Waypoint

Plan:

- Introduction
    - pourquoi du comment
- Le jeu
    - mécaniques, screenshots, déroulement d'une partie
- Structure
    - structure (unity, comment network)
- Démarche
    - Base (TW)
- Problèmes rencontrés, aspects intéressants
    - FPS of server
    - Scène parsing
    - Leaderboard sorting
    - Collisions
    - Spawn points selection
    - Bullet trajectory
    - Art style
    - Map design
    - Weapon design
    - UI
- Démonstration
    - Screen/video
- Conclusion

Introduction

Internet est un outil qui a révolutionné notre mode de vie, il nous permet de communiquer avec plus de 4,9 milliards de personnes tout autour du globe afin de découvrir et de partager tout ce qui est imaginable. Bien qu'un nombre gigantesque de personnes y ont accès, assez peu de monde sait réellement comment fonctionne ce réseau informatique mondial, qui repose pourtant sur l'idée assez simple de faire communiquer des ordinateurs. Étant passionnés d'informatique, nous sommes vite arrivés au point où nous voulions nous immerger dans le code réseau (Networking en anglais) ce qui nous permettrait de faire des programmes mettant en communication plusieurs personnes. L'idée de faire un jeu fut assez évidente puisqu'elle nous permettrait de le faire essayer à n'importe qui. Alors quel jeu allons nous faire et comment allons nous le créer ?

Les outils

L'outil de base pour la création de notre jeu vidéo sera 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 de s'occuper du rendu des images sur l'écran ou encore de l'exportation du jeu sur les différents systèmes d'exploitation. Unity est un moteur qui à été publié en 2005 par *Unity Technologies* et qui 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, commercialisé par Microsoft depuis 2002 et destiné à développer sur la plateforme Microsoft .NET. Il est dérivé du C++ et très proche du Java dont il reprend la syntaxe générale ainsi que les concepts, y ajoutant des notions telles que la surcharge des opérateurs, les indexeurs et les délégués. 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.

Le Jeu

En ce qui concerne l'idée du jeu, nous sommes partis sur un concept très simple; Les joueurs, dans une arène, ont accès à des armes et peuvent s'éliminer en se tirant dessus. Quand un joueur en élimine un autre il gagne un point et celui qui a le plus de points à la fin du temps délimité à gagné.
[Add commented screens]

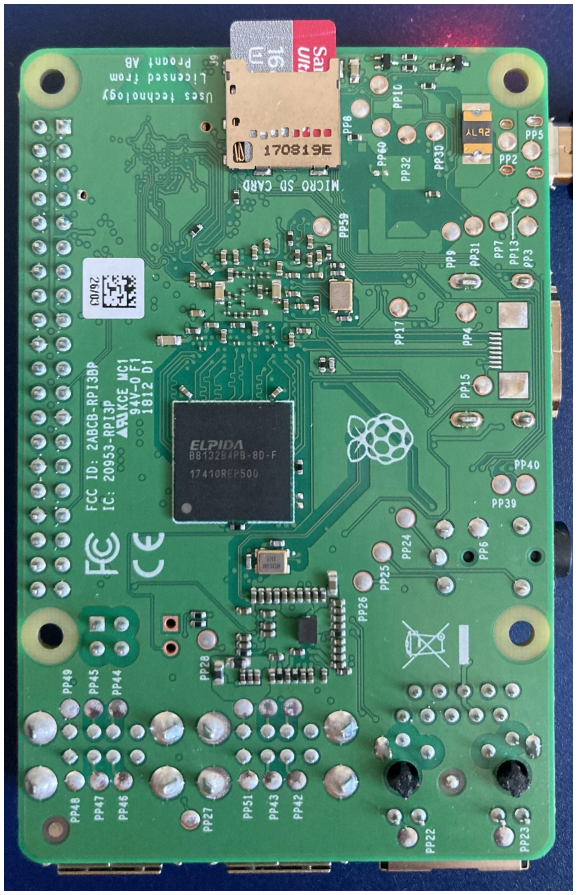
Structure:

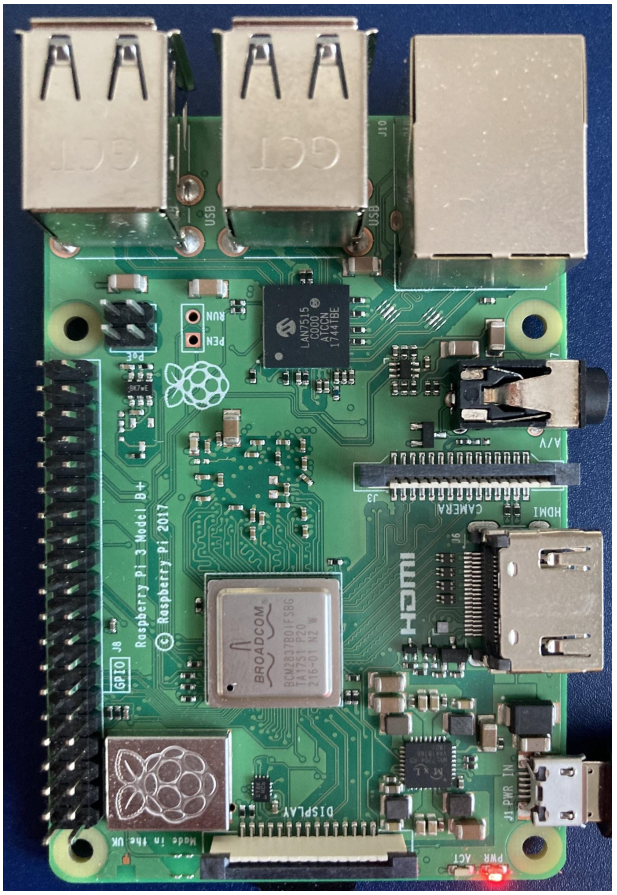
Ce projet peut être séparé en deux parties: une partie matérielle (hardware) et une partie logicielle (software). En raison de problèmes d'approvisionnement dû à la récente pandémie et de la complexité inattendue de la partie logicielle, la partie matérielle à été moins développée que ce que nous souhaitions.

Partie matérielle (Hardware):

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

Voir ci dessous deux photos de la Raspberry Pi, ainsi que deux photos avec la batterie.





Partie logicielle (Software):

La partie logicielle constitue en tout le code nécessaire au fonctionnement du jeu. Afin de conserver un historique des versions ainsi que pour pouvoir collaborer, nous avons utilisé le programme git (https://git-scm.com/).
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 sur votre ordinateur avec `git`:

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

Vous devriez maintenant avoir un dossier nommé `Waypoint` contenant le code source.

Voici une vue d'ensemble du projet:

```
~/Waypoint$ tree -a -L 1
Waypoint
├── Client
├── Server
├── Design
├── Statistics
└── .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). 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. Finalement, le dossier contient les scripts générant les graphiques de la section Quelques chiffres.

Server:

Ce dossier à la structure d'une application *.NET* classique. *.NET* est une plateforme de développement open source et gratuite, édité par Microsoft. Elle est basée sur le langage *C#*, un langage de programmation à usage général typé statiquement orienté objet.
Pour installer *.NET* sur Ubuntu, 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 VSCode, car le dossier `.vscode` contient la configuration nécessaire. Sinon, pour simplement lancer le projet, exécuté:

```
~/Waypoint/Server$ dotnet run
```

Et pour compiler le projet pour linux :

```
~/Waypoint/Server$ build GameServer.csproj -r linux-x64 -o bin/Linux/
```

Ou pour compiler le projet pour la Raspberry Pi:

```
~/Waypoint/Server$ build GameServer.csproj -r linux-arm -o bin/Raspberry/
```

Après avoir compilé le projet, n'oubliez pas de copier le dossier Scenes là où vous souhaitez exécuter le projet!

Regardons maintenant en détail les différents éléments de ce serveur:

```
~/Waypoint$ 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
```

Dans l'ordre, nous avons d'abord le dossier `.vscode`, qui contient la configuration de notre éditeur, VSCode, adapté à ce projet. `tasks.json` contient les instructions pour compiler le projet, `launch.json` celle pour le déboguer et `settings.json` nos différents réglages de l'éditeur.
Puis, le dossier `bin`, qui n'est pas inclus sur github, contient tous nos builds pour les différentes plateformes, en l'occurrence Linux et Raspberry.
Ensuite, le dossier `obj` est créé lors de la compilation et ne contient pas de fichiers importants.
Vient maintenant le dossier `Scenes`, qui contient une copie des terrains du client, au format `.unity`. Ce format s'appelle UnityYAML et, comme son nom l'indique, c'est le format des scènes Unity, basé sur le format YAML. Ces fichiers sont interprétés par le serveur via le script `Scene.cs`, qui contient un parser customisé pour ce format. Le dossier contient la salle d'attente, `WaitingRoom.unity` et toutes les cartes du jeu: `Map1.unity`, `Map2.unity`,...
Finalement, nous avons quelque fichiers:

- `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 dotnet 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` qui nous permet de parser du YAML.
- `.gitignore` contient une liste des fichiers ou dossiers que git 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
`Program.cs`*	Premier script appelé lors de l'exécution, `Program.cs` est responsable du démarrage du serveur et de la game loop dans deux différents threads. Il actualise la game loop à intervalles réguliers.
`ThreadManager.cs`*	Ce script gère simplement la communication entre le thread du serveur et la game loop (main thread).
`Server.cs`*	`Server.cs` contient la classe la plus importante du projet, la classe Server. 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.
`Client.cs`*	Ce fichier contient la classe Client, 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 Player. Chaque Client a un Player associé et est chargé de l'initialiser et de le réinitialiser quand le client se déconnecte.
`Packet.cs`*	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.
`ServerSend.cs`	Cette classe contient les fonctions responsables de la construction et de l'encodage des paquets envoyés par le serveur aux clients.
`ServerHandle.cs`	Cette classe contient les fonctions responsables du décodage et de l'interprétation des paquets envoyés par les clients au serveur.
`GameLogic.cs`	C'est ici que se situe la logique du jeu. Ce script contient la fonction Update, 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.
`Scene.cs`	Ce fichier est responsable de l'interprétation des scènes, qui se situe dans le dossier `Scenes`. Il est responsable de trouver certains objets dans les scènes, tel que les obstacles, les éléments déclencheur d'action (trigger) ainsi que les points d'apparition (spawn points).
`Player.cs`	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 leur interactions, tel que récupérer des armes ou tirer des projectiles.
`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 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: RectCollider et CircleCollider, représentant respectivement des rectangles et des cercles. Chaque entité du jeu possède un de ces collider. Chacune des classes contient une fonction permettant de vérifier qu'il n'intersecte pas avec un autre collider.
`Utilities.cs`	Cette classe contient divers fonction pratiques, liées notamment à la génération de nombres aléatoires, de la génération de position aléatoire spécifiques et du 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 défauts des joueurs.

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

Client:

Passons maintenant au deuxième grand dossier du projet, le dossier `Client`. Ce dernier contient l'entièreté du code source du client.
Ce dossier est un projet Unity. Unity est une plateforme de développement de jeux multi plateformes, aussi basé sur le langage C#. Le lien ci-dessous explique comment installer Unity: https://unity3d.com/get-unity/download
Ce projet utilise la version 2020.3.2f1, mais fonctionne peut-être sur d'autres versions.

Voici une vue d'ensemble du projet après l'avoir téléchargé:

```
~/Waypoint$ tree -a --dirsfirst Client
Client
├── Assets
│   ├── Animations
│   │   └── ...
│   ├── Fonts
│   │   └── ...
│   ├── Graphic
│   │   └── ...
│   ├── _Heathen Engineering
│   │   └── ...
│   ├── Hexanim
│   │   └── ...
│   ├── Materials
```

```
|   |   └── ...
|   ├── Prefabs
|   |   ├── EndScreen
|   |   |   ├── Bar.prefab
|   |   |   ├── Confettis.prefab
|   |   ├── Items
|   |   |   ├── Black Item.prefab
|   |   |   ├── Blue Item.prefab
|   |   |   ├── Bronze Item.prefab
|   |   |   ├── Brown Item.prefab
|   |   |   ├── Cyan Item.prefab
|   |   |   ├── Green Item.prefab
|   |   |   ├── Lime Item.prefab
|   |   |   ├── Orange Item.prefab
|   |   |   ├── Pink Item.prefab
|   |   |   ├── Purple Item.prefab
|   |   |   ├── Red Item.prefab
|   |   |   ├── Siilver Item.prefab
|   |   |   ├── White Item.prefab
|   |   |   └── Yellow Item.prefab
|   |   ├── Leaderboard
|   |   |   └── Entry.prefab
|   |   ├── Players
|   |   |   ├── LocalPlayer.prefab
|   |   |   ├── Player [Local].prefab
|   |   |   └── Player.prefab
|   |   ├── Projectiles
|   |   |   ├── EnnemyProjectileExplode.prefab
|   |   |   ├── EnnemyProjectile.prefab
|   |   |   ├── FriendlyProjectileExplode.prefab
|   |   |   ├── FriendlyProjectile.prefab
|   |   |   └── Turret.prefab
|   |   └── SpawnPart.prefab
|   ├── 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 ne contiennent que des métadonnées concernant les fichiers du même nom.

Pour commencer nous avons le dossier `.vscode`, qui, comme sur le serveur, contient la configuration notre éditeur. En l'occurrence, le fichier `settings.json` contient simplement une liste de fichier à ne pas afficher dans l'éditeur, pour simplifier l'affichage.
Puis, les dossiers `ProjectSettings` et `UserSettings` contiennent respectivement les réglages du projet et ceux de l'utilisateur.
Ensuite, 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.
Nous avons aussi le dossier `Scenes` qui contient tous les terrains du jeu, et doit être exactement le même que sur le serveur.
Finalement, nous avons le dossier `Scripts`, qui contient tous les scripts du projet.

Voici un tableau résumant leurs fonctions respectives:

Script	Fonction
`ThreadManager.cs`*	Ce script gère simplement la communication entre le thread du client et la game loop (main thread).
`Client.cs`*	Cette classe est responsable de la connexion avec le serveur. Elle gère le transfert de paquets via TCP et UDP.
`Packet.cs`*	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 entitées 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 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 scores.
`LeaderboardEntry.cs`	Cette classe est associée à chaque entrée du tableau des scores et stocke leur position et texte et est responsable de l'animation de leur position.

`EnsScreenManager.cs`	Ce fichier contient la classe responsable de l'écran apparaissant à la fin d'une partie
`EndScreenBar.cs`	Cette classe, rattachée aux bars apparaissant à la fin d'une partie, est responsable de leur mouvement et de leur apparence.
`DontDestroy.cs`	Ce script est très petit, et ne fait qu'indiquer au compilateur de ne pas détruire certains éléments qu'en une nouvelle carte est chargée.

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

Protocole:

Pour communiquer entre eux, le client et le serveur utilisent deux protocoles de base: TCP et UDP. Ils transfèrent des séquences d'octets, appelées paquets (packets). La signification d'un paquet dépend de son packet id, 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:

Field Name	Field Type	Note
length	int	Longueur du message. Est égale à la longueur du packet id + la longueur des donnée du packet
packet id	int	Type du paquet, voir les sections suivantes
data	bytes	Liste d'octet dépendant du packet id

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

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 conserver l'ordre des paquets. UDP est plus rapide que TCP, mais ne vérifie pas la réception ni l'ordre des paquets. Cependant, UDP est beaucoup plus rapide que TCP.
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 que nous avons remarqué est que certains réseaux, notamment celui de notre collège, semblent bloquer ce protocole. C'est pourquoi nous souhaitions ajouter à notre jeu un mode "TCP only", ce que nous n'avons malheureusement pas encore eu le temps de faire.

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 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 paquet de mouvement sont envoyés en un seul tic, le serveur n'applique que le dernier reçu. Cela permet de sécuriser le serveur.
La seule exception à cette règle est la rotation du joueur, que le client contrôle entièrement afin d'éviter un délai nettement visible.

Type de données

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

Name	Size (in bytes)	Encodes	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

Server Packets (envoyés du serveur au client):

Welcome:

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

Packet ID	UDP/ TCP	Field name	Field type	Note
1	TCP	message	string	Message de bienvenue, toujours défini comme "Welcome to the server!", utilisé uniquement pour débogage.

		client id	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	Field name	Field 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^o 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	Field name	Field type	Note
3	TCP	client id	int	ID du joueur

Set Name:

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

Packet ID	UDP/ TCP	Field name	Field 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	Field name	Field 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	Field name	Field type	Note
6	TCP	(aucun)		

Game Time:

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

Packet ID	UDP/ TCP	Field name	Field type	Note
7	UDP	time	float	Temps restant, en secondes

Player Position:

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

Packet ID	UDP/ TCP	Field name	Field 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	Field name	Field 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	Field name	Field type	Note
12	TCP	client id	int	ID du joueur

Player Hit:

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

Packet ID	UDP/ TCP	Field name	Field 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	Field name	Field type	Note
11	TCP	ammo	int	Nombre de balles du joueur

Item Spawned:

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

Packet ID	UDP/ TCP	Field name	Field 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	Field name	Field 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	Field name	Field 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	Field name	Field 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	Field name	Field type	Note
17	TCP	item id	int	ID du projectile

Client Packets (envoyés du serveur au client):

Welcome Received:

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

Packet ID	UDP/ TCP	Field name	Field type	Note
1	TCP	client id	int	ID du client

Player Name:

Envoyé au serveur lorsque le joueur change son nom

Packet ID	UDP/ TCP	Field name	Field type	Note
2	TCP	name	string	Nouveau nom du joueur

Player Movement:

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

Packet ID	UDP/ TCP	Field name	Field 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	Field name	Field type	Note
4	TCP	(aucun)		

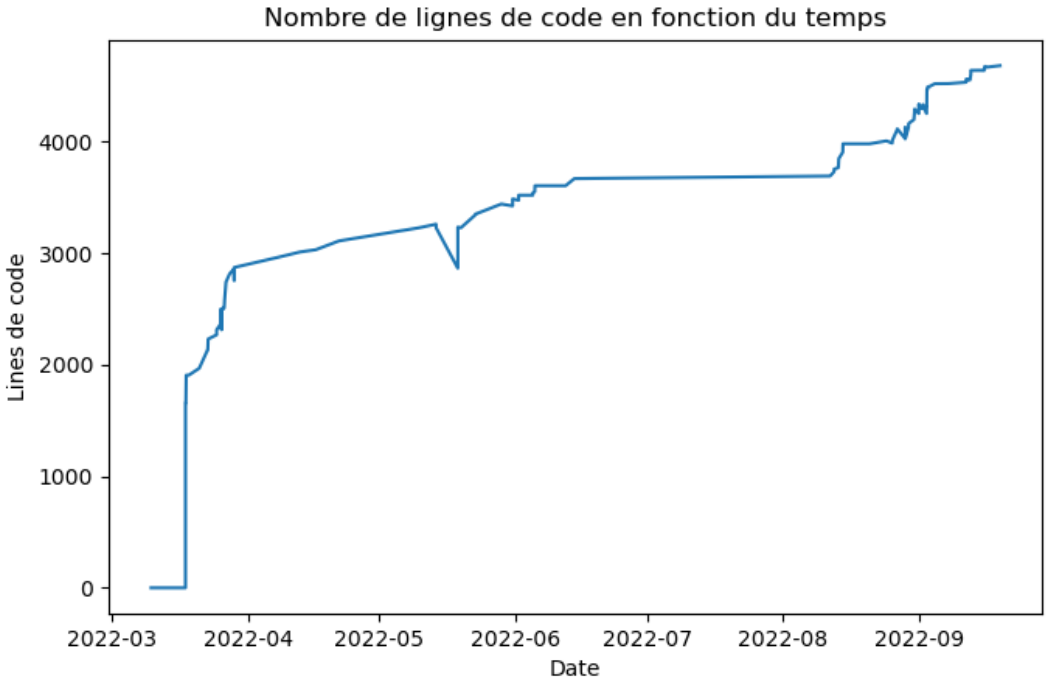
Player End Gamet:

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	Field name	Field type	Note
5	TCP	(aucun)		

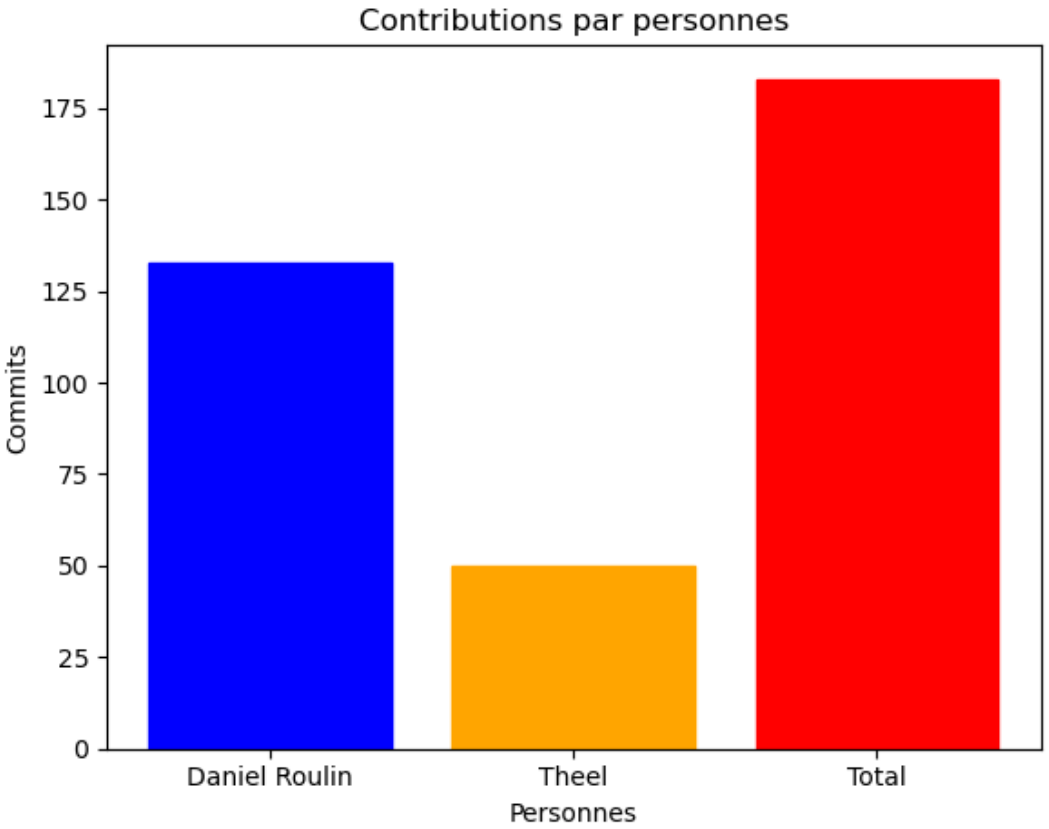
Quelques chiffres:

Lignes de code en fonction du temps



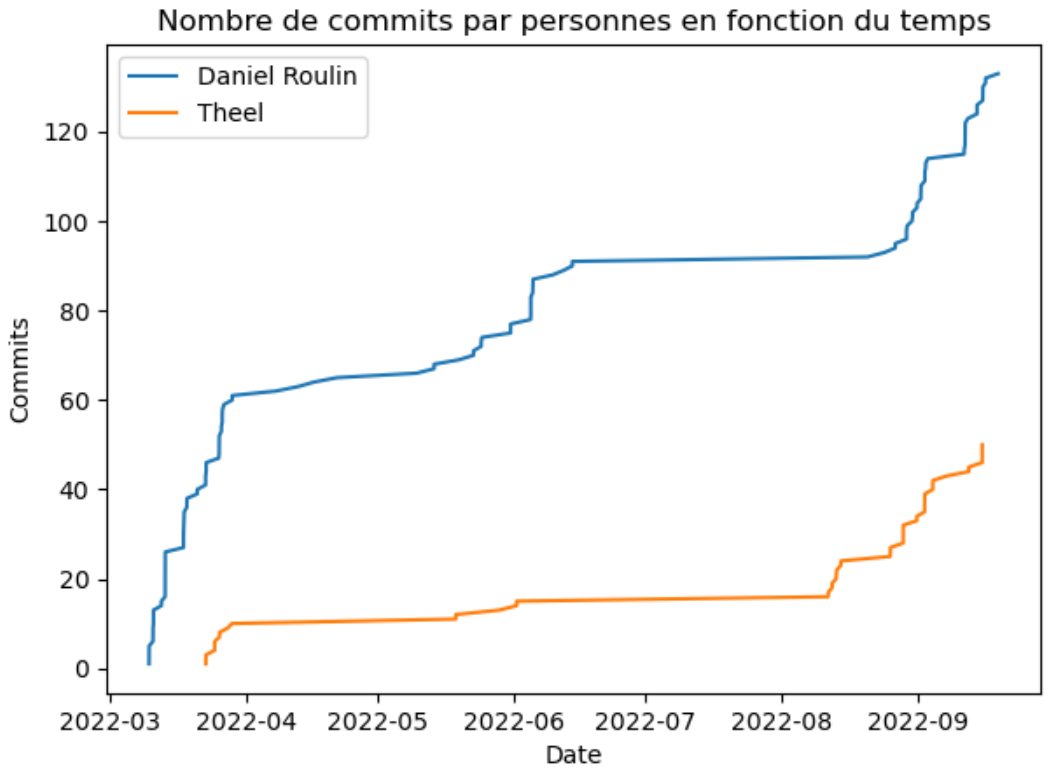
Ce graphique montre le nombre de lignes contenu dans les fichiers **.cs** (fichiers de code C#) en fonction du temps. Il a pu être fait en remontant dans l'historique des versions via Git. On constate très bien la réorganisation du projet mi mars 2022, suivi de l'ajout du code du tutoriel, puis l'adaptation du projet à nos besoin fin 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 ensuite voir les vacances, de juin à début mai. L'agitation fin mai correspond à de nouveau des conflits de merge, car nous travaillions beaucoup en même temps aux mêmes endroits du code. Finalement, les changements du mois d'octobre représente notre débogage, c'est pourquoi peu d'ajout de code peuvent être observés.

Contributions



Ce second graphique montre le nombre de commits par personne. A première vue, on pourrait penser que Fabio (Theel) à beaucoup moins contribué aux projets. Cependant, ce graphique montre les commits, qui ne sont pas représentatif du travail fourni. En effet, nous avons tous les deux différentes façons de commit. Par exemple, Daniel préfère créer un commit à chaque changement tandis que Fabio commit qu'en il a fini toute sa partie. De plus, Fabio s'est aussi concentré sur la partie artistique, qui n'est pas mesurée par les commits.

Commits



Finalement, ce graphique montre l'évolution du nombre de commits en fonction du temps. On y voit clairement nos deux différents styles de commits, 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 démarche). C'est pourquoi les premiers commits n'ont été faits uniquement par Daniel et commencent plus tôt.