Université de Franche-Comté

Rapport de projet tuteuré
Licence informatique 3ème Année

Création d'un Third-Person Shooter avec le Shine Engine

Virgil Manrique Quentin Guillien

Encadrant :
Sylvain Grosdemouge

Année 2015-2016

Remerciements

Ce projet tuteuré n'aurait pu être réalisé sans l'aide de plusieurs personnes que nous tenons à remercier.

Tout d'abord, nous remercions notre encadrant M. Sylvain Grosdemouge pour nous avoir guidé et prodigué ses conseils durant la réalisation de ce projet.

Ensuite, nous voulons remercier M. Bastien Schatt, pour nous avoir aidés lors de notre apprentissage de l'utilisation du Shine Engine ainsi que pour sa réactivité lorsque nous avons rencontré des problèmes.

Nous voulons également remercier M. Nicolas Diot, pour son assistance apportée par rapport au fonctionnement de l'éditeur de niveaux du Shine Engine et des ressources graphiques.

Nous tenons enfin à remercier toutes les personnes qui nous ont aidés de près ou de loin.

Table des matières

1 Cadre du projet

Dans le cadre de notre 3ème année de licence informatique à l'université de Franche-Comté, il nous a été demandé de réaliser un projet dans le cadre du module projet tuteuré. La durée du projet s'étendait d'Octobre à Mars.

Parmis les sujets proposés, l'un d'entre eux a particulièrement retenu notre attention. Il s'agissait du développement d'un Third-Person Shooter en utilisant un moteur de jeu : le `Shine Engine` développé par M. Sylvain Grosdemouge.

Nous avons pu nous voir attribuer ce sujet l'ayant placé en première position dans notre liste de choix.

Pour comprendre l'intérêt que nous portions à ce sujet, il faut nous pencher sur les différents éléments qui le constituent, à savoir les Third-Person Shooter, les moteurs de jeu, et le `Shine Engine`.

1.1 Les Third-Person Shooter

Un TPS (Third-Person Shooter ou jeu de tir à la troisième personne en français) est un sous-genre des jeux de tir et donc des jeux d'action. Les jeux de tir mettent souvent la rapidité et la réactivité du joueur à l'épreuve. L'objectif de ce genre de jeu est de vaincre ses ennemis en utilisant une arme de tir. La particularité des `TPS`[1] est que le joueur voit son personnage de manière externe contrairement aux `FPS`[2] où le joueur voit à travers les yeux de son personnage. (Voir Figure 1.1 page 6)

1. **T**hird-**P**erson **S**hooter
2. **F**irst-**P**erson **S**hooter : Jeu de tir à la 1ère personne

1.2 Les moteurs de jeu

"Un moteur de jeu est un ensemble de composants logiciels qui effectuent des calculs de géométrie et de physique utilisés dans les jeux vidéo. L'ensemble forme un simulateur en temps réel souple qui reproduit les caractéristiques des mondes imaginaires dans lesquels se déroulent les jeux. Le but visé par un moteur de jeu est de permettre à une équipe de développement de se concentrer sur le contenu et le déroulement du jeu plutôt que la résolution de problèmes informatiques."

– *Définition Wikipédia*

1.3 Le Shine Engine

Le `Shine Engine` est un moteur de jeu créé par notre encadrant, Monsieur Sylvain Grosdemouge. Monsieur Grosdemouge a commencé le développement du `Shine Engine` en 2005 et en 2012 est sorti R.A.W.(Realm Of Ancient War), premier jeu développé en utilisant le `Shine Engine`. Ce moteur de jeu est développé en C++ et permet de faire du développement multi-plateforme. Il permet aussi de gérer facilement la 3D ce qui permet de s'affranchir de beaucoup de limites pour le développement d'un jeu. De plus le moteur fonctionne avec un éditeur de niveau, le Shine Game Editor, qui facilement grandement la création de niveaux. (Voir Figure 1.2 page 6)

1.4 Le Projet

Les trois éléments cités précédemment rendaient pour nous le projet attrayant : les TPS sont un genre de jeu au concept simple mais distrayant ; utiliser un moteur de jeu permet de se concentrer sur les mécaniques de jeu, qui sont pour nous la partie la plus intéressante du développement d'un jeu ; et enfin, le `Shine Engine`, un moteur qui a fait ses preuves, qui est efficace et qui nous permet de développer en C++, le langage que nous préférons.

FIGURE 1.1 – Un exemple de `TPS` : Resident Evil 4

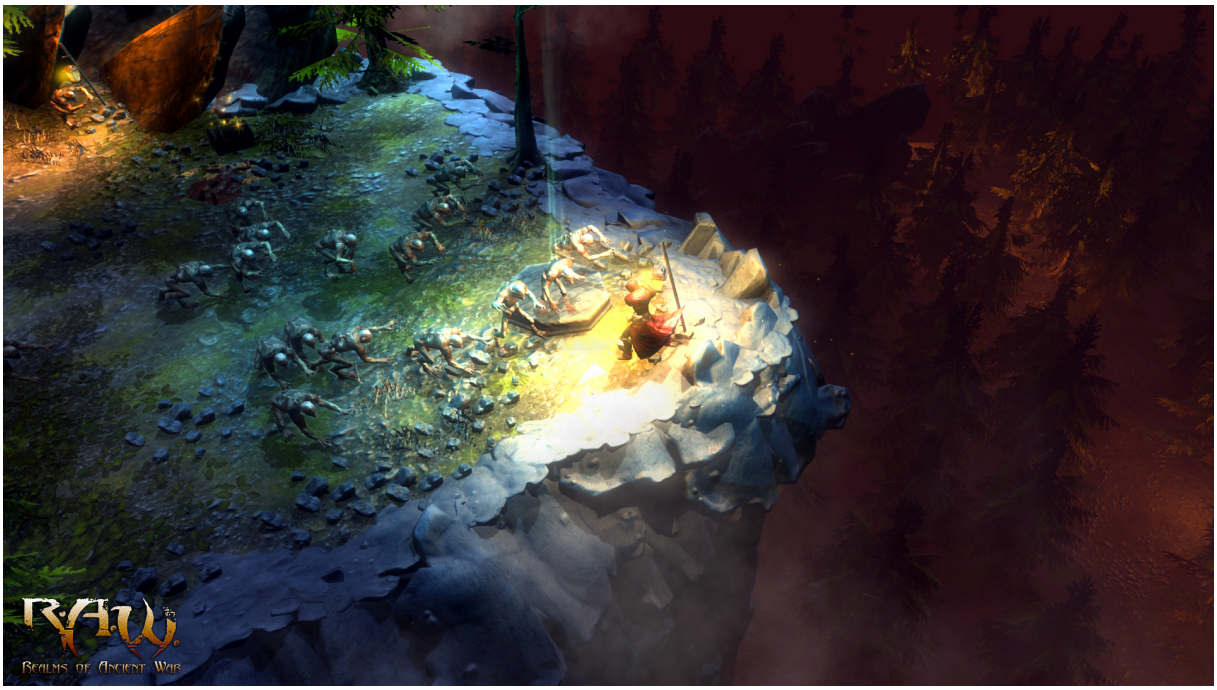

FIGURE 1.2 – REALM OF ANCIENT WAR, un jeu réalisé avec le `Shine Engine`

2 Préparation

2.1 Installation des outils

Avant de commencer à coder, il a d'abord fallu installer plusieurs outils logiciels.

Tous les outils mentionnés ont été utilisés, dans notre cas, sur Windows uniquement.

2.1.1 Les indispensables

Les trois logiciels indispensables sont le `Shine SDK`[1], le `Shine Editor`[2] et `Microsoft Visual Studio 2010`.

Le `Shine Editor` et le `SDK` ne requérant pas d'installation, n'ont pas posé de problèmes dans un premier temps. En revanche, il n'a pas été facile de retrouver la version 2010 de `Visual Studio`, car aujourd'hui, Microsoft ne propose que la version *Community* de leur logiciel. `Visual Studio` est un `IDE`[3]. Il en existe d'autres, mais le `Shine Engine` a été développé et prévu pour être intégré à `Visual Studio`.

Même après avoir installé les outils `Shine`, il ne peuvent pas encore être exécutés indépendamment. En effet, ils requièrent le `DirectX SDK` mais aussi `Microsoft .NET Framework` (version 4.0 ou supérieur).

2.1.2 Pour plus de confort

Bien qu'ils ne soient pas réellement indispensables, les outils suivants nous été extrêmement utiles.

Nous avons utilisé `Git`, plus précisément `Github`, l'application graphique de Git pour Windows. `GitHub` est un service web d'hébergement et de gestion de développement de logiciels utilisant Git, le logiciel de gestion de versions décentralisé.

Nous avons également utilisé `Trello`, un outil de gestion de projet en ligne permettant d'assigner facilement des tâches à des utilisateurs.

1. SDK = Software Development Kit (trousse de développement logiciel en Français)
2. Editeur graphique de Shine permettant d'ajouter facilement des éléments visuels dans un jeu
3. IDE = Integrated Development Environment (Environnement de Développement en Français)

GIMP [4] est un outil d'édition et de retouche d'image. GIMP a été utile pour créer des `sprite` [5] simples.

En complément de GIMP nous avons utilisé `XnView`, qui nous a permis de vérifier le formats de certains fichiers, et de convertir au bon format si besoin.

Enfin, nous avons utilisé LaTeX pour la rédaction de ce rapport.

2.2 Formation au Shine Engine

Pour apprendre à utiliser le moteur, nous avons assisté aux formations au `Shine Engine` dispensées par `Shine Research` au Cub' à Essais, à l'incubateur d'entreprises innovantes de Franche-Comté, tous les jeudis à 17h pendant 10 semaines. Ces formations nous ont été utiles pour apprendre les bases du fonctionnement du `Shine Engine` et ainsi pouvoir maitriser par nous-même des fonctionnalités plus avancées.

4. GNU Image Manipulation Program

5. Elément graphique qui peut se déplacer sur l'écran. Dans notre cas, cet élément était une simple image en deux dimensions

3 Réalisation

Dans ce chapitre nous allons expliquer chaque élément constituant le jeu, que ce soit le joueur ou les ennemis, les armes et leurs munitions ou bien la gestion des collisions.

3.1 Jeu ou Plugin ?

Pour réaliser ce projet, notre encadrant nous a demandé de développer le jeu non pas de manière classique mais en utilisant la fonctionnalité de `plugin` du `Shine Engine`. Nous avons donc développé les mécaniques de jeu en gardant en tête que la fonction de `plugin` permettait d'utiliser le plugin simplement en l'activant lors de la création d'un niveau, par exemple en utilisant l'éditeur de Shine, ce qui nous a poussé à développer de manière générique et non pas en fonction du niveau.

Monsieur Grosdemouge nous a donné des `sprite` simples pour faire un niveau de test mais nous avons par la suite réalisé d'autres `sprite` pour faire d'autres niveaux afin de vérifier le bon fonctionnement du `plugin`.

3.2 Éléments du plugin

3.2.1 Les personnages

Le `plugin` doit pouvoir gérer deux types de personnages : le personnage du joueur, contrôlé par ce dernier et le ou les personnages ennemis, qui disposent de leur propre IA [1]. Pour cela nous avons crée une classe générique `Character` de laquelle héritent les classes `Player` et `Enemy`.

Cette classe a pour grande utilité de permettre un traitement générique ainsi que la possibilité d'une modification générique. Ainsi, lorsque nous avons voulu implémenter la gestion de la 3D, il nous a suffit d'ajouter un attribut à cette classe pour stocker le modèle 3D. Le joueur et les ennemis ont donc été dotés de cet attribut.

Un personnage peut posséder une arme et tirer avec celle-ci, nous détaillerons cela plus en détail dans la partie consacrée à la classe associée aux **armes**, à savoir la classe

1. **I**ntelligence **A**rtificielle

`Gun`. Un personnage peut aussi mourir s'il entre en contact avec un projectile, exception faite des projectiles qu'il a tiré lui-même.

3.2.2 Le Joueur

Pour bien différencier le joueur des ennemis, et puisque les deux n'ont pas le même fonctionnement, nous avons crée une classe `Player` qui hérite de la classe `Character`, la rendant donc similaire à la classe `Enemy` mais dénué d'IA et contrôlé à l'aide des touches du clavier.

Le joueur peut ainsi faire avancer son avatar avec la flèche haut ou la touche Z. Les touches flèche gauche (ou la touche Q) et flèche droite (ou la touche D) permettent d'effectuer une rotation respectivement vers la gauche ou la droite. Le joueur peut aussi reculer en utilisant la touche flèche bas (ou la touche S) mais se déplace dans ce cas à la moitié de sa vitesse.

Initialement, notre encadrant ne nous avait pas demandé de permettre au joueur de reculer mais nous avons rajouté cette fonctionnalité pour un meilleur confort de jeu après avoir fait quelques test lors de l'implémentation des projectiles et de la possibilité pour l'avatar de mourir [2].

3.2.3 Les Ennemis

La classe `Enemy` permet de représenter un ennemi auquel le joueur peut-être confronté. Le `plugin` gère une liste d'ennemis pour représenter tous les ennemis d'un niveau. La classe `Enemy` héritant de la classe `Character`, un `Enemy` est sensiblement identique au joueur mais à un fonctionnement différent : il est contrôlé par une IA.

L'IA ne fut pas simple à mettre au point bien qu'elle ait elle-même un fonctionnement simple : si l'ennemi ne voit pas le joueur, il reste sur place, si il le voit, il se déplace vers lui tout en tirant. L'IA est basée sur un automate à deux états, l'état `attaque` et l'état `repos`. L'état `repos` est son état par défaut et il passe en état `attaque` lorsqu'il voit le joueur.

L'ennemi possède un attribut `Target` qui représente la dernière position connue du joueur. A chaque `update` du jeu [3], le gestionaire de collisions va vérifier si l'ennemi a une ligne de vue directe sur le joueur. Si c'est le cas, il met à jour l'attribut `Target` de l'ennemi. Ce faisant, l'ennemi va passer en état `attaque` s'il ne l'est pas déjà et va se rendre jusqu'à la position cible tout en tirant en direction de celle-ci. Si, en cours de chemin, il perd sa ligne de vue directe avec le joueur, il se rendra tout de même à sa

2. *Il est difficile d'esquiver les projectile si on ne peut pas revenir rapidement sur ses pas*
3. Pour chaque image rendu du jeu, on effectue une `update`, c'est-à-dire qu'on met à jour tous les éléments qui composent le jeu (leur position, leur direction, leur état...)

dernière position connue, ce qui peut lui permettre de voir à nouveau le joueur. Nous avons doté les ennemis d'une vision à 360 degrés pour pouvoir repérer le joueur.

3.2.4 Les Armes

Déjà évoquée plus haut, la classe `Gun` est la classe représentant les armes dans le `plugin`. Une arme est détenue par un personnage et possède plusieurs attributs notables.

Elle dispose d'un *nom* ; d'un *chargeur* contenant des munitions (6 par défaut) ; une *puissance*, qui correspond à la vitesse de déplacement donnée aux balles tirées par l'arme ; ainsi qu'une *cadence de tir* et son *temps de rechargement* associé. Le nom n'est actuellement pas utilisé mais pourrait servir pour d'éventuelles améliorations futures. Sa cadence de tir permet de s'assurer que le joueur ou les ennemis peuvent tirer plus ou moins vite en fonction de l'arme qu'ils possèdent. Lors du lancement du `plugin`, l'arme va créer ses munitions de départ et les stocker dans son chargeur, prêtes à être utilisées.

Une arme dispose d'une fonction `Shoot`, utilisée lors de l'appel à la fonction du même nom du personnage qui possède l'arme, et qui renvoie une munition, prête à se déplacer dans le niveau. Lors de l'appel à cet fonction, la munition ainsi préparée sera alors placée dans la liste de munitions actuellement en mouvement dans le niveau et retirée du chargeur de l'arme, empêchant sa réutilisation avant qu'elle n'ait rencontré un obstacle.

3.2.5 Les Munitions

La classe `Ammo` permet de représenter les munitions. Ces munitions possèdent divers attributs : une *position*, correspondant à leur position dans le niveau ; une *direction*, correspondant au sens dans lequel elles se déplacent ; une *vitesse*, qui correspond à leur vitesse de déplacement ; un *sprite*, si le niveau est en 2D ; un *modèle*, si le niveau est en 3D ; un booléen *Moving* qui permet de savoir si elle sont en cours de déplacement ou pas ; une *origine*, qui désigne quel personnage a tiré la munition ; et un booléen qui indique si la munition est en *3D* ou pas.

La position et la direction de la munition sont données à celle-ci lorsqu'elle est tirée par le joueur, ladite munition apparait au bout du `sprite` du joueur et se dirige dans la direction où il regarde au moment où il a tiré, puis se déplace normalement, indépendamment du joueur. La vitesse est donnée par l'arme au moment du tir. Le booléen `Moving` permet de savoir si la munition a rencontré un obstacle et donc si elle peut être remise dans le chargeur. L'origine de la munition est le personnage qui a tiré celle-ci, cela permet au personnage de ne pas se faire tuer par son propre projectile.

3.2.6 La Caméra

La classe `Camera` est la classe permettant de gérer la caméra du jeu et donc la vue que peut avoir le joueur sur le niveau.

Cette classe fut plutôt simple à faire grâce au `Shine Engine` qui dispose déjà d'une classe permettant de gérer une caméra. Notre caméra se contente d'encapsuler celle du `Shine Engine` en créant toutefois quelques fonctions qui permettent un positionnement simple de ladite caméra pour correspondre à l'utilité dont peut en faire un TPS.

La caméra a 3 modes : un mode *vue de dessus*, qui suit le joueur quand il se déplace tout en permettant d'avoir une bonne vision de ce qui se trouve autour de lui, pratique pour les niveaux en 2D ; un *mode de vue TPS*, qui se place derrière le joueur ; et un *mode de vue haute*, qui est proche de la vue du dessus mais dans lequel le joueur reste immobile et c'est le monde qui défile par rapport à lui. On peut passer d'un mode à l'autre grâce à la touche C. Il y a aussi possibilité d'augmenter ou de diminuer le champs de vision [4] lorsqu'on est en mode caméra TPS avec les touches + et - .

3.2.7 Le Gestionnaire de collisions

Le gestionnaire de collisions est représenté grâce à la classe `CollisionsManager`. C'est cette classe qui va gérer toutes les collisions du `plugin`, que ce soit les personnages et les munitions, les munitions et les murs ou bien le champs de vision de ennemis.

La classe possède deux attributs : une liste de `CollisionShape` et le nombre de `CollisionShape` contenus dans cette liste. Une explication des `CollisionsShape` et de la gestion des collisions dans le `plugin` est donnée dans la **prochaine partie**, consacrée à la gestion des collisions.

4. En anglais : **FOV** : **F**ield **O**f **V**iew

3.3 Mécaniques de jeu

Dans cette partie nous allons expliquer le fonctionnement de la gestion des collisions dans le `plugin` ainsi que le fonctionnement des mécaniques de jeu du `plugin`.

3.3.1 Gestion des collisions

La partie de gestion des collisions nous fut grandement simplifiée grâce au `Shine Engine` et notamment grâce à 3 fonctionnalités de celui-ci :

— Les `CollisionShape` : il s'agit de sortes de segments qui servent dans le moteur notamment pour faire office de murs physiques. Ils peuvent également être placés directement depuis l'éditeur, ce qui en fait un outil très pratique pour les collisions avec le décor.
— Le `Character Controller` : il s'agit d'une classe qui intègre nativement les collisions entre elle et les `CollisionShape` ainsi qu'entre les différentes instances de la classe, empêchant ainsi un `Character Controller` de passer à travers un autre `Character Controller` ou à travers un `CollisionShape`.
— La fonction `Intersect` pour les entités 2D : il s'agit d'une fonction qui permet de savoir si deux entités 2D, c'est-à-dire des `sprite`, se croisent.

Grâce à ces 3 éléments, peu de travail sur les collisions restaient à faire. Nous avons doté la classe `Character` d'un attribut de type `Character Controller` et à partir de là, aucun personnage ne pouvait passer à travers un mur (représenté par un `CollisionShape`), ou à travers un autre personnage. Pour gérer la collision entre les munitions se déplaçant dans le monde et un personnage, la fonction `Intersect` est utilisée pour voir si les deux `sprite` se touchent ou se croisent.

Il nous restait donc à gérer les collisions entre les munitions et les `CollisionShape` ainsi que les lignes de vue des ennemis.

De ce fait pour gérer ces collisions le gestionnaire de collisions procède ainsi :

— Pour les munitions et les `CollisionShape`, il vérifie si il y a intersection entre le `CollisionShape` et la trajectoire de la balle entre sa position actuelle et sa position après déplacement.
— Pour les lignes de vue entre les ennemis et le joueur, pour chaque ennemi, le gestionnaire vérifie si il y a un `CollisionShape` (un mur) entre le joueur et l'ennemi. Si ce n'est pas le cas alors l'ennemi a une ligne de vue directe sur le joueur et peut donc passer en état `attaque`. [5]

5. Pour plus de détails concernant le comportement des ennemis, reportez-vous à la **section correspondante** (page 10)

Pour les collisions en 3D, et par souci de simplicité pour le développement du projet, le gestionnaire va utiliser les `sprite` 2D associés à chaque élément. Pour comprendre comment il peut faire cela, intéressons-nous à la prochaine partie, consacrée au fonctionnement du `plugin`.

3.3.2 Fonctionnement du `plugin`

Maintenant que nous avons vu les divers éléments constituant le `plugin` ainsi que la gestion des collisions, nous pouvons nous intéresser au fonctionnement du `plugin`.

Le `plugin` fonctionne en deux temps. Tout d'abord lors de l'appel à la fonction `OnPlayStart` du `Shine Engine`, puis lors de l'appel à la fonction `OnPostUpdate` du moteur de jeu. De plus le `plugin` est conçu pour fonctionner avec des niveaux en 2D ou bien en 3D.

La fonction `OnPlayStart` est appelée lorsque un niveau est débuté. Dans l'éditeur cela correspond aussi à l'appui sur le bouton `Play`, qui sert à tester le niveau créé avec l'aide de l'éditeur.

Lors de l'appel à la fonction `OnPlayStart`, le `plugin` va initialiser tout ce qu'il peut en fonction du niveau, il va donc initialiser le joueur en fonction de la position du `sprite` ou du modèle de celui-ci, en sachant qu'il faut respecter une convention de nommage propre du nom donné au joueur dans l'éditeur pour que le `plugin` puisse le trouver. Puis le `plugin` va créer et initialiser autant d'ennemis qu'il trouve de `sprite` ou de modèles respectant la convention de nommage.

Il est à noter que si le `plugin` trouve des modèles 3D, il va charger des `sprite` 2D depuis la bibliothèque et créer les entités 2D associées afin de pouvoir gérer les collisions. Ce fonctionnement pourrait poser problème pour un vrai jeu 3D mais nous avons conçu le `plugin` comme cela pour le projet pour un souci de simplicité.

La fonction `OnPostUpdate` est appelée par le moteur de jeu après sa propre `update`. C'est dans cette fonction que le `plugin` se met à jour.

A chaque appel à la fonction `OnPostUpdate`, le `plugin` se met à jour. Il va d'abord vérifier l'appui sur les différents inputs du joueur pour faire réagir l'avatar de celui-ci ou la caméra en conséquence. Il va ensuite faire appel au gestionnaire de collisions pour s'occuper des lignes de vue de chaque ennemi ainsi que les éventuels tirs de ceux-ci ; puis il va, toujours en faisant appel au gestionnaire de collisions, vérifier les collisions des munitions avec les différents éléments de l'environnement, personnages inclus ; enfin il appelle succinctement les différentes fonctions `update` des différents éléments du niveau (*joueur*, *ennemis* et *munitions*).

3.4 Problèmes rencontrés

3.4.1 Problèmes logiciels

Nous l'avons mentionné dans la partie **Installation** (page 7), trouver la version complète de `Microsoft Visual Studio 2010` n'a pas été tâche facile.

Cependant, cela n'a pas été le plus important des problèmes logiciel que nous avons eu. En effet, il nous est arrivé par exemple de ne plus pouvoir lancer l'éditeur de `Shine` alors qu'il marchait auparavant, sans savoir pourquoi. Réinstaller l'éditeur, ou installer une version plus récente résolvait en général les problèmes rencontrés. Il nous est également arrivé de simplement ne pas du tout pouvoir faire fonctionner le `Shine Editor` sur un PC (la seule option était ici d'utiliser un autre PC).

3.4.2 Problèmes de code

Le problème majeur qui nous a couté le plus de temps est celui-ci :

```
1   /// @todo comment
```

Cette ligne, combinée à l'inexistence de documentation, nous a souvent laissé dans le flou quand à la manière de procéder pour effectuer des tâches parfois assez simples. La seule chose dont on pouvait s'aider était les fichiers d'entête dans les dossier du `Shine SDK` : en lisant les définitions des méthodes et les noms des paramètres, on pouvait au mieux *deviner* la façon d'utiliser ces méthodes.

Bien sûr, quand on ne pouvait plus avancer, nous avons pris contact avec l'équipe de `Shine`. Cependant, la réponse n'étant évidemment pas immédiate, il nous est arrivé de rester bloquer des après-midis entiers à essayer de faire marcher un bout de code.

4 Bilan

4.0.1 Optimisations Possibles

Ennemis avec cône de vision Améliorations d'armes liés au nom de l'arme Gestion des collisions 3D avec hitbox 3D tout ça. Minimap comme demandé par Grosdemouge.