux connaissent la position des obstacles, des points d'apparition et des déclencheurs d'événement, le premier afin d'appliquer la logique du jeu, le deuxième dans le but d'afficher correctement ces éléments. La meilleure solution que nous avons trouvée consiste à ajouter des informations destinées au serveur sur les scènes du client, puis de les copier sur le serveur pour qu'il puisse les interpréter. Cependant, il n'existe pas de *parser* [13] pour le format des scènes utilisé par Unity (UnityYaml [11]). En revanche, comme son nom l'indique, le format est basé sur le YAML [14], pour lequel des librairies existent en C#. Nous avons donc décidé de créer notre propre *parser* par-dessus, que l'on peut trouver dans le fichier *Scene.cs*.

5.2.1 Format

```
%YAML 1.1
%TAG !u! tag:unity3d.com,2011:
--- !u!1 &835401395
GameObject:
  m_ObjectHideFlags: 0
  m_Component:
    - component: {fileID: 835401396}
  m_Name: Square (2)
  m_TagString: ServerCollideable
  m_IsActive: 1
--- !u!4 &835401396
Transform:
  m_ObjectHideFlags: 0
  m_GameObject: {fileID: 835401395}
  m_LocalRotation: {x: -0, y: -0, z: -0, w: 1}
  m_LocalPosition: {x: 0, y: 7.5, z: 0}
  m_LocalScale: {x: 30, y: 1, z: 1}
  m_Children: []
  m_Father: {fileID: 1928389500}
  m_RootOrder: 1
  m_LocalEulerAnglesHint: {x: 0, y: 0, z: 0}
```

Extrait 2: Exemple d'un fichier UnityYaml avec un seul GameObject.

Comme on le voit dans l'extrait 2, un fichier *.unity* comporte d'abord un en-tête. De plus, ce fichier est composé d'une liste de fichiers *Yaml* valides, séparés par une courte chaîne de caractères contenant leur type et leur id. Chaque fichier *Yaml* représente un composant de la scène. Chacun de ces composants sont liés entre eux par leur id pour former une hiérarchie cohérente.

5.2.2 Parsing

Pour commencer, nous découpons le fichier par l'en-tête de chaque sous-fichiers, qui commence toujours par "----". Puis, nous parsons chaque fichier en utilisant la librairie YamlDotNet [12]. Ensuite, nous ajoutons le résultat à un dictionnaire, avec pour index son *object_id*. C'est la fonction *ParseObjects*, visible dans l'extrait 3.

Ensuite, dans la fonction *ParseScene*, visible dans l'extrait 4, nous étudions chaque objets parsés et vérifions le *tag* de chaque *GameObject*:

```

private void ParseObjects()
{
    string file = File.ReadAllText(path);
    string[] files_array = file.Split("--- ");
    List<string> files = files_array.ToList<string>();
    files.RemoveAt(0);

    objects = new Dictionary<string, dynamic>();
    foreach (var f in files)
    {
        string[] splits = f.Split("\n", 2);
        string header = splits[0];
        string document = splits[1];
        string object_id = header.Split("&")[1];

        objects[object_id] = ParseYaml(document);
    }
}

```

Extrait 3: Fonction ParseObjects.

- Si c'est un *ServerCollideable*, nous le transformons en un *collider* du serveur avec la fonction `ObjectToCollider` et l'ajoutons à la liste d'obstacles de cette scène.
- Si c'est un *ServerTrigger*, nous le transformons aussi en *collider* et assignons la variable `trigger`. Il ne peut donc y avoir qu'un seul *trigger* par scène, et seule la scène WaitingRoom possède un trigger.
- Si c'est un *ServerSpawn*, nous utilisons la fonction `ObjectToVector2` pour le transformer en vecteur et ajoutons ce vecteur à la liste de points d'apparitions du serveur.

```

foreach (KeyValuePair<string, dynamic> element in objects)
{
    string type = ((IDictionary<String, Object>)element.Value).Keys.First();

    if (type == "GameObject")
    {
        dynamic gameObject = element.Value.GameObject;
        string tag = gameObject["m_TagString"];

        switch (tag)
        {
            case "ServerCollideable":
                obstacles.Add(ObjectToCollider(gameObject));
                break;
            case "ServerTrigger":
                trigger = ObjectToCollider(gameObject);
                break;
            case "ServerSpawn":
                Vector2 spawn_position = ObjectToVector2(gameObject);
                mapSpawns.Add(spawn_position);
                break;
        }
    }
}

if (!obstacles.Any())
{
    throw new Exception($"No obstacles found in scene {path}");
}

```

Extrait 4: Fonction ParseScene.

5.2.3 Fonctions ObjectToCollider et ObjectToVector2

Dans cette fonction, qui apparaît dans l'extrait 5, nous récupérons le composant `Transform` de l'objet reçu en entrée, qui contient des informations sur sa position, sa rotation et sa taille. Nous créons ensuite un *collider* à partir de ces informations. Il est intéressant de noter l'utilisation du type `dynamic`, qui nous permet d'avoir des variables dont on ne connaît pas le type, telle que le résultat du *parsing*. De plus, les *colliders* ne peuvent pas gérer les rotations, donc le programme rapporte une erreur si un objet en possède. Finalement, nous utilisons aussi l'option `CultureInfo.InvariantCulture` quand nous transformons la chaîne de caractères en nombre à virgule flottante. En effet, sans cette option, le programme fonctionnait sur l'ordinateur de Daniel, mais pas sur celui de Fabio, car ce dernier était en français. La fonction `ObjectToCollider` est similaire à la précédente, sauf qu'elle renvoie un vecteur à deux dimensions plutôt qu'un *collider*.

5.2.4 Améliorations

En rédigeant ce texte, nous avons réalisé qu'il était possible d'optimiser ce programme en n'utilisant le parser YAML que quand il est nécessaires. En effet, nous avons réalisé après coup que le type de composant est présent dans l'en-tête de chaque sous-fichier. Il serait donc possible de ne *parser* que les fichiers de type *GameObject* et *Transform*.

```

private RectCollider ObjectToCollider(dynamic gameObject)
{
    foreach (var component in gameObject["m_Component"])
    {
        string fileID = component["component"]["fileID"];
        dynamic component_object = objects[fileID];
        string component_type = ((IDictionary<String,
→ Object>)component_object).Keys.First();

        if (component_type == "Transform")
        {
            dynamic transform = component_object.Transform;
            if (float.Parse(transform["m_LocalRotation"]["x"]) != 0 ||
                float.Parse(transform["m_LocalRotation"]["y"]) != 0 ||
                float.Parse(transform["m_LocalRotation"]["z"]) != 0)
            {
                throw new Exception($"[Import Error]: GameObject {gameObject["m_Name"]} is
→ rotated. This is not supported");
            }

            return new RectCollider(
                new Vector2(float.Parse(transform["m_LocalPosition"]["x"],
→ CultureInfo.InvariantCulture), float.Parse(transform["m_LocalPosition"]["y"],
→ CultureInfo.InvariantCulture)),
                new Vector2(float.Parse(transform["m_LocalScale"]["x"],
→ CultureInfo.InvariantCulture), float.Parse(transform["m_LocalScale"]["y"],
→ CultureInfo.InvariantCulture))
            );
        }
        throw new Exception($"[Import Error]: GameObject {gameObject["m_Name"]} does not have
→ a transform component");
    }
}

```

Extrait 5: Fonction ObjectToCollider

5.3 Tri du classement

1	Player 2	3
2	Player 1	2
3	Player 3	1

1	Player 2	2
1	Player 3	2
3	Player 1	1

Figure 13: Images montrant le classement des joueurs, avec à droite une situation où deux joueurs ont le même score.

Un problème que Daniel a rencontré en travaillant sur l’interface du client a été le suivant : comment trier et afficher le score des joueurs ? En effet, la solution naïve serait de simplement de trier les joueurs par ordre croissant et leur donner une position dans le classement correspondant à leur rang. Cependant, il se trouve que plusieurs joueurs peuvent avoir le même score, ce qui peut créer des erreurs ou afficher des entrées du classement par-dessus d’autre. De plus, le rang des joueurs peut ne pas être continu, comme on le voit à droite de la figure 13. L’extrait 6 montre la solution que nous avons trouvée.

Pour commencer, nous utilisons les fonctions `GroupBy()` et `OrderByDescending()`, qui viennent du *namespace* [19] `System.Linq`. Ce *namespace* permet de facilement trier des listes et des dictionnaires. La première fonction permet de grouper les éléments d’une liste dans un dictionnaire, en fonction d’une condition. Ici, nous regroupons toutes les entrées avec le même score. La deuxième fonction nous permet de trier ces groupes par ordre décroissant de leurs scores.

Durant la suite du programme, nous utilisons deux variables : `totalRank` et `rank`. `totalRank` est l’ordre apparent des entrées, tandis que `rank` correspond au rang affiché à droite des noms des joueurs (voir la figure 13). Ces deux valeurs sont initialisées à 1 au début du programme. Le reste du programme fonctionne de la façon suivante :

D’abord, on regarde chaque groupe trié. Puis, on donne à la variable `rank` la valeur de la variable `totalRank`. Cela permet d’avoir des rangs discontinus. Ensuite, pour chaque entrée du groupe, on met à jour son rang et sa couleur en fonction de la variable `rank` et de sa position en fonction de `totalRank`. Finalement, on ajoute 1 à la variable `totalRank`.

C’est donc en utilisant quelque fonction de triage et deux variables que nous arrivons à résoudre le problème apparemment simple du tri du classement.

```
public void SortLeaderboard()
{
    var grouped = entries.GroupBy(p => int.Parse(p.Value.kills.text));
    var ordered = grouped.OrderByDescending(g => g.Key);
    int totalRank = 1;
    foreach (var group in ordered)
    {
        var reorderedGroup = group.OrderBy(entry => entry.Value.username.text);
        int rank = totalRank;
        foreach (KeyValuePair<int, LeaderboardEntry> entry in reorderedGroup)
        {
            entry.Value.rank.text = rank.ToString();
            float y = entry.Value.rectTransform.rect.height/2 -
→      totalRank*entry.Value.rectTransform.rect.height;
            entry.Value.targetPosition = new Vector2(0, y);
            totalRank++;
        }
    }
}
```

Extrait 6: Fonction `SortLeaderboard`, qui donne un rang et une position à chaque joueur en fonction de son score. Extrait tiré du fichier `Leaderboard.cs` du client.

5.4 Sélection des points d'apparition

```
public static Vector2 RandomFreeGoodPosition(float _radius)
{
    Vector2 bestPosition = Vector2.Zero;
    float bestDistance = 0f;
    for (int i = 0; i < 20; i++)
    {
        Vector2 _position = RandomFreeCirclePositionInMap(_radius);
        float smallestDistance = 10000f;
        foreach (Item item in Server.items.Values)
        {
            float distance = Vector2.Distance(item.position, _position);
            smallestDistance = MathF.Min(smallestDistance, distance);
        }
        foreach (Client client in Server.clients.Values)
        {
            if (client.player == null) continue;
            float distance = Vector2.Distance(client.player.position, _position);
            smallestDistance = MathF.Min(smallestDistance, distance);
        }
        if (bestDistance < smallestDistance)
        {
            bestDistance = smallestDistance;
            bestPosition = _position;
        }
    }
    return bestPosition;
}
```

Extrait 7: Fonction `RandomFreeGoodPosition`, qui permet de générer des bonnes positions pour l'apparition d'objets et de joueurs. Tiré du fichier `Utilites.cs` du serveur.

Au début de la création du jeu, nous faisions naïvement apparaître les joueurs et les objets en choisissant des coordonnées X et Y aléatoirement entre 0 et la largeur (ou la longueur) du terrain. Cependant, cette méthode comportait plusieurs problèmes. D'abord, il était possible que ces entités apparaissaient dans des obstacles, ce qui empêchait les joueurs de bouger et rendait les objets inaccessibles. Nous avions donc créé une fonction générant des positions aléatoires jusqu'à en trouver une qui n'intersectait aucun des obstacles de la scène. Cette solution permettait d'éviter le problème initial, mais en introduisait un autre : si un joueur est particulièrement malchanceux, il était possible qu'il apparaisse toujours désarmé en face d'un ennemi et que les armes se retrouvent toujours du côté de cet ennemi. Pour parer à cet inconvénient, nous avons rajouté une étape à la génération de positions, visible dans l'extrait 7.

Le fonctionnement de cette fonction est simple : on génère 20 positions grâce à la méthode précédemment expliquée, puis on choisit la position la plus loin de tout joueur et de toute arme. Pour ce faire, on commence par initialiser deux variables, `bestPosition` et `bestDistance`. Ces variables représentent la meilleure position et la plus grande distance trouvée à cet instant. Puis, pour chaque position, on regarde sa distance à l'objet ou le joueur le plus proche. Si cette distance est supérieure à `bestDistance`, alors `bestDistance` devient cette distance et `bestPosition` devient la position actuelle. Cette méthode nous permet d'avoir une bonne distribution des entités sur le terrain. La qualité de cette distribution est en partie liée au nombre de positions testé. Dans notre cas, après quelques tests, 20 semblait être le meilleur compromis entre la qualité et la performance.

5.5 Trajectoires des balles

Nous avons trouvé l'idée amusante que les joueurs puissent tirer des projectiles avec des trajectoires spéciales. La trajectoire la plus difficile à implémenter a été celle sinusoïdale, car nous ne comprenions pas comment implémenter cette fonctionnalité. Le problème a été résolu grâce au code ci-dessous qui, chaque 30^{ème} de seconde, modifie la trajectoire perpendiculaire à la direction de tir et lui donne la valeur du sinus de la distance entre la position de tir et la position de la balle. Le résultat est une balle qui avance et qui se balance de gauche à droite :

```
Vector2 dist = position - initialPos;
float totaldist = MathF.Abs(MathF.Sqrt(dist.X * dist.X + dist.Y * dist.Y));
Vector2 perp;
if (direction.Y == 0)
{
    perp = new Vector2(0, 1);
}
else
{
    perp = new Vector2(1, -direction.X / direction.Y);
}
float lenght = MathF.Sqrt(perp.X * perp.X + perp.Y * perp.Y);
perp = new Vector2(perp.X / lenght, perp.Y / lenght);
float crossZ = direction.X * perp.Y - perp.X * direction.Y;
if (crossZ < 0)
{
    perp = new Vector2(-perp.X, -perp.Y);
}
float mod = position.X / direction.X;
position += direction * speed * 2;
position += perp * speed * MathF.Cos(totaldist * frequency) * height;
```

Extrait 8: Code responsable de la trajectoire des balles sinusoïdale. Tiré du fichier `Projectile.cs` du serveur.

5.6 Style artistique

Bien que nous voulions à la base créer un jeu dans le style "pixel art", nous nous sommes plutôt dirigés vers un style "néon", plus simple, plus rapide à réaliser et mieux adapté à l'ambiance du jeu. Le style graphique a pu être renforcé grâce à du *post-processing* (traitement d'image) qui donne au jeu un ton plutôt explosif et lumineux.



(a) Jeu avec post-processing.



(b) Jeu sans post-processing.

6 Conclusion

Ce projet nous a appris à utiliser de nouveaux outils, tel que le moteur de jeu Unity, qui était nouveau pour Daniel, le code en *networking* et les systèmes de communication, ainsi que l'écriture en L^AT_EX. En plus des connaissances techniques, ce projet nous a aussi appris à collaborer sur une longue période et à faire preuve de persévérance.

Le projet a aussi un peu dévié par rapport à l'objectif initial. L'idée originale prévoyait de faire un jeu mobile, ce qui n'a pas été possible à cause de contraintes de temps. De plus, l'aspect de construction de sort a aussi été abandonnée et a été remplacé par le système d'arme. La création d'un boîtier personnalisé a aussi dû être mise de côté, car la réalisation du jeu a pris plus de temps que prévu et Daniel a décidé de se concentrer sur ce dernier. Le style graphique du jeu a aussi été simplifié afin de dédier plus de temps aux fonctionnalités.

En somme, nous sommes contents du résultat final; nous avons réussi à ouvrir une troisième voie dans le domaine du jeu multijoueur en créant un jeu local sur des appareils distincts. Nous sommes convaincus que cette idée a du potentiel pour de futurs projets, voire un avenir commercial.

7 Remerciements

Nous tenons à remercier toutes les personnes qui nous ont permis de venir à bout de ce projet:
M. Daniel Kessler, notre maître accompagnant, qui nous a conseillés, orientés et corrigés durant notre travail.

Nos parents, qui ont attentivement relu et corrigés ce document.

Gérard Ineichen, qui nous a prêté du matériel indispensable.

Et bien sûr nos amis, qui ont participé au design et qui furent les premiers à essayer le jeu.

Références

- [1] Un VPS (Virtual Private Server) est un serveur dédié pouvant être loué au près d'hebergeurs web. Plus d'informations à cette adresse:
https://fr.wikipedia.org/wiki/Serveur_d%C3%A9di%C3%A9_virtuel
- [2] .NET est une plateforme de développement dotée d'outils et de bibliothèques permettant de créer tout type d'application, notamment web, mobile, desktop, jeux, IoT, cloud et microservices. C'est la plateforme de notre serveur. Plus d'information à ces adresses:
<https://dotnet.microsoft.com>
https://fr.wikipedia.org/wiki/.NET_Core
- [3] Tom Weiland est un créateur de contenu et programmeur, connu notamment pour ses tutoriels sur Unity et surtout sur le networking en C#, qui nous ont servis de base pour ce projet. Ci-joint l'adresse de son site web, son profil github et le tutoriel que nous avons suivi:
- Site web: <https://tomweiland.net/>
 - Github: <https://github.com/tom-weiland>
 - Tutoriel: <https://youtube.com/playlist?list=PLXkn83W0QkfnqsK8I0RAz5AbUxfg3bOQ5>
- [4] Du code est dit Open Source s'il est mis à disposition gratuitement pour d'éventuelles modifications et redistributions. Plus d'information à cette adresse:
https://en.wikipedia.org/wiki/Open_source
- [5] En informatique, un build est la version compilé et exécutable d'un programme, contrairement au code source.
- [6] Le protocole Secure Shell (ssh) est un protocole de réseau cryptographique permettant d'exploiter des services réseau en toute sécurité sur un réseau non sécurisé. Ses applications les plus notables sont la connexion à distance et l'exécution de lignes de commande. Plus d'information à cette adresse:
https://fr.wikipedia.org/wiki/Secure_Shell
- [7] La Raspberry Pi est un nano-ordinateur monocarte à processeur ARM de la taille d'une carte de crédit conçu par des professeurs du département informatique de l'université de Cambridge. Plus d'information à ces adresses:
https://fr.wikipedia.org/wiki/Raspberry_Pi,
<https://www.raspberrypi.org/help/what-%20is-a-raspberry-pi/>
- [8] Les architectures ARM sont des architectures de type RISC 32 bits. Dotés d'une architecture relativement plus simple que d'autres familles de processeurs et faibles consommateurs d'électricité, les processeurs ARM sont aujourd'hui dominants dans le domaine de l'informatique embarquée. C'est l'architecture des Raspberry Pi. Plus d'information à cette adresse:
https://fr.wikipedia.org/wiki/Architecture_ARM
- [9] En informatique, un environnement de bureau (de l'anglais *desktop environment*) est l' interface utilisateur graphique qui ressemble à un bureau. Plus d'information à cette adresse:
https://fr.wikipedia.org/wiki/Environnement_de_bureau
- [10] Git permet de gérer les versions et de collaborer sur un projet. Plus d'information à cette adresse:
<https://git-scm.com/>

- [11] UnityYAML est un format de fichier optimisé par Unity pour stocker des scènes, basé sur le format YAML (<https://yaml.org/>) mais ne supportant qu'un sous-ensemble des spécifications. Plus d'informations à cette adresse:
<https://docs.unity3d.com/Manual/UnityYAML.html>
- [12] YamlDotNet est une librairie pour la plateforme .NET qui permet de parser du YAML. Nous l'utilisons pour l'interprétation des scènes sur le serveur. Plus d'informations à cette adresse:
<https://github.com/aaubry/YamlDotNet>
- [13] En informatique, un parser est programme qui prend des données d'entrée (souvent du texte) et les transforment en une représentation compréhensibles par l'ordinateur. Plus d'informations à cette adresse:
https://fr.wikipedia.org/wiki/Analyse_syntaxique
- [14] YAML (YAML Ain't Markup Language) est un langage de sérialisation de données lisible par l'homme. Il est couramment utilisé pour les fichiers de configuration et dans les applications où les données sont stockées ou transmises. YAML ressemble dans son usage au JSON et XML. Il utilise l'indentation de style Python pour indiquer l'imbrication et est facilement extensible. Plus d'informations à cette adresse:
<https://fr.wikipedia.org/wiki/YAML>
- [15] Le protocole TCP (Transmission Control Protocol) permet le transfert fiable de données entre ordinateurs sur le même réseau. Contrairement au protocole UDP, il est orienté sur des connections et garanti l'ordre des paquets. Documentation officielle accessible ici:
<https://www.rfc-editor.org/rfc/rfc9293.html>
- [16] Le protocole UDP (User Datagram Protocol) est un protocole permettant la transmission rapide de données entre ordinateurs sur le même réseau. UDP ne garantit ni l'ordre ni l'arrivée des paquets. Il est cependant bien plus rapide que le protocole TCP. Documentation officielle accessible ici:
<https://www.rfc-editor.org/rfc/rfc768.html>
- [17] Trello est un outil de gestion de projet en ligne. Il repose sur une organisation des projets en planches listant des cartes, chacune représentant des tâches. Les cartes sont assignables à des utilisateurs et sont mobiles d'une planche à l'autre, traduisant leur avancement. C'est l'outil que nous avons utilisé pour décider qui ferait quoi. Plus d'informations à ces adresses:
<https://fr.wikipedia.org/wiki/Trello>
<https://trello.com/fr>
- [18] Un tileset (ensemble de tuiles en français) est un ensemble d'images de même taille permettant de créer facilement des environnements.
- [19] Les programmes C# sont organisés à l'aide de *namespace*. Ces derniers sont utilisés système d'organisation interne d'un programme et permettent de présenter les éléments de programme qui sont exposés à d'autres programmes. Plus d'informations à cette adresse:
<https://learn.microsoft.com/fr-fr/dotnet/csharp/language-reference/language-specification/namespaces>

A Paquets

A.1 ServerPackets (envoyés du serveur au client)

Welcome Premier message envoyé par le serveur, sert à donner un ID au client.

Packet Id	UDP/TCP	Nom	Type	Note
1	TCP	message	string	Message de bienvenue, toujours défini comme "Welcome to the server!", utilisé uniquement pour débogage.
		clientId	int	Id du nouveau client

Spawn Player Envoyé par le serveur lorsqu'un nouveau joueur rejoint la partie, aussi envoyé au joueur se connectant.

Packet Id	UDP/TCP	Nom	Type	Note
2	TCP	client id	int	ID du joueur (identique à son ID client)
		username	string	Nom du joueur
		position	Vector2	Position du joueur(0:0 représente le milieu de l'écran)
		rotation	float	Rotation de l'arme du joueur, en degré, sens trigonométrique, 0° est à l'horizontale à droite

Disconnect Player Envoyé par le serveur aux joueurs restants lorsque un joueur averti le serveur qu'il se déconnecte.

Packet Id	UDP/TCP	Nom	Type	Note
3	TCP	(aucun)		

Set Name Envoyé par le serveur à tous les joueurs lorsqu'un joueur change son nom.

Packet Id	UDP/TCP	Nom	Type	Note
4	TCP	client id	int	ID du joueur
		username	string	Nom du joueur

Start Game Envoyé par le serveur à tous les joueurs au début d'une partie

Packet Id	UDP/TCP	Nom	Type	Note
5	TCP	duration	float	Durée de la partie, en secondes
		map id	int	ID de la map, 0 car 0 représente la salle d'attente.

End Game Envoyé par le serveur à tous les joueurs à la fin d'une partie

Packet Id	UDP/TCP	Nom	Type	Note
6	TCP	(aucun)		

Game Time Envoyé par le serveur à chaque tic pour les informer du temps restant.

Packet Id	UDP/TCP	Nom	Type	Note
7	UDP	(aucun)		

Player Position Envoyé par le serveur à tous les joueurs lorsqu'un joueur bouge.

Packet Id	UDP/TCP	Nom	Type	Note
8	UDP	client id	int	ID du joueur
		position	Vector2	Position du joueur

Player Rotation Envoyé par le serveur à tous les joueurs lorsqu'un joueur tourne.

Packet Id	UDP/TCP	Nom	Type	Note
9	UDP	client id	int	ID du joueur
		rotation	float	Rotation du joueur

Player Respawned Envoyé par le serveur à tous les joueurs lorsqu'un joueur réapparaît.

Packet Id	UDP/TCP	Nom	Type	Note
12	TCP	(aucun)		

Player Hit Envoyé par le serveur à tous les joueurs lorsqu'un joueur est touché par un projectile.

Packet Id	UDP/TCP	Nom	Type	Note
10	TCP	client id	int	ID du joueur touché
		by	int	ID du joueur ayant tiré le projectile

Player Ammo Envoyé par le serveur au joueur lorsqu'il tire ou récupère une arme pour l'informer du nombre de balles qu'il possède.

Packet Id	UDP/TCP	Nom	Type	Note
11	TCP	(aucun)		

Item Spawned Envoyé par le serveur à tous les joueurs lorsqu'un objet apparaît.

Packet Id	UDP/TCP	Nom	Type	Note
13	TCP	item id	int	ID de l'objet
		position	Vector2	Position de l'objet
		type	int	Type de l'objet. (pistolet, fusil, mitraillette,...)

Item Picked Up Envoyé par le serveur à tous les joueurs lorsqu'un objet est récupéré par un joueur.

Packet Id	UDP/TCP	Nom	Type	Note
14	TCP	item id	int	ID de l'objet
		client id	int	ID du joueur

Projectile Spawned Envoyé par le serveur à tous les joueurs lorsqu'un projectile est tiré.

Packet Id	UDP/TCP	Nom	Type	Note
15	TCP	projectile id	int	ID du projectile
		position	Vector2	Position du projectile
		client id	int	ID du joueur qui a tiré le projectile

Projectile Position Envoyé par le serveur à tous les joueurs lorsqu'un projectile bouge.

Packet Id	UDP/TCP	Nom	Type	Note
16	UDP	projectile id	int	ID du projectile
		position	Vector2	Position du projectile

Projectile Destroyed Envoyé par le serveur à tous les joueurs lorsqu'un projectile est détruit.

Packet Id	UDP/TCP	Nom	Type	Note
17	TCP	(aucun)		

A.2 Client Packets (envoyés du client au serveur)

Welcome Received Premier message envoyé par le client, sert à confirmer l'ID du client.

Packet Id	UDP/TCP	Nom	Type	Note
1	TCP			(aucun)

Player Name Envoyé au serveur lorsque le joueur change son nom

Packet Id	UDP/TCP	Nom	Type	Note
2	TCP			(aucun)

Player Movement Envoyé au serveur à chaque tic, contient les boutons que le joueur presse ainsi que sa rotation.

Packet Id	UDP/TCP	Nom	Type	Note
3	UDP	inputs length	int	Taille de la liste de boutons
		inputs	list [bool]	État des boutons: false = normal, true = pressé
		rotation	float	Rotation du joueur (angle entre l'horizon, le joueur et la souris)

Player Shoot Envoyé au serveur quand le client clique.

Packet Id	UDP/TCP	Nom	Type	Note
4	TCP			(aucun)

Player End Game Envoyé au serveur quand le client appuie sur le bouton Continue quand la partie est finie. Signal au serveur que le joueur est prêt à jouer à nouveau.

Packet Id	UDP/TCP	Nom	Type	Note
5	TCP			(aucun)