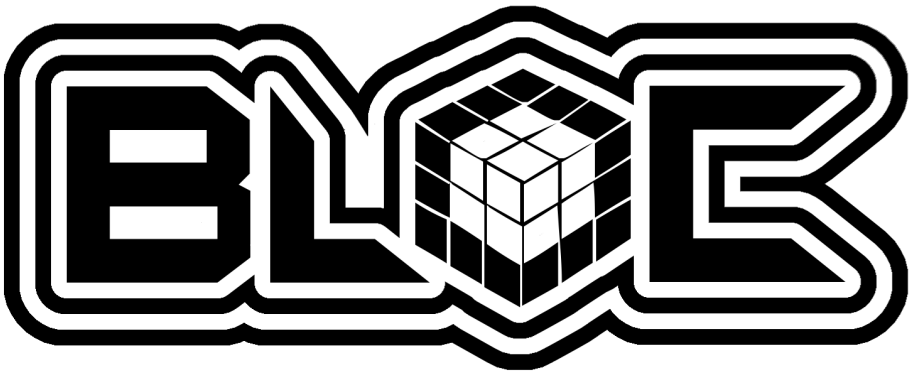

Rapport de projet « The ByteCoder »

Jeu de plate-formes en deux dimensions

BLOC Software

Epita

Pierre « BLUESCREEN »Boulay boulay_p

Serge « Kh4L »Panev panev_s

Paul-Henry « Dr_c0w »Perrissel perris_p

Alessandro « d0t_iKs »Pisu pisu_a

2013/2014

Sommaire

1 Introduction **4**

2 La structure de rendu **6**
2.1 Rendu graphique des différents éléments 6
2.2 Ajout et suppression des objets 7

3 Les niveaux **9**
3.1 Gestions des collisions 9
3.2 La caméra 11
3.3 Le scrolling background 11

4 Les ennemis **13**
4.1 Le modèle 13
4.2 La classe de l'ennemi 14
4.3 Les mécanismes de l'ennemi 14
4.4 Le comportement des ennemis en multijoueur 15

5 L'événementiel **16**
5.1 L'interpréteur de scripts 16
5.2 Objets scriptés 17
5.3 Les événements 19

6 Le joueur **20**
6.1 L'attaque à distance 20
6.2 L'attaque en saut 21
6.3 Le « dash »ou la glissade 22
6.4 La lance 23
6.5 La gestion des contrôles 23

7 Le multijoueur **24**
7.1 Le multijoueur local 24
7.1.1 Les entrées utilisateur 24
7.1.2 Le partage de l'écran 25
7.2 Le réseau 26
7.3 Le broadcast UDP 28
7.3.1 L'implémentation partie serveur 28
7.3.2 L'implémentation partie client 29

8 Les effets **30**
8.1 Les particules 30
8.2 L'effet d'apparition 30
8.3 L'effet de mort 31
8.4 Le feu 31

8.4.1 FireParticle . . . 31
8.4.2 LittleFire . . . 32
8.4.3 FireManager . . . 32
8.4.4 L'effet d'hyperespace . . . 33
8.5 Effet du menu de mort . . . 34
8.6 Effet du menu principal . . . 34
8.7 L'effet d'interférence magnétique . . . 35

9 L'univers **36**
9.1 Le scénario . . . 36
9.2 Les personnages non joueurs, non hostiles . . . 36
9.2.1 Les PNJ dans l'aventure . . . 36
9.2.2 Le fonctionnement des PNJ . . . 37
9.3 L'univers Graphique . . . 37
9.3.1 Le monde « Western » . . . 37
9.3.2 Le monde de glace . . . 38
9.3.3 L'usine . . . 38
9.4 Bug Brother . . . 39
9.5 Le Blob . . . 40
9.6 La mine de proximité . . . 40
9.7 La grenade . . . 41
9.8 Apache . . . 41
9.9 Le lance-missiles . . . 41

10 Le son **42**

11 L'interface utilisateur **43**
11.1 Menu . . . 43
11.1.1 Les bases . . . 43
11.1.2 Menu de sélection des touches . . . 45
11.1.3 Menu de configuration graphique . . . 45
11.1.4 Menu multijoueur . . . 46
11.1.5 Menu de sélection de niveau . . . 46
11.1.6 Menu de sélection d'un serveur . . . 46
11.2 Le HUD . . . 47

12 Les outils de développement **48**
12.1 La console en jeu . . . 48
12.2 L'éditeur de niveau . . . 48
12.2.1 L'interface . . . 48
12.2.2 Son fonctionnement . . . 49
12.3 Les outils de débogage . . . 49
12.4 Les couleurs TSL . . . 50
12.5 Le système de traduction . . . 50
12.6 Optimisation des performances . . . 51

13 La gestion des ressources **51**
13.1 L'importance de l'instanciation 52

14 Communication **53**
14.1 Le site web 53
14.2 Les réseaux sociaux 53

15 Conclusion **54**

16 Conclusions Personnelles **55**

17 Webographie **60**

1 Introduction

Ce rapport a pour but de présenter le projet « The ByteCoder ». C'est un jeu de plateformes en deux dimensions que nous avons développé dans le cadre d'un projet libre de notre première année à l'EPITA. Nous avons eu un temps de travail total de six mois. Durant ces six mois nous avons eu trois présentations évaluées par un jury.

Le nom de notre équipe, « BLOC », est l'acronyme de « Best Lines Of Code ». Comme notre nom l'indique, le développement d'un projet est pour nous l'occasion de nous exprimer, de partager nos idées et nos connaissances afin d'aboutir à un résultat auquel il nous plaira de jouer. Le héros de ce jeu est le ByteCoder qui se voit confier comme mission sacrée de rétablir la stabilité d'un système d'exploitation. Son voyage à travers les différentes couches du système va l'emmener dans des mondes tous plus impressionnants les uns que les autres, jusqu'à atteindre le noyau du système.

L'objectif de ce projet était, dans un premier temps, d'apprendre la conception d'un jeu vidéo et le C#, puis dans un second temps, d'apprendre à travailler en équipe et à s'organiser pour optimiser nos performances ainsi que nos temps de travail. Avant de commencer le développement de ce projet, nous avons dû définir nos méthodes de travail, la première chose à faire était d'utiliser les mêmes logiciels. Nous avons choisi de tous utiliser Microsoft Visual Studio 2013 pour écrire du code, et d'utiliser Microsoft Team Foundation Server comme logiciel de gestion de versions. Pour rester synchronisés au niveau du programme lui-même, nous avons décidé d'écrire le code de ce projet selon les quatre règles suivantes :

- Sécurité : le code doit gérer les erreurs, rester stable et prévenir toute défaillance du programme.
- Flexibilité : les composants que nous créons doivent rester flexibles et configurables, pour ne pas avoir à réécrire ce qui a déjà été fait.
- Performance : le code doit s'exécuter rapidement en utilisant le moins de ressources de la machine possible.
- Clarté : le code doit être clair, commenté, indenté et rapide à comprendre pour faciliter la relecture et le débogage.

L'entraide et la coordination ont été nos moteurs pour mener à bien ce projet. Ce rapport présente notre jeu sous trois aspects. D'abord le point de vue de l'utilisateur, c'est-à-dire les mécaniques de jeu, l'histoire, l'environnent graphique, les ennemis, les

objets, et les niveaux puis la partie technique qui explique les algorithmes, les méthodes et les bibliothèques que nous avons utilisées et enfin la partie communication présente notre site web ainsi que notre présence sur les réseaux sociaux. Il faut néanmoins garder à l'esprit que le développement d'un jeu vidéo nécessite plus que du code. En effet, un jeu vidéo a besoin de textures graphiques et d'effets sonores. Les images sont toutes de notre création. Les effets sonores proviennent quant à eux d'un de nos amis qui a bien voulu nous aider en composant des musiques spécifiques à ce projet.

2 La structure de rendu

2.1 Rendu graphique des différents éléments

The ByteCoder est un jeu de plateformes en deux dimensions. Néanmoins, cela ne signifie pas que le rendu graphique du jeu ne doit gérer que deux dimensions. En effet, les objets du jeu sont dessinés à l'écran les uns après les autres. Ce qui implique que l'objet qui est affiché en premier sera à l'arrière-plan, et donc l'élément dessiné en dernier sera au premier plan. La structure de rendu du jeu doit donc permettre de définir l'ordre d'affichage des objets.

Mais ce n'est pas la seule exigence que la structure de rendu doit satisfaire. Il existe plusieurs méthodes pour afficher un objet à l'écran. En effet, il peut être dessiné soit simplement, soit à partir d'une moyenne de couleurs. La moyenne de couleurs permet de réaliser certains effets graphiques tels que le feu, les étincelles et les particules. La difficulté ici est de réaliser un rendu graphique en utilisant le moins de ressources possible, car il est très coûteux en termes de performances de passer d'un mode de dessin « normal »à un mode « moyenne de couleurs ». Ainsi, notre structure de rendu se présente de la manière suivante :

Chaque objet qui apparaît à l'écran doit hériter de la classe GameObject. Cette classe possède les membres suivants :

- Le constructeur, qui prend en paramètre deux booléens, le premier définit si l'objet peut prendre des coups, le deuxième définit si l'objet doit être dessiné en mode « moyenne de couleurs »ou non.
- La fonction Next() : la fonction qui met l'objet à jour avant chaque rendu, cette fonction retourne un booléen. Dès que cette fonction retourne faux, l'objet est détruit et il ne sera plus mis à jour ni dessiné.
- La fonction Draw() : la fonction qui permet à l'objet d'être affiché à l'écran.
- La fonction Hit() : la fonction est appelée lorsque le joueur attaque et que l'objet peut prendre des dégâts. Elle prend en paramètre un rectangle, la zone dans laquelle le joueur a attaqué, et un entier représentant les dommages que l'objet va subir.

2.2 Ajout et suppression des objets

Pour optimiser l'ajout et la suppression des objets, la liste des objets est stockée dans une liste doublement chaînée. L'ajout est géré par la fonction AddObject() qui prend en paramètre un objet et le plan dans lequel il doit être dessiné. Cet index est compris entre 0 et 10. La suppression dans une liste doublement chaînée est peu coûteuse, mais l'ajout n'est pas aussi trivial. En effet, les éléments de cette liste chaînée doivent être triés en fonction de leurs index. Mais si le nombre d'éléments est élevé, le temps de parcours peut être long. Pour pallier à ce problème, les références des premiers nœuds de chaque plan sont mémorisées dans un tableau de 10 nœuds. De même, certains objets peuvent prendre des coups, mais parcourir toute la liste chainée pour appeler la fonction hit des objets qui possèdent cette propriété est également coûteux. Pour optimiser la gestion des dégâts, les objets qui peuvent prendre des coups sont, d'une part stockés dans la liste doublement chaînée pour être mis à jour et dessinés, et, d'autre part dans une autre liste chaînée. Cette liste ne contenant que les objets qui peuvent prendre des dégâts, il est beaucoup plus rapide de la parcourir que la grande liste qui contient tous les objets du jeu.

Le principe de la suppression est donc le suivant : lorsque l'on supprime un objet, on vérifie d'abord s'il peut encaisser des dégâts, si oui, on le supprime de la liste d'objets que l'on peut attaquer. Ensuite, on regarde si le tableau qui mémorise les nœuds contient cet élément, si oui, on remplace la valeur de l'élément du tableau par l'élément suivant dans la liste chaînée, s'il est nul, on le remplace par la valeur précédente, si elle est nulle, on remplace l'élément de ce tableau par la valeur nulle. Le schéma ci-dessous décrit le fonctionnement de la routine AddObject().

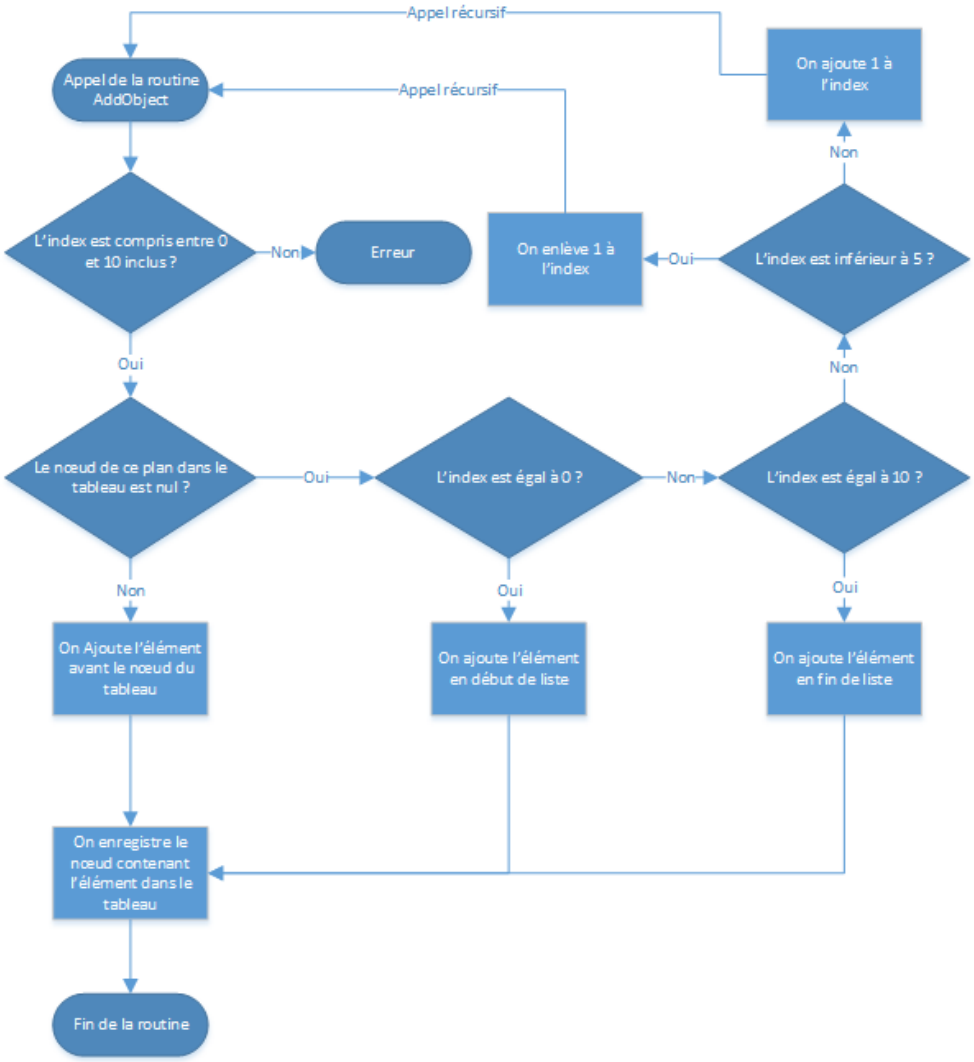

FIGURE 1 – Fonctionnement de la routine AddObject()

3 Les niveaux

Le niveau est composé de blocs. Ce système de blocs permet de charger facilement des niveaux à partir de fichiers. Cependant, les fichiers de niveaux ne contiennent pas que la carte des blocs, ils contiennent aussi les différentes propriétés des événements et scripts inclus dans ceux-ci.

3.1 Gestions des collisions

La gestion des collisions est l'élément le plus important dans un jeu de plateforme. La moindre petite erreur (mur invisible, ou passage à travers un mur) ruine complétement le jeu. La gestion des collisions se traduit par un tableau de booléens à deux dimensions. Pour chaque pixel, il définit si celui-ci est libre ou non (en réalité, ce tableau n'est pas à proprement parler un tableau de booléens, mais un tableau de bits, ce qui permet de diminuer sa taille en mémoire). Ce tableau est utilisé par tous les éléments qui peuvent générer des collisions, à savoir :

— Le joueur.
— Les ennemis.
— Les murs du niveau.
— Les objets scriptés.

Pour tester si un objet de hauteur h et de largeur w peut aller à la position *(x,y)*, il suffit de tester tous les points compris dans le rectangle *(x,y,x+w,y+h)* sont bien libres. La situation se complique lorsque que l'on a besoin du point exact de la collision.

En effet, l'intersection entre deux rectangles peut ne pas existé, soit être un point, soit un segment, soit plusieurs segments, soit un rectangle. Par exemple, si un ennemi lance des shurikens, et qu'il est nécessaire d'afficher des étincelles lorsqu'il rentre en collision avec un mur, il n'est pas aisé d'obtenir le point de d'impact exact. Le shuriken tourne sur lui-même, sa zone de collision peut donc être considérée comme un carré.

Les schémas ci-dessous expliquent les situations possibles :

FIGURE 2 – Ici, la collision est un segment, les étincelles seront affichées au centre du segment.

FIGURE 3 – Dans ce cas, la collision est représentée par deux segments, les étincelles seront donc envoyées à l'intersection des deux segments.

FIGURE 4 – La collision est un point, les étincelles seront affichées à partir de ce point.

Néanmoins, les objets du jeu ne peuvent pas 9 tous être représentés par des rectangles, pour cette raison, un système de tableau de booléens à deux dimensions a été mis au point, représentant le rectangle d'affichage de l'objet. Pour chaque pixel de ce rectangle, le tableau définit si ce pixel peut générer une collision ou non.

3.2 La caméra

La caméra permet de définir ce qui est visible par le joueur. Elle peut soit suivre simplement le joueur au cours de sa progression dans un niveau, soit être fixe, par exemple lors d'un combat avec un ennemi puissant. La caméra de ce projet possède les composantes suivantes :

— Position (le centre de l'écran)
— Zoom
— Rotation

Ces trois composantes peuvent être modifiées dynamiquement par des scripts ou des commandes. Lors du rendu graphique, l'utilisation de la caméra se traduit par une matrice de translation obtenue de la manière suivante :

On crée une matrice de translation à partir de la position de la caméra *(x, y, 0)*. On lui applique une rotation à partir de la valeur Rotation de la caméra (en radians). On multiplie ensuite toutes ces valeurs par Zoom. Et enfin, on lui applique la translation (largeur de l'écran/2, Hauteur de l'écran /2).

3.3 Le scrolling background

ByteCoder est un jeu en deux dimensions, mais il se doit d'être un jeu dynamique. De nombreux effets vont dans cette direction. Nous avons alors pensé à améliorer tous les niveaux en rendant l'arrière-plan dynamique. Il est composé de plusieurs couches qui se déplacent à des vitesses différentes, donnant un effet de profondeur au paysage.

Cet effet est représenté par une classe, qui est instanciée à la création du niveau. Elle va charger des images qui seront des parties de l'arrière-plan. Chacune sera associée à une couche du fond d'écran. Pour chaque monde, un jeu d'images différent sera chargé.

Pour chaque couche, des images sont chargées aléatoirement. Il convient de noter que toutes les images s'enchaînent visuellement : le joueur aura l'impression que l'arrière-plan défile continuellement.

Lorsque la caméra bouge horizontalement, toutes les images de l'arrière-plan se déplacent dans le sens opposé à celui de la caméra, à des vitesses différentes en fonction des couches : la couche la plus au fond ira le moins vite et la plus en avant ira le plus

vite. Cela a pour effet de rappeler le phénomène bien connu de profondeur que l'on peut observer dans la vie de tous les jours. Lorsque l'on conduit près d'une montagne par exemple, on verra les objets proches comme les barrières de sécurité disparaître très rapidement, alors que les montagnes seront presque immobiles, se déplaçant beaucoup plus lentement.

Lorsqu'une image quitte l'écran, elle est effacée, et lorsqu'il en manque une d'un côté de l'écran, une image aléatoire est rajoutée à la liste qui représente les différentes parties d'une couche de l'arrière-plan.

4 Les ennemis

Les ennemis dans ByteCoder sont des objets gérés grâce à une classe, l'EnemyManager. Ils ont un but : tuer le personnage incarné par le joueur. Ils ont différentes caractéristiques définies par un modèle (l'apparence de l'ennemi, sa puissance, ses armes...).

4.1 Le modèle

Chaque ennemi est représenté par un fichier qui sert de modèle. Lorsqu'on créé un modèle dans le jeu (un ninja par exemple) un fichier semblable à celui ci-dessous sera analysé et toutes ces variables seront gardées en mémoire :

```
staticenemy 0
pv 100
distance 1
melee 1
meleedistance 70
meleewait 20
meleedamages 30
canjump 1
deathwaittime 0.0002
attackDistance 350
weaponThrowSpeed 10
weaponThrowInterval 10
weaponspeed 5
weapondamages 30
rotation 30
throwFrequence 80
xspeed 4
yspeedmin 8
yacceleration 0.25
enemyw 74
enemyh 100
weaponw 18
weaponh 18
weaponparticles 1
maximumweaponangle 80
framewait 10
framecount 8
walkspriteend 9
enemySpriteCount 10
```

FIGURE 5 – Extrait d'un fichier de configuration

Toutes ces variables définiront s'il attaque à distance ou de mêlée, sa vitesse, ses points de vie, ou l'emplacement des images à charger. Elles sont dès lors en mémoire et prêtes à être utilisées par l'ennemi.

Les textures sont également découpées pour l'un des effets spéciaux de l'ennemi, qui sera expliqué plus tard.

4.2 La classe de l'ennemi

Lorsqu'un ennemi est instancié, il va charger toutes les variables à l'aide du modèle qui lui est passé en paramètre. Un des avantages de cette technique est d'éviter d'avoir plusieurs copies en mémoire d'une image, une seule servant à tous les ennemis du même modèle.

Une méthode de l'EnemyManager permet de créer un ennemi à partir d'un modèle, son nom que l'on aura donné au moment chargement du modèle, et d'une position en abscisse et ordonné. Cette méthode peut être appelée au chargement du niveau ou alors dans un script : par exemple, l'ennemi serait instancié lorsque le joueur franchit une porte ou arrive à une certaine position dans le niveau.

4.3 Les mécanismes de l'ennemi

Une fois créé, l'ennemi n'aura plus qu'un seul objectif : infliger le plus de dégâts au personnage du joueur. Pour ce faire, il va devoir passer par différents états. Nous allons donc décrire ces différents états qui ont chacun une méthode respective appelée si certaines conditions sont remplies.

Tout d'abord, on peut diviser les ennemis en deux grandes catégories : ceux à distance et les attaquants de mêlée. Il convient également de souligner qu'un ennemi peut avoir les deux caractéristiques.

Voyons donc les états de l'ennemi en fonction de la position du joueur :

— L'ennemi est à une distance inférieure ou égale à la distance d'attaque, à la distance définie dans le fichier modèle, du joueur (seulement si l'ennemi peut attaquer à distance) : à une fréquence définie dans le modèle, il va lancer une arme, ce qui se traduit par l'instanciation de l'objet « Weapon », qui va se gérer de manière autonome. En effet, celle-ci va aller dans la direction du joueur à une certaine vitesse jusqu'à ce qu'elle rencontre le joueur ou autre objet, et si le joueur est touché, elle lui infligera un certain nombre de dégâts tout en étant détruite.

— L'ennemi est à une distance inférieure ou égale, à la distance d'attaque en mêlée définie dans le fichier modèle, du joueur (seulement si l'ennemi peut attaquer de

mêlée) : à une fréquence définie dans le modèle, l'ennemi va attaquer le joueur en lui infligeant un nombre de dégâts, à noter que si cette fonction est appelée, la fonction d'attaque à distance ne le sera pas.
— Si le joueur se trouve à une distance supérieure à la distance d'attaque à distance et/ou celle de mêlée de l'ennemi, et s'il n'y a pas d'obstacle trop grand entre les deux, l'ennemi va avancer vers le joueur tout en évitant les petits obstacles tant qu'il n'est pas à une distance à laquelle il peut attaquer.
— L'ennemi reçoit des dégâts : il est immobilisé pendant une fraction de secondes, il se produit un effet visuel d'interférences magnétiques qui sera expliqué plus tard.

Ce sont ces différents états qui s'enchaînent tant que les points de vie de l'ennemi sont supérieurs à 0, puis l'ennemi disparaît en faisant apparaître un effet visuel de désintégration pendant quelques secondes (expliqué dans le chapitre des effets).

4.4 Le comportement des ennemis en multijoueur

En mode multijoueur, l'ennemi a plusieurs cibles. Ainsi, il doit choisir la plus adaptée. Pour cela, il va devoir comparer la distance entre les joueurs. Mais ce n'est pas tout, il se peut qu'un joueur soit plus proche mais inatteignable : par exemple, un mur peut se trouver entre les deux protagonistes. Ainsi, un calcul de trajectoire entre les différents joueurs présents sur le niveau doit être fait à chaque changement de position.

5 L'événementiel

5.1 L'interpréteur de scripts

Les scripts permettent de dynamiser le jeu, de rendre les niveaux moins répétitifs et de surprendre le joueur. Le but est d'obtenir un langage de script simple, pratique et performant. Un script peut être exécuté à partir d'un fichier ou d'un tableau de chaînes de caractères. Les lignes sont interprétées de la première à la dernière. L'interpréteur de script dispose des fonctionnalités suivantes :

— Créer, modifier et utiliser des variables
— Exécuter des conditions
— Exécuter des boucles
— Arrêter le script en cours avant la dernière ligne
— Charger un niveau

Toutes les variables de scripts sont des entiers flottants. Leur utilisation se fait de la manière suivante :

```
 setlocal ma_variable 0 //On déclare ma_variable et lui affecte la valeur 0
+ ma_variable 4 // On l'incrémente de 4
- ma_variable  1337 // On lui enlève 1337
```

Pour utiliser sa valeur, on utilise l'opérateur « @ ».

```
setlocal ma_jolie_vaiable @ma_variable
// On déclare ma_jolie_variable et on lui affecte la valeur de ma_variable
```

Les conditions fonctionnent avec la syntaxe suivante :

```
if @ma_jolie_variable=42
     //Du code
else
     //Encore du code
end if
```

De la même façon, les boucles s'utilisent comme ceci :

```
while @ma_variable=@ma_jolie_variable
    //Toujours du code
end while
```

Il est possible de stopper un script en utilisant l'instruction « return ».
Pour charger un niveau, la syntaxe est la suivante : load « fichier ».

Si l'interpréteur de script trouve une instruction qui est erronée, ou qui a créé une erreur (Exemple : une instruction load qui échoue car le fichier spécifié n'a pu être ouvert), le script s'arrête immédiatement et un avertissement apparaît sur la console.

Néanmoins, le fait d'interpréter des scripts peut poser des problèmes, en effet, le rendu graphique du projet fonctionne comme une boucle qui met à jour le jeu en permanence. Si un script est très long à s'exécuter, l'image ne sera pas mise à jour et le jeu apparaîtra comme ralenti, voire bloqué. Pour résoudre ce problème, chaque script s'exécute en calcul parallèle (dans un Thread séparé). Il est tout de même possible de forcer l'exécution d'un script sans créer un nouveau thread. (Au cas où par exemple un Script en appelle un autre et doit attendre que celui-ci soit terminé pour continuer). Le nombre de threads en cours d'exécution est plafonné à 10 (ce nombre ne devrait jamais être atteint, mais en cas de problème, il vaut mieux éviter qu'une boucle crée des threads à l'infini).

5.2 Objets scriptés

De plus, l'interpréteur de scripts permet d'animer des objets dynamiques qui se mettent à jour en temps réel en exécutant des commandes. Ces composants sont stockés dans des fichiers « .obj ». Les fichiers contenant les objets scriptés se présentent de la forme suivante :

```
onload // Fonction appelée lors du chargement de l'objet.
    setspritecount 1 // définit le nombre d'images à charger.
    setsprite Data\Sprites\misc\circle.png 0
    loadhitbox  //Charge le rectangle de collision de l'objet
oncreate //Fonction appelée lors de l'instanciation de l'objet.
    this.setglobal speedx 10 //Crée une variable à portée globale
    this.setglobal speedy 8
    this.setframe 0 //définit l'image à dessiner
    this.sethitbox 100 100 //définit la taille de l'objet
onframe  //Fonction appelée à chaque mise à jour de l'objet.
    if @this.touchesplayer=1
        //l'opérateur " @ " indique que ce qui suit est une opération
```

```
        if @this.speedx>0
            hit 1 //la commande "  hit " provoque des dégâts au joueur,
                  //1 si il doit être éjecté à droite, 0 à gauche
        else
            hit 0
        end if
    else
//du code
end if
if (@(this.cangoto (@@this.posx + @this.speedx) (@@this.posy + @this.speedy)))=1
    // cangoto permet de savoir s'il peut se déplacer sans créer de collisions
this.setpos (@@this.posx + @this.speedx) (@@this.posy + @this.speedy)
else
if @(this.cangoto (@@this.posx + @this.speedx) @this.posy )=1
this.setglobal speedy (@@this.speedy * -1)
return //return permet d'arrêter le script
else
this.setglobal speedx (@@this.speedx * -1)
end if
end if
ondestroy //Fonction appelée lors de la destruction de l'objet.

//Du code
```

L'objet ci-dessus est une boule qui rebondit contre les murs à la manière du jeu « pong ». Les commandes commençant par « this. »sont liées à l'objet. Le processus d'exécution d'une commande est le suivant :

- On détermine d'abord si la commande est une condition ou une boucle. Pour ce faire, on regarde si la ligne commence par « if »ou « while »(dans ce cas on enregistre le numéro de la ligne à laquelle revenir) . Si oui on vérifie si la condition est vraie, on exécute les lignes suivantes jusqu'à tomber sur une ligne « end while »(dans le cas d'une boucle) ou « end if »dans le cas d'une condition. Si la condition est fausse, on continue de lire le fichier sans les exécuter jusqu'à arriver sur une ligne « end while »ou « else »puis on exécute jusqu'à la ligne « end if ». Si cette ligne n'est ni une boucle ni une condition, on l'exécute.
- Lorsque l'on exécute une commande, on doit d'abord résoudre les arguments.

Pour ce faire, on sépare d'abord les arguments en fonction des espaces et des parenthèses. On enregistre ces arguments dans un tableau. On parcourt ensuite ce tableau. Pour chaque index, on teste si la chaîne de caractères commence par un « @ ». Si oui, on résout cet argument en exécutant cette chaîne de caractères ôtée de sa première lettre en tant que commande. On résout ainsi récursivement les sous arguments de chaque sous-commande.

— Une fois que tous les arguments d'une commande ont été résolus, on l'exécute. Si celle-ci commence par « this. »c'est une commande d'objet scripté, on passe donc la commande à l'instance d'objet associée. Si celle-ci n'existe pas, alors on déclenche une erreur. Sinon, on passe cette commande à l'interpréteur de commandes.

5.3 Les événements

Les événements permettent de définir lorsqu'une action doit se produire au cours d'un niveau. Les événements sont composés des éléments suivants :

— Quatre entiers qui représentent un rectangle, lorsque le joueur touche ce rectangle, l'événement est déclenché.

— Un entier qui représente le nombre de fois que l'événement peut être déclenché avant de disparaître, 0 pour infini.

— Une chaîne de caractères, c'est la commande à exécuter lorsque l'événement est déclenché, par exemple la commande « exec »qui permet de lancer un script.

Ce système permet de créer des éléments de jeu dynamiques tels que des portes qui s'ouvrent lorsque le joueur s'en approche, des ennemis qui apparaîssent lorsque l'on passe un certain point, créer des effets de caméra, etc. Les événements se créent avec la commande : « event »en utilisant la syntaxe suivante :

Event Min_X Min_Y Max_X Max_Y *nombre_execution*. Avec Min_X, Min_Y, Max_X et Max_Y représentant les quatre points d'un rectangle. La fonction échoue si le rectangle est invalide, ou si des arguments sont manquants.

6 Le joueur

6.1 L'attaque à distance

Jusqu'à présent, le joueur n'était capable de se battre qu'au corps-à-corps. Il était donc fortement désavantagé face aux ennemis qui peuvent lancer des objets. Pour cette raison, il est maintenant capable d'attaquer à distance. Cette attaque à distance se présente sous la forme d'une arme qu'il faut charger pendant quelques secondes avant de pouvoir tirer. Il faut savoir, que pour charger, le joueur doit maintenir la touche de chargement enfoncée. Pendant ce temps de chargement, il ne peut pas se déplacer (il peut néanmoins changer de direction). L'arme tire au relachement de la touche.

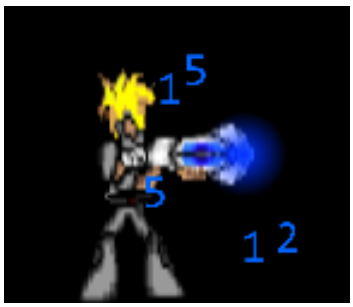

FIGURE 6 – Chargement de l'arme à distance

L'animation du chargement est représentée par trois éléments. Dans un premier temps, il faut définir le centre de l'animation. C'est l'extrémité du canon de l'arme. La première partie de cette animation est composée de chiffres qui convergent vers le centre de l'animation. On définit donc une distance et on calcule les coordonnées du chiffre en générant un angle aléatoire en utilisant les équations suivantes :

— $X = X_{centre} + \cos angle$
— $Y = Y_{centre} + \sin angle$

Une fois que les coordonnées du chiffre sont définies, il faut le faire converger vers le centre de l'animation. Pour ce faire, on lui applique deux vitesses, une vitesse en abscisse et une en ordonnées. On définit la vitesse de rapprochement des chiffres par une constante (ici $Vitesse_R$). Les vitesses sont définies par les équations suivantes :

— $Vitesse_X = -Vitesse_R * (x - X_{centre})$
— $Vitesse_Y = -Vitesse_R * (y - Y_{centre})$

Il suffit ensuite d'augmenter la valeur de l'angle pour faire tourner le chiffre.

Le cercle doit représenter une lumière bleue, pour cette raison, il doit être d'une transparence nulle au centre et la transparence de chaque point de sa texture doit être fonction de la distance de ce point par rapport au centre de ce cercle. La transparence d'un point de coordonnées (X, Y) est définie par l'équation :
Transparence (comprise entre 0 et 255) $= 255 - Min(255, (\sqrt{(x - rayon)^2 + (y - rayon)^2) * 255})$.

Le tir est lui représenté par une suite de chiffres se déplaçant à une vitesse donnée. Si cette suite de chiffres rencontre un objet, celui-ci va encaisser des dégâts. Si ce groupe de chiffre ne rencontre pas d'objet ou de mur au-delà d'une distance donnée, il explose en projetant des lettres autour de lui. Le schéma ci-dessous illustre cette trainée de chiffres.

FIGURE 7 – Trainée de chiffres

6.2 L'attaque en saut

Après avoir nous-mêmes testés notre jeu, nous nous sommes rendu compte que le joueur se fait souvent attaquer pendant qu'il saute sans pouvoir se défendre. Nous avons donc décidé de permettre au joueur d'attaquer en saut. Si le joueur est en l'air et qu'il appuie sur la touche saut, il va attaquer, ce qui va lui redonner de l'élan dans son saut.

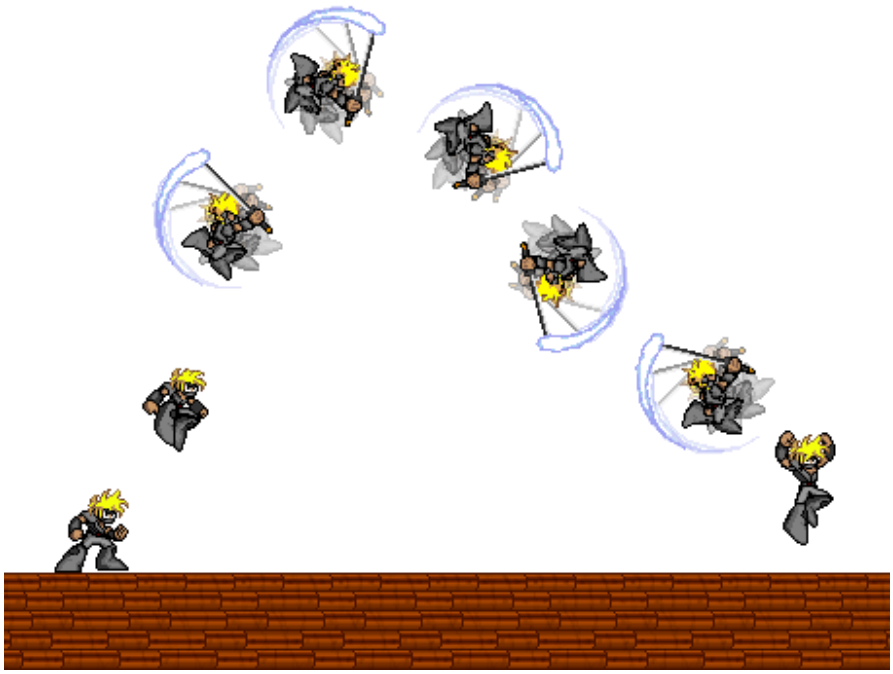

FIGURE 8 – Aperçu de l'attaque en saut.

6.3 Le « dash »ou la glissade

Au début du jeu, le joueur ne pouvait attaquer que s'il était immobile, ce qui rendait l'attaque difficile. Pour pallier ce problème, nous avons pris la décision de rajouter une attaque durant laquelle le personnage avance sur quelques mètres. Cette attaque, nommée « dash »ou glissade en français, permet au joueur d'infliger des dommages au premier ennemi qu'il rencontre durant sa course. Pour ce faire, il suffit de maintenir la touche de course et de presser la touche d'attaque au corps-à-corps.

FIGURE 9 – Exemple d'utilisation de la glissade

6.4 La lance

La lance est une arme qui, lorsqu'elle rencontre un mur peut servir de plateforme au joueur.

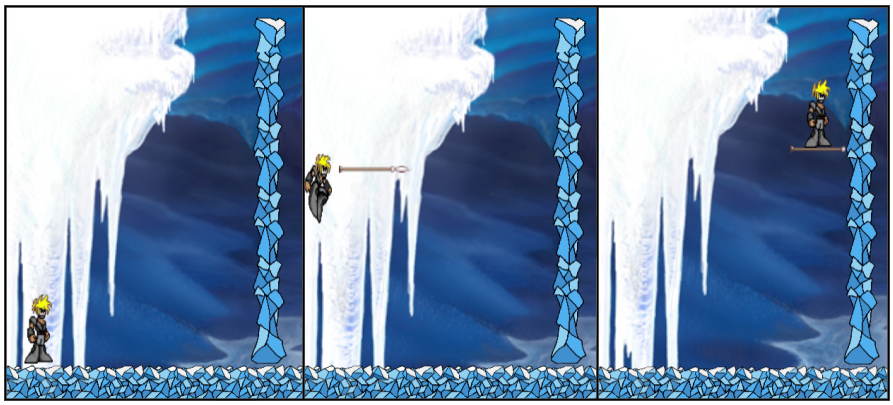

FIGURE 10 – Utilisation de la lance

6.5 La gestion des contrôles

La gestion des contrôles a pour but d'associer une touche à une commande. Ces associations sont enregistrées dans une liste chaînée d'enregistrements du type suivant.

```
Enregistrement AssocTouche
    Touche touche
    Booleen relachement
    /* Définit si la touche doit être relachée
       pour être de nouveau interprétée */
    Booleen relachee  /* Définit si la touche a été relachée */
    Chaine cmd /* La commande associée a la touche */
Fin Enregistrement AssocTouche
```

Lors de la mise à jour, on parcourt la liste, et on teste pour chaque enregistrement si AssocTouche.touche a été pressée. Si oui, on exécute la commande si :

— Soit AssocTouche.relachement est faux.

— Soit AssocTouche.relachement est vrai et AssocTouche.relachee est vrai.

Puis on met AssocTouche.relachee à faux. L'utilisation d'une liste chaînée permet l'ajout et la suppression d'associations en jeu.

7 Le multijoueur

7.1 Le multijoueur local

La possibilité de jouer à plusieurs ajoute une nouvelle dimension aux mécaniques de jeu de ce projet. Mais « multijoueur »peut avoir plusieurs sens. En effet, ce mot peut signifier soit la coopération, soit le combat. Dans le cadre de ce projet, la coopération a été privilégiée, car elle convient mieux à l'univers du jeu. En effet, il est plus logique de voir deux joueurs combattre côte à côte pour maintenir la stabilité d'un système d'exploitation plutôt que de les voir s'affronter sans aucune raison logique. De plus, l'esprit d'équipe est un meilleur moteur d'amusement que l'adversité.

Néanmoins, avoir des idées est une chose, les réaliser en est une autre. Le code de ce projet est écrit en respectant quatre règles, la sécurité, la flexibilité, les performances et la clarté. Le fait d'avoir un code souple a facilité l'intégration du multijoueur. En effet, la structure décrite plus haut implique que chaque objet vu à l'écran hérite de GameObject. L'objet représentant le joueur est donc lui aussi, un GameObject. Ce qui signifie qu'une fois qu'il est ajouté à la liste d'objets à afficher, sa mise à jour et son affichage sont gérés automatiquement jusqu'à sa destruction. Il est donc possible d'ajouter autant de joueurs que l'on souhaite à la liste de rendu. Le nombre maximum de joueurs a été fixé à 4, car au-delà de ce nombre, le jeu devient trop anarchique.

7.1.1 Les entrées utilisateur

Il ne suffit pas de dessiner les images des joueurs pour faire fonctionner le multijoueur. En effet, il faut transmettre les entrées utilisateurs aux joueurs pour qu'ils puissent se déplacer, attaquer, etc. Il est donc nécessaire de garder un tableau à quatre dimensions contenant les références des joueurs pour pouvoir leur transmettre les commandes de l'utilisateur, si le joueur n'est pas défini, la valeur de son index dans le tableau est nulle.

Les entrées utilisateurs sont gérées par des commandes. Comme expliquées plus haut, chaque touche (du clavier ou de la manette) se voit attribuer une commande. Lorsque la touche est interprétée par le jeu, la commande associée est exécutée. Les commandes qui ont pour but d'interagir avec le personnage commencent par un « + »et se terminent par le numéro du joueur auquel elle est liée. Par exemple, pour faire sauter le joueur un, la commande est : « +jump 0 »(les index commencent à 0). Pour faire

attaquer le joueur 4, la commande est donc : « +attack 3 ». La commande déclenche bien sûr une erreur si l'index n'est pas compris entre 0 et 3 ou si le joueur auquel elle fait référence n'a pas été créé.

7.1.2 Le partage de l'écran

Le multijoueur local implique le partage de l'écran. En effet, plusieurs personnages vont être affichés à l'écran. Il existe deux moyens pour que deux personnes jouent sur le même écran :

- On peut scinder l'écran en plusieurs parties et attribuer chaque partie à un joueur unique, ce qui implique que l'écran de chaque joueur sera plus petit que l'écran de base.
- On peut choisir d'afficher tous les joueurs sur un seul et même écran et diminuer le zoom lorsque les personnages s'éloignent pour que tous les personnages restent visibles. Mais lorsque les joueurs s'éloignent beaucoup, le zoom est tellement faible que l'on ne voit plus rien.
- On peut reprendre la méthode décrite ci-dessus mais cette fois-ci empêcher les personnages de s'éloigner au-delà d'une distance prédéfinie. Mais cela peut gêner le joueur dans certains niveaux.

Ces trois procédés présentent tous des avantages et des inconvénients, pour donner au joueur la meilleure expérience de jeu possible, seuls les avantages ont été conservés pour aboutir à la méthode suivante : lorsque les personnages sont proches, ils sont tous affichés sur un seul écran, au fur et à mesure qu'ils s'éloignent, le zoom diminue et dès que la distance entre les joueurs est au-delà d'une limite prédéfinie, l'affichage passe en mode écran scindé. Le joueur peut néanmoins choisir de forcer un des trois types d'affichage s'il préfère jouer ainsi.

Pour aboutir à un tel résultat, il est nécessaire de connaître la plus grande distance entre les personnages. Pour ce faire, il faut comparer les distances de chaque joueur avec tous les autres et conserver la valeur la plus élevée. Pour calculer les distances, on utilise donc la formule : $\sqrt{(x_1 + x_2)^2 + (y_1 + y_2)^2}$.

Ensuite, il faut calculer le zoom approprié. La valeur du zoom est calculée de la manière suivante : Soit une distance x (en pixel), la valeur de cette distance sera, après application du *zoom* : $\frac{x}{zoom}$. Le zoom par défaut est donc de 1. Soit *d la distance la plus élevée entre les personnages du jeu. Cette distance peut représenter deux situations :*

— *Un joueur est en bas à droite et l'autre en haut à gauche, la distance est donc maximale.*
— *Les deux sont à la même altitude avec l'un à gauche et l'autre à droite.*

Cela implique qu'à distance égale, le zoom peut être différent. Le zoom est minimal dans le premier cas. Pour cette raison, les distances entre les joueurs sont toujours interprétées comme dans le premier cas, ce qui permet d'éviter qu'un personnage soit en dehors de l'écran. Ce qui signifie que la diagonale de l'écran doit être de $\sqrt{distance^2 + (longueurecran)^2}$, d'après le théorème de Pythagore. Soit X la valeur de la diagonale de l'écran lorsque le zoom est à 1. On obtient donc : $zoom = \sqrt{distance^2 + (longueurecran)^2)}/X$. (Pour optimiser les performances, la longueur de l'écran est mise au carrée, ce qui permet d'éviter le calcul d'une racine carrée). Enfin on applique : $zoom = zoom * 0.9$ pour que les joueurs aient un peu de marge et puissent voir devant eux.

7.2 Le réseau

Pour implémenter le multijoueur en réseau, plusieurs décisions ont dues être prises. La première était le choix du protocole. Il existe deux possibilités :

— L'UDP(*User Datagram Protcol*). Ce protocole est le plus rapide, mais la perte de paquets n'est pas gérée et il n'est pas garanti qu'ils arrivent dans le même ordre.
— Le TCP (*Transmission Control Protocol*). Ce protocole est moins rapide, mais il garantit que tous les paquets arriveront et qu'ils seront dans le bon ordre.

Etant donné que l'on ne peut pas se permettre de perdre des paquets (et encore moins de les recevoir dans le désordre) le TCP a été choisi.

Le jeu en réseau fonctionne avec une architecture client-serveur. Le serveur est ici séparé du jeu. Il faut donc que le serveur soit lancé puis que tous les joueurs s'y connectent. Pour la connexion, un broadcast UDP(détaillé plus bas) permet de lister tous les serveurs présents sur le réseau local. Il est néanmoins possible de rentrer une adresse IP(*Internet Protocol*) et un port manuellement, par exemple si deux joueurs veulent jouer derrière un Nat.

Le serveur peut accueillir jusqu'à quatre joueurs sur 4 machines différentes. (Il est possible de jouer à deux sur la même machine, on peut par exemple jouer à 4 avec deux

machines). Une fois que tous les joueurs sont connectés et que le niveau a été choisi, la partie commence. Pour le déplacement et l'animation des joueurs, le serveur transmet les déplacements de chacun aux autres machines. Cela signifie que si par exemple le joueur 1 appuie sur la touche de saut la commande suivante va être exécutée : « +jump 0 ». L'interpréteur de commande va analyser cette commande. Elle commence par un « + » : c'est donc une commande d'entrée utilisateur. Elle se termine par 0, elle s'applique donc au joueur d'indice 0 dans le tableau. On regarde ensuite si ce joueur est un joueur local ou distant. Si c'est un joueur distant, la commande est exécutée et cela s'arrête là. Si c'est un joueur local, la commande est transmise au serveur qui va la faire suivre aux autres clients.

Cette méthode présente néanmoins un problème car la mise à jour du jeu se fait dans un thread tandis qu'un autre thread enfile les commandes en attente dans une file. Pendant la mise à jour du jeu, on va exécuter toutes les commandes en attente (on ne peut pas exécuter de commandes de manière asynchrone).

A chaque mise à jour du jeu, la fonction Next est appelée pour chaque joueur. Cette fonction sert par exemple à augmenter la vitesse verticale d'un joueur si celui-ci est en train de tomber. Imaginons qu'un joueur saute, puis se déplace à droite. Il va donc monter en se déplaçant vers la droite. Les commandes suivantes vont donc être transmises au serveur : « +jump 0 », « +right 0 ». Imaginons à présent que par malchance un client reçoive la commande « +jump 0 »juste avant de faire la mise à jour. Il exécute donc cette commande puis appelle Next sur le joueur et il ne recevra la deuxième commande qu'à la mise à jour suivante. Ce qui va créer un décalage entre les deux machines.

Pour résoudre ce problème, il faut que chaque client n'appelle les fonctions Next que sur les joueurs locaux. Lorsque qu'une fonction Next est exécutée, la commande « next (numero_joueur) »est transmise au serveur, ce qui permet aux autres clients d'appeler la fonction Next sur les joueurs distants. Cette méthode assure que les déplacements seront bien identiques sur toutes les machines. Le défaut de cette solution est l'effet de tremblement. En effet, à cause du temps que met une commande à arriver, il est possible qu'à certaines mises à jour, aucun appel de Next ne soit exécuté pour qu'à la mise à jour suivante, deux appels soit exécutées. Il arrive donc qu'un joueur soit immobile pendant une mise à jour et qu'il bouge deux fois plus vite qu'à la normale à la mise à jour suivante.

On pourrait croire que cette méthode est parfaite et qu'aucun décalage n'est possible. Malheureusement, ce n'est pas le cas. Imaginons que deux joueurs soient face à un ennemi et que les deux l'attaquent exactement au même moment. Les deux joueurs se verront attribuer la mort de cet ennemi. Ce qui crée un décalage. Pour pallier à ce problème, on doit forcer la synchronisation de tous les clients. Lorsque la partie démarre, le serveur teste les temps de réponse de chaque client. Il définit ensuite le client avec le temps de réponse le plus faible comme « maître de la partie ». Ce client va donc envoyer régulièrement la totalité de l'environnement au serveur (les positions de chaque ennemi, l'état de chaque objet, etc.), ce qui empêche définitivement un décalage progressif de la partie. Le protocole fonctionne donc avec deux ports. L'un est utilisé pour envoyer les commandes, et l'autre pour les recevoir.

7.3 Le broadcast UDP

Lors d'une partie en LAN (réseau local), les différents joueurs doivent se connecter entre eux. Une solution consisterait à taper au clavier l'adresse IP du serveur que l'on veut rejoindre. Cette option est d'ailleurs disponible dans le menu réseau, mais il est rarement agréable ou même évident, pour l'utilisateur, de devoir aller demander l'adresse IP.

Nous avons donc décidé de nous servir du protocole UDP afin de permettre à l'utilisateur de trouver les parties disponibles sur son réseau sans n'avoir rien à faire, si ce n'est, cliquer afin de rejoindre la partie.

7.3.1 L'implémentation partie serveur

Le broadcast UDP permet d'envoyer des paquets à une adresse IP dite "de broadcast" (le plus souvent c'est 255.255.255.255). Ces paquets sont reçus par tous les ordinateurs connectés au réseau.

Le broadcast est géré par une fonction appelée dans un thread séparé par le serveur. Celle-ci va créer une chaîne de caractères contenant différente contenant les informations suivantes :

- L'adresse IP du serveur
- Le port du serveur
- Le nom du niveau en court
- Le nombre de joueurs

A l'aide d'une boucle et de pause du thread, un paquet contenant cette chaine de caractères sera envoyé chaque seconde tant que la partie n'a pas commencé.

7.3.2 L'implémentation partie client

La réception des paquets broadcastés se fait grâce à une méthode du menu de connexion. Cette dernière va détecter tous les paquets broadcastés dans le réseau local pendant un certain temps (8 secondes). Dès qu'un paquet est reçu, il va être analysé afin de déterminer si son contenu correspond au modèle attendu. Si oui et que l'on en a pas déjà reçu un, la chaîne de caractères reçu va être divisée (grâce aux tirets qui séparent chaque information) et ces éléments vont être stockés dans une structure qui sera elle-même placée dans une liste chainée, représentant tous les serveurs sur le réseau. Le joueur n'aura plus qu'à choisir le serveur sur lequel il veut se connecter à l'aide du menu.

8 Les effets

8.1 Les particules

Toujours dans l'optique de rendre le jeu le plus dynamique possible, nous avons décidé d'implémenter un moteur à particules. Celui-ci sert pour la majorité des effets simples du jeu. Ce moteur à particules se résume en trois classes. Voici une explication détaillée de leur rôle et de leur fonctionnement.

Dans notre jeu, les effets, comme les étincelles, sont un ensemble de particules réagissant à certaines règles, qui, unies, vont créer quelque chose de reconnaissable par le joueur. Cet objet, la particule, prend plusieurs arguments dans son constructeur : sa position, sa couleur, sa vitesse, une variable représentant la gravité, ainsi que sa durée de vie.

Le modèle de l'effet est décrit dans un fichier. Les particules composant l'effet doivent avoir des règles bien définis. C'est le rôle de cette classe qui va gérer les différentes variables de chaque effet (le sens des particules, leur durée de vie, le nombre de particules à créer avant la fin de l'effet, etc.).

Le moteur à particules, appelé ParticleManager, va charger ce modèle, puis créer des particules à la fréquence donnée par le modèle, et supprimer chaque particule lorsque sa transparence (comprise entre 1 et 0) atteint 0. En effet chaque particule diminue sa transparence à chaque image.

8.2 L'effet d'apparition

Cet effet est utilisé lors de la création des joueurs. Il repose sur deux classes : *SpawnEffectManager* et *SpawnParticle*. La classe *SpawnParticle* définit les attributs pour une particule alors que la classe *SpawnEffectManager* va se charger de déplacer les particules selon l'axe des x et l'axe des y en fonction de leurs vitesses. L'objectif de cet effet est de rassembler toutes les particules en un point. Cela donne donc l'impression que le personnage est créé de toute pièce, d'autant plus que la texture des particules provient de celle du personnage.

La classe *SpawnEffectManager* va donc instancier des *SpawnParticles* lors de sa création. Elle dispose d'une méthode *Next* permettant de mettre à jour les coordonnées

des particules et d'en créer des nouvelles si nécessaire. L'arrêt de cet effet est dû au fait qu'aucune particule ne peut être mise à jour en retournant un booléen *faux* lors de l'appel à sa méthode *Next*. On pourra alors libérer la mémoire occupée par ces particules et ce *SpawnEffectManager* en appelant la méthode *Dispose* de ce dernier

8.3 L'effet de mort

Lorsqu'un ennemi meurt, ou que le personnage principal meurt, il s'agit de lui donner une animation digne de ce nom. Ainsi, lorsque quelqu'un meurt dans le jeu, il ne fait pas que disparaître d'une image à l'autre, mais se désintègre lentement jusqu'à redevenir poussière (à l'exception des boss qui se doivent d'être plus spectaculaires).

Les particules composant cet effet sont des petites parties de la texture de l'objet en train de mourir, découpée grâce au TextureBuilder que nous avons créé. Les particules sont représentées par une liste dynamique, créée dans le constructeur. L'entier symbolisant le temps d'attente avant l'animation de la particule, qui est passé en paramètre, diffère en fonction de la position de la particule. Ainsi, les particules ayant le temps d'attente le plus court, seront celles qui sont les plus proches du point d'impact.

Enfin, les deux méthodes habituelles sont présentes dans la définition de la classe. La méthode de mise à jour appelle la méthode éponyme de chaque particule, en vérifiant ce qu'elle retourne. Si elle retourne l'octet signifiant la fin de la durée de vie, elle supprime la particule de la liste. La seconde dessine chaque particule de la liste.

8.4 Le feu

Nous avons décidé d'implémenter le phénomène naturel, et le plus souvent spectaculaire, qu'est le feu. Etant donné la complexité de celui-ci, ainsi que les différentes formes qu'il peut prendre en fonction du vent et de l'intensité, nous avons convenu de dédier trois classes à celui-ci.

L'effet reprend le principe du ParticleManager, tout en l'adaptant aux besoins techniques qu'il exige.

8.4.1 FireParticle

Cette classe représente la plus petite unité de feu. Celui-ci est composé de plusieurs instances de cet objet. Son constructeur prend en paramètres :

— la texture
— sa position de départ
— sa taille
— une variable représentant la diminution de la transparence de la particule
— sa couleur
— son angle

Elle possède deux méthodes. La première gère les mises à jour des positions et de la transparence, qui diminue. Elle renvoie un booléen indiquant si sa transparence est nulle ou non. La deuxième dessine la particule.

8.4.2 LittleFire

Cette classe représente les flammèches qui composeront l'effet de feu. Le constructeur prend en paramètre :
— la plage de l'abscisse dans laquelle les particules peuvent apparaître
— l'angle minimal et maximal qui influencera la vitesse en x
— la vitesse en y des particules
— un générateur de variables aléatoires

La liste dynamique de FireParticles est le composant principal de la classe. Un masque, représenté par un tableau de booléens permet de savoir où peuvent apparaître les particules à chaque mise à jour. Ainsi, à chaque rafraîchissement, le masque est changé à l'aide du générateur de variables aléatoires passé en paramètre, ce qui permet d'avoir une distribution non uniforme, et d'autant plus naturelle, des particules.

Une méthode permet l'affichage des particules à chaque mise à jour en parcourant la liste de particules et appelant leurs méthodes de dessin.

8.4.3 FireManager

Cette classe est le pilier central de l'effet de feu. Elle gère toutes les flammèches qui le composent. Le constructeur prend le gestionnaire de variables central en paramètre, ainsi que l'intervalle en abscisse dans lequel le feu peut apparaître. Plusieurs flammèches sont ensuite instanciées en prenant en paramètre des variables du gestionnaire de variables (vitesse, angle, etc.). On retrouve une méthode de mise à jour et une de dessin, qui appellent celles des flammèches.

8.4.4 L'effet d'hyperespace

Le menu est souvent la première impression que l'on se fait du jeu, étant donné que c'est la première chose que l'on voit quand on l'ouvre. Ce n'est pas parce que ce n'est pas proprement dit un élément du jeu qu'il doit être délaissé. Nous avons alors choisi de reprendre un concept connu des fans de science-fiction, qui est l'hyperespace.

L'hyperespace désigne un mode de transport dans la science-fiction : le voyage à la vitesse de la lumière. Il est utilisé dans beaucoup d'œuvres cinématographiques se déroulant en partie dans l'espace, de *Doctor Who* à *Star Wars* en passant par H_2G_2.

Cet effet spécial représente en fait la distorsion apparente du paysage (en l'occurrence composé d'étoiles) due à la vitesse supraluminique du vaisseau se déplaçant. Etant donné que notre jeu se déroule dans un univers numérique, nous avons pensé que cet effet d'hyperespace pouvait s'appliquer aux électrons voyageant dans les circuits de l'ordinateur.

FIGURE 11 – L'effet d'hyperespace dans le menu

Pour faire cet effet, nous avons créé une classe chargée au début du jeu. Celle-ci va créer des particules. Chaque particule est en fait une file d'une autre sous-classe, composée d'un Rectangle et d'un flottant compris entre 0 et 1, représentant la transparence

de la texture à dessiner. La particule prend en paramètre un angle, à partir duquel elle va calculer la vitesse, puis l'accélération de la particule, grâce à la taille de l'écran.

Lors de son instanciation, la transparence alpha est à 0 : la sous-particule est invisible. A chaque rafraichissement du jeu, on ajoute une sous-particule en tête de file et on défile la position la plus ancienne. Tant que la sous-particule défilée est comprise dans l'écran d'affichage, la méthode Update() renvoie un booléen valant *vrai*, sinon on renvoie un booléen valant *faux*.

Cette classe prend en paramètre la taille maximale qu'une file de sous-particules peut avoir. Sa fonction de mise à jour va ajouter une particule avec un angle aléatoire compris entre 0 et 360 degré à la liste doublement chaînée de HyperspaceParticle. Puis elle va vérifier le booléen renvoyé par chaque particule en appelant leur méthode Update(). Si cette dernière renvoie la valeur *faux*, la particule est retirée de la liste et détruite.

8.5 Effet du menu de mort

Lors de la mort d'un joueur, nous avons choisi, pour le menu qui apparaît, de mettre un effet qui correspond à l'univers du jeu : rien de mieux qu'un Kernel Panic. Pour cela, le jeu va lire un fichier dans lequel sont écrites les instructions qui pourraient apparaître lors d'un Kernel Panic ordinaire. On peut donc y voir la pile des appels, le problème qui a causé le Kernel Panic, celui-ci étant évidemment dû à un problème de RAM (*Random Access Momory*) car la vie du personnage est représentée par une barre de RAM.

8.6 Effet du menu principal

Le fond dynamique est l'effet utilisé dans l'écran principal. Il permet le défilement vertical de séries de caractères. Ce défilement permet de faire apparaître le nom du projet : The ByteCoder. Pour faire cet effet, nous avons utilisé un système de "Font Manager" qui permet de créer des caractères à partir d'une texture. Ensuite, les caractères sont disposés en colonnes. Ces colonnes défilent soit de haut en bas, soit de bas en haut, et chacune des colonnes a une transparence et une vitesse de défilement générées aléatoirement, ces propriétés sont propres à chacune des colonnes.

8.7 L'effet d'interférence magnétique

Lorsqu'un ennemi prend des dégâts, il est paralysé pendant une fraction de secondes et il se voit remplacé par un effet d'interférence. Ce dernier est représenté par une classe qui prend en paramètre la position, la taille de l'ennemi ainsi qu'un tableau de sous-textures représentant une texture de l'ennemi découpé horizontalement.

L'effet consiste à faire légèrement translater horizontalement alternativement vers la droite et vers la gauche afin de créer cet effet d'image brouillée. Cela donner l'effet suivant :

FIGURE 12 – L'effet d'interférence sur un Dalek

9 L'univers

9.1 Le scénario

L'histoire du ByteCoder se déroule dans un monde purement informatique : Memoria. Dans ce monde, un programme, que l'on pourrait associer à un virus, corrompt le système mis en place. Il arrive ainsi à détourner les différents programmes en cours d'exécution et à les retourner contre la machine. Son but est de récupérer les ressources de la machine pour devenir plus puissant que jamais. Cependant, le ByteCoder réussit à échapper à cette influence, sauvé par l'utilisateur. Il se retrouve donc dans un monde où tout ou presque essaie de le tuer, hormis quelques autres programmes qui ont réussi à se cacher, mais qui sont trop faibles pour se battre.

Le ByteCoder a donc pour mission de rétablir la stabilité de ce système. Pour cela il va s'aventurer dans les différentes couches de l'informatique, à commencer par le Web. Il devra ensuite se confronter aux dangers du .net. Après, viendra le monde plutôt rude, le cœur de la machine : le noyau. Cependant un dernier monde se cache derrière tout ça : l'algorithmique formel, le plus redoutable de tous les mondes, où tout peut arriver.

Dans chacun des mondes, le ByteCoder affrontera des personnages emblématiques comme FlashPlayer dans le monde du Web, ennemi qui n'abandonne jamais sa proie tant qu'elle est vivante. Dans un monde de glace, il se retrouvera face à des "blobs" qui envahissent le système et qui se reproduisent à foison. Mais le ByteCoder devra surtout affronter des boss pour chaque fin de niveau. Chacun des univers est basé sur un boss unique. Il est donc possible de retrouver Apache, emblème du monde du Web, encore une fois. Mais certaines rumeurs disent que Tux se trouve peut-être dans l'un de ces mondes...

9.2 Les personnages non joueurs, non hostiles

9.2.1 Les PNJ dans l'aventure

Au cours de son aventure, le joueur rencontrera plusieurs PNJ (Personnage Non Joueur) ces derniers lui donneront des indications sur le déroulement de l'intrigue ou plus directement sur le jeu. On peut facilement imaginer un niveau d'initiation ou un PNJ nous parle pour nous expliquer les différents mouvements et actions que le joueur peut faire.

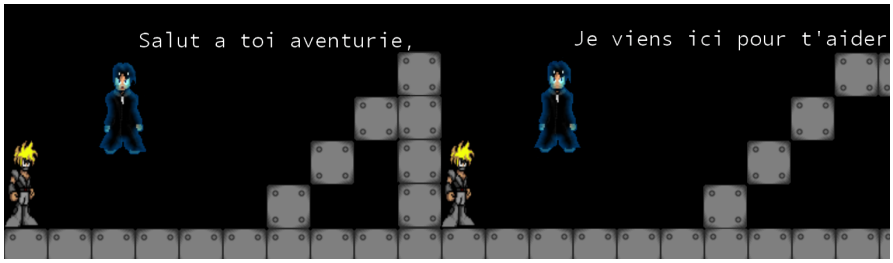

FIGURE 13 – Voici un personnage non joueur qui propose son aide au joueur.

9.2.2 Le fonctionnement des PNJ

Pour implémenter un PNJ dans un niveau, il faut tout d'abord charger son modèle. Ce modèle contient toutes les informations nécessaires à son affichage ainsi qu'à son animation.

Pour permettre à un PNJ de converser avec le joueur, ce dernier possède des lignes de dialogues préenregistrés. Ce qui permet à tout moment de faire apparaître un PNJ pour donner des informations au héros. Son comportement étant totalement scripté il est possible de le déplacer ou de changer son texte dynamiquement.

9.3 L'univers Graphique

Les graphismes dans un jeu vidéo sont sûrement un des aspects les plus importants. En effet, le scénario ainsi que les mécaniques de jeu en sont dépendants. Durant l'élaboration du jeu nous avons trouvé plus intéressant de proposer plusieurs univers aux joueurs. Ce qui offre au jeu une plus grande profondeur.

9.3.1 Le monde « Western »

FIGURE 14 – Une partie des éléments du monde « Western ».

Pour la réalisation du premier monde nous nous sommes inspirés de l'univers du « Far West », qui se marie aisément avec la science-fiction. Contrairement aux autres mondes proposés il est le seul doté de couleurs chaudes.

9.3.2 Le monde de glace

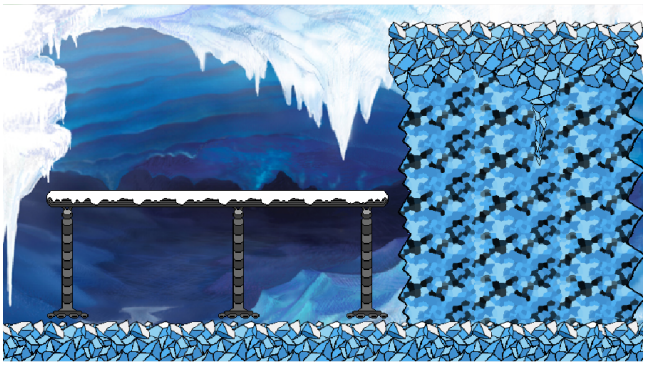

FIGURE 15 – Quelques éléments du monde de glace.

L'inspiration de ce monde provient de jeux tels que Megaman, en effet dans ce dernier le monde de glace est un élément récurrent. Cet univers étant moins chaleureux, il prépare le joueur à faire face à une plus grande difficulté et marque ainsi l'avancée du joueur dans son périple.

9.3.3 L'usine

L'usine étant le premier monde que nous avons imaginé, il est quelque peu plus étoffé. Étant aussi récurrent à plusieurs licences de jeu et s'intégrant parfaitement à

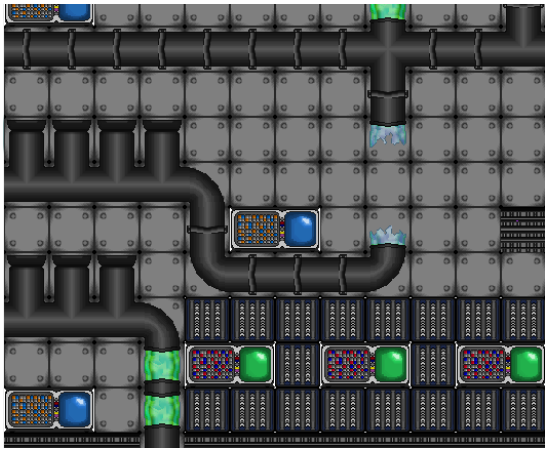

FIGURE 16 – Une zone caractéristique de l'usine.

notre scénario, cet univers était un candidat de choix. Ce dernier permet d'élaborer des niveaux longs et labyrinthiques qui mettront à mal la persévérance du joueur.

9.4 Bug Brother

Bug Brother est un ennemi coriace au déplacement lent. Il est armé d'une puissante mitrailleuse. Il est plutôt difficile de l'approcher ce qui rend cet ennemi extrêmement difficile à tuer. Lorsqu'il apparaît, une musique est jouée pour avertir le joueur.

FIGURE 17 –
Bug Brother attaquant le joueur.

9.5 Le Blob

Contrairement à Bug Brother, le blob est un ennemi facile à tuer lorsqu'il est seul. C'est une entité qui a le pouvoir de sauter de temps en temps dans une direction aléatoire. Ce qui le rend réellement dangereux, c'est sa capacité à pouvoir se dédoubler. Parfois lorsqu'il saute, un autre blob apparaît. Ce qui signifie que plus il y en a, plus ils se multiplient. Il suffit de quelques minutes aux blobs pour submerger tout un niveau. Par sécurité, les blobs ne peuvent plus se multiplier lorsqu'ils sont plus de 1000. Cela permet d'éviter au jeu de demander trop de ressources.

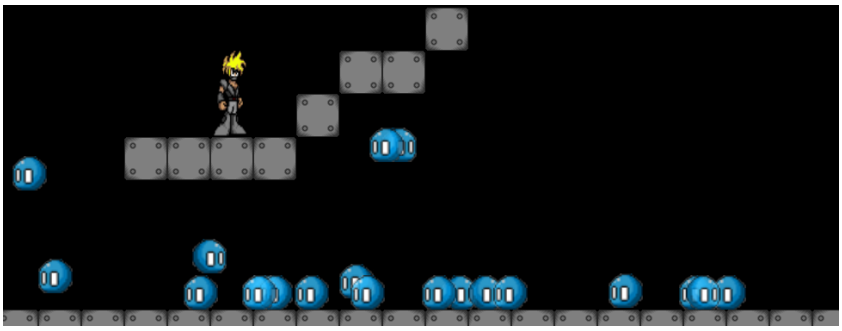

FIGURE 18 –
Une invasion de Blobs.

9.6 La mine de proximité

Comme son nom l'indique, la mine de proximité est un objet qui explose lorsque le joueur s'en approche d'un peu trop prés. L'explosion peut également infliger des dégâts aux ennemis. La lumière clignote juste avant l'explosion.

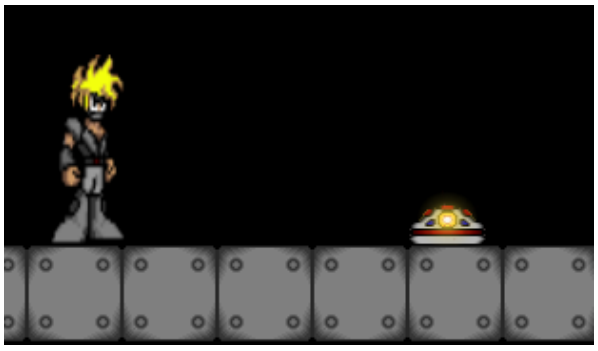

FIGURE 19 –
Une mine de proximité.

9.7 La grenade

Les grenades peuvent être lancées par les joueurs ou par certains ennemis. Le joueur droit rester prudent lorsqu'il s'en sert car celle-ci peut rebondir contre un mur et se retourner contre lui.

9.8 Apache

Apache est le premier boss du jeu. C'est lui le maître du monde du web. Il est représenté par un indien. Il possède plusieurs types d'attaque.

FIGURE 20 –
Les trois attaques d'Apache.

9.9 Le lance-missiles

Le lance-missiles est un objet statique. Il tire des missiles à intervalles régulier. Une fois tiré, le missile va chercher le joueur le plus proche. Une fois qu'il a trouvé sa cible, il va se diriger vers joueur jusqu'à ce qu'il rencontre un obstacle. S'il rencontre un obstacle ou un joueur, il explose, infligeant des dégâts à tout ce qui se trouve dans la zone d'explosion, y compris les ennemis.

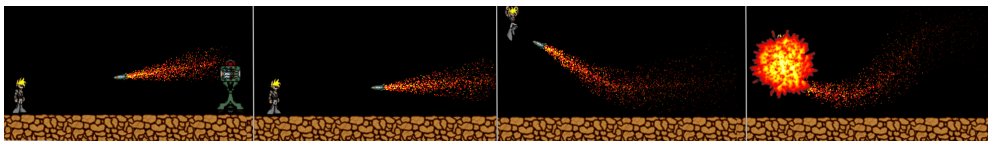

FIGURE 21 –
Un lance-missiles en plein action.

10 Le son

Un jeu vidéo est une expérience audio-visuelle. Il ne faut donc pas négliger l'importance du son. Dans ce projet, deux types de son ont été dégagés :

— Les effets, tels que les explosions, les bruits de tirs, etc. Ces effets sonores sont d'une durée relativement courte.

— Les sons d'ambiance tels que les musiques. Ces sons ont une durée plus longue et peuvent durer pendant plusieurs niveaux.

Les sons d'effets sont donc lus jusqu'à la fin. Les sons d'ambiances fonctionnent d'une manière différente. Lorsque l'on veut changer le son d'ambiance, on appelle la fonction « SetAmbiance » qui prend en paramètres le nom de la musique. Cette fonction compare le nom de l'ambiance avec celui de l'ambiance actuellement mise en place. Si ces deux noms sont égaux, la fonction ne fait rien, sinon, elle fait lentement descendre le volume du son d'ambiance jusqu'à 0 puis démarre la nouvelle musique d'ambiance en augmentant lentement le volume. Cette méthode permet d'adoucir les transitions entre deux ambiances.

11 L'interface utilisateur

11.1 Menu

11.1.1 Les bases

Les menus ont été construits sur une base dynamique. Il est en effet possible de créer des menus à partir d'un fichier. Ce fichier va ensuite être parcouru, et les éléments comme le nombre de boutons, ou encore la position des boutons à l'écran va être attribué à l'élément concerné. Il est donc facile à partir de cette architecture de créer des menus simples contenant un arrière-plan, plusieurs boutons avec des textes différents, éventuellement une action associée à chacun des boutons (déclenchement d'un script ou exécution d'une commande).

Voici comment se présente un menu configuré depuis un fichier :

```
set buttoncount 4
bindkey escape "multiple (inmenu 0) (bind escape (inmenu 1) 1 1) (joybind Back 0 (inmenu 1) 0 1) (joybind Back 1 (inmenu 1) 0 1) (joybind Back 2 (inmer
joybind all Back "multiple (inmenu 0) (bind escape (inmenu 1) 1 1) (joybind Back 0 (inmenu 1) 0 1) (joybind Back 1 (inmenu 1) 0 1) (joybind Back 2 (inm
set 0.relative.x 50
set 0.relative.y 20
set 1.relative.x 50
set 1.relative.y 40
set 2.relative.x 50
set 2.relative.y 60
set 3.relative.x 50
set 3.relative.y 80
setstring 0.text "Back To Game"
setstring 1.text "Options"
setstring 2.text "Restart Level"
setstring 3.text "Main Menu"
setcolor 0.color 211 0 160
setcolor 1.color 211 0 160
setcolor 2.color 211 0 160
setcolor 3.color 211 0 160
setcolor 0.highlight 0 129 198
setcolor 1.highlight 0 129 198
setcolor 2.highlight 0 129 198
setcolor 3.highlight 0 129 198
exec 0 "multiple (inmenu 0) (timescale 1)"
exec 1 "isinconfig 1"
exec 2 "multiple reloadlevel (inmenu 0)"
exec 3 "setmenu main.menu"

```

FIGURE 22 – Edition d'un menu

Nous avons ici le fichier de configuration d'un menu. Voici l'interprétation qu'en fait le jeu :

- La première ligne indique le nombre de boutons à créer.
- La seconde ligne permet d'affecter une certaine action à effectuer lorsqu'une touche particulière du clavier est appuyée, ici cela va permettre de réinitialiser les touches permettant de rentrer dans ce menu, puis de le fermer.
- La troisième ligne permet le même procédé que la précédente, mais à partir d'une touche de manette.

— Les 6 lignes qui suivent permettent le placement des boutons à l'écran. La présence du mot clef « relative »permet de les placer à une certaine position qui va varier en fonction de la résolution de l'écran. Cette position est définie en x et en y.
— Ensuite, on attribue un texte à chacun des boutons.
— Une couleur leur est également assignée. Celle-ci se fait en deux temps : le couleur « normal »et la couleur « highlight ». La première sera utilisée dans le cas où le bouton n'est pas sélectionné, la seconde dans le cas contraire.
— Enfin, on attribue une certaine commande au bouton. Celle-ci sera exécutée lorsque la touche *Entrée* sera appuyée lorsque le bouton sélectionné.

Voici donc comment se présente un fichier de configuration d'un menu, et ci-dessous le résultat obtenu après interprétation par le jeu de chacune des lignes du fichier présenté :

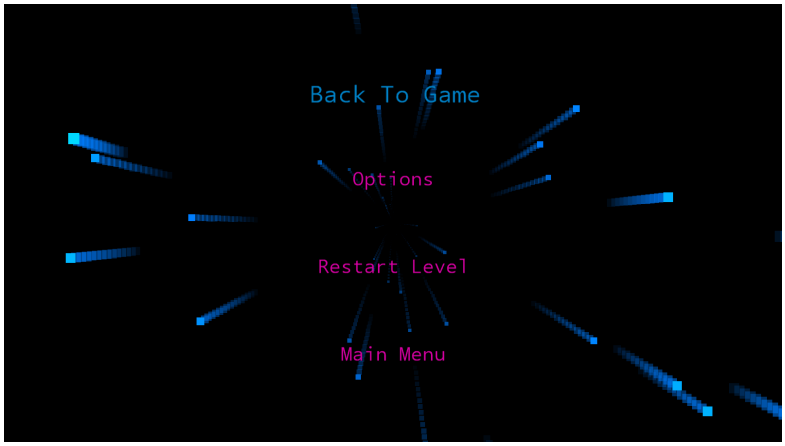

FIGURE 23 – Menu pause

L'élément principal de nos menus est la classe *Bouton*. Celle-ci a plusieurs attributs comme montrés plus haut : une position, deux couleurs, un booléen, un texte ainsi qu'une commande ou un script associé. Cette classe possède cependant plusieurs méthodes : un constructeur, une méthode *Draw*,une méthode *Finish* et une méthode *Update*. Le constructeur permet d'initialiser un nouveau *Bouton* et la méthode *Draw* permet de dessiner le Bouton à l'écran. La méthode *Finish* permet de calculer le centre du *Bouton*, cela va donc permettre de réellement centré le *Bouton* sur un certain point de l'écran. Ainsi, le Bouton ayant pour coordonnées (10,10) aura en réalité des coordonnées de départs différentes mais sera centré en (10,10). La méthode *Update*, quant à elle, permet d'appliquer un effet de zoom sur le bouton ainsi que le changement de couleur. En effet, lorsque l'on sélectionne un bouton, celui-ci va s'agrandir, tandis que celui qui vient d'être désélectionné va rétrécir et reprendre sa couleur normal.

On voit donc que le fichier de configuration est parcouru, et le jeu va vérifier que chacune des données entrées est correcte. Seulement cette base a été utile pour les premiers menus dont nous avions besoin, mais elle est vite devenue insuffisante, et nous avons eu besoin de créer des menus plus complets.

11.1.2 Menu de sélection des touches

Le menu de sélection des touches est un bon exemple de menu qui a eu besoin d'être développé séparément. En effet, la configuration des touches est quelque chose de propre à chaque joueur. Ainsi, pour une meilleure expérience de jeu, nous avons tenu à ce que le joueur puisse choisir sa façon de jouer. Cela s'est à nouveau confirmé avec la possibilité d'utiliser la manette.

Pour jouer au clavier, la méthode est simple : d'abord le joueur choisit les touches qu'il veut utiliser. Celles-ci s'affichent sur la droite de l'écran pour que le joueur puisse avoir un aperçu de ce qu'il vient de sélectionner. Ensuite, lorsque le joueur va quitter le menu, ce dernier va associer chacune des touches à une commande du jeu.

Pour pouvoir jouer avec la manette, il y a eu une contrainte supplémentaire : il a d'abord fallu identifier la manette qui allait être utilisée, car le joueur 1 peut choisir de jouer avec la manette 2 alors que le joueur 2 va utiliser la manette 1. Cette identification peut se faire de deux façons : soit le joueur utilise déjà la manette, auquel cas, il suffit de récupérer cet identifiant qui a été préalablement enregistré Une fois l'identification de la manette faite, tout se déroule de manière similaire qu'au clavier. La seule différence demeure dans le fait que nous avons choisi de représenter les touches de la manette de façon graphique, nous avons donc associé une image à chacune des touches de la manette.

11.1.3 Menu de configuration graphique

Le menu de configuration graphique a également nécessité une implémentation particulière pour deux raisons. Premièrement, il était nécessaire de connaître les résolutions disponibles et adaptées pour le matériel utilisé par le joueur. Pour cela, une fonction se charge de récupérer les différentes résolutions disponibles sous forme de tableau. De plus, nous avions besoin de changer certaines valeurs comme, encore une fois, la résolution à utiliser. C'est pourquoi nous avons mis en place un système de boutons dont la valeur change en utilisant les flèches droites et gauche du clavier. Nous avons également intégré

dans ce menu la possibilité de mettre le jeu en plein écran, de changer la qualité du feu, élément gourmand en ressource, ainsi que d'activer ou de désactiver l'anticrénelage. Une fois ces paramètres modifiés, il suffit d'appliquer les changements.

11.1.4 Menu multijoueur

Le menu multijoueur est arrivé plus tard dans l'année et a permis à plusieurs joueurs de jouer ensemble. Ici, nous avons choisi de réutiliser le système de boutons cité précédemment. Il est ici possible de choisir le nombre de joueurs qui vont participer au jeu. Il faut savoir que nous proposons trois types de caméras décrites dans la partie *Multijoueur local*.

11.1.5 Menu de sélection de niveau

Un élément important dans un jeu est de pouvoir choisir le niveau sur lequel l'utilisateur va jouer. C'est pourquoi nous avons créé le menu de sélection de niveau. Celui-ci repose sur le principe suivant :

- Il parcourt le dossier où se situent les différents niveaux.
- A l'aide d'un utilitaire qui va permettre de pré-visualiser le niveau, il va afficher la texture qui lui correspond. Cette texture, lorsqu'elle a été calculée est ensuite enregistrée. En effet, le calcul de chacun des niveaux peut prendre du temps, l'enregistrement de ces textures permet donc à l'utilisateur d'économiser du temps au chargement de ces textures.
- Une fois ces textures créées, elles sont affichés dans l'ordre avec le nom du niveau qui lui correspond.

Ce niveau apparaît lorsque le joueur veut choisir un nouveau niveau et lorsqu'il lance une nouvelle partie que ce soit en mode solo ou en multijoueur local.

11.1.6 Menu de sélection d'un serveur

Après l'ajout du mode multijoueur en ligne, la création d'un menu pour pouvoir sélectionner le serveur auquel on veut accéder s'est imposé d'elle-même. Pour cela, nous avons conçu à nouveau un nouveau type de bouton, il s'agit de boutons rectangulaires contenant des données en texte brut à l'intérieur comme l'adresse IP du serveur, ainsi que le port utilisé. Nous avons choisi d'y renseigner le nombre de joueurs connectés et le nom du niveau que le serveur exécute.

Le joueur a également la possibilité d'entrer manuellement l'adresse ip et le port du serveur auquel il désire se connecter. Il lui suffit simplement d'appuyer sur les touches décimales de son clavier pour entrer l'adresse, puis d'appuyer sur entrée pour s'y connecter.

11.2 Le HUD

Le HUD comprend la vie des personnages. Celle-ci est représentée par une barrette de RAM qui diminue au fur et à mesure que le personnage prend des dégâts. Cette barre s'adapte également au mode multijoueur. En effet, lorsque la caméra est en mode Zoom, les barres vont se superposer. Tandis qu'en mode écran partagé, chacun des joueurs aura sa barre de vie représentée dans sa partie de l'écran.

Cependant, la barre de RAM se modifie de façon dynamique. En effet, lorsque le personnage est attaqué, et donc qu'il perd de la vie, la barre représentant celle-ci va diminuer progressivement. Ainsi, si une barre représente 10 points de vie, lorsque le joueur va perdre 4 de ces points, la texture correspondant ne va être dessinée qu'à 6 dixième. De la même façon, en gardant cette base de 10 points de vie pour une barre de RAM, si un joueur n'a plus que 58 point de vie, 5 barres seront totalement représentées et les 8 dixième seulement de la dernière seront dessinés à l'écran.

12 Les outils de développement

12.1 La console en jeu

En multijoueur, lorsque l'on veut communiquer avec les autres joueurs, il est peu commode de devoir ouvrir la console extérieure. Il a donc fallu trouver un compromis entre la console et le jeu. Nous avons alors décidé de faire une console dans le jeu, qui apparaîtrait en bas à gauche de l'écran, et qui serait invoquée lors de la réception d'un nouveau message ou bien lorsque le joueur l'invoque grâce à une touche configurée (*Entrée* par défaut).

Cette console prend la forme d'une classe. Elle a une méthode d'entrée, qui prend en paramètre une chaîne de caractères ainsi que le nom de la personne, ou du thread qui écrit le message. Celle-ci va rajouter ce dernier à une file de chaîne de caractères représentant la conversation. Les éléments de cette file sont donc dessinés sur l'écran dans l'ordre chronologique.

Une méthode permet aussi de commencer la saisie : c'est celle appelée grâce à la touche de conversation (*Entrée*). Toutes les entrées clavier sont redirigées vers la console, dans laquelle le joueur voit son texte apparaître au fur et à mesure de l'écriture de son message, jusqu'à ce qu'il appuie à nouveau sur la touche de conversation pour envoyer le message au serveur et le rajouter à la console, ou bien sur la touche *Echap* pour annuler la saisie.

12.2 L'éditeur de niveau

L'éditeur de niveau a rapidement été développé, en effet, un outil nous permettant de créer rapidement et facilement des niveaux est vite devenu obligatoire durant le développement du jeu.

12.2.1 L'interface

Dans l'optique de créer un outil simple et puissant à utiliser, nous avons réalisé un croisement entre une fenêtre graphique et une fenêtre utilitaire. Cette interface est composée de plusieurs onglets permettant de rajouter au niveau tous les éléments présents dans le jeu.

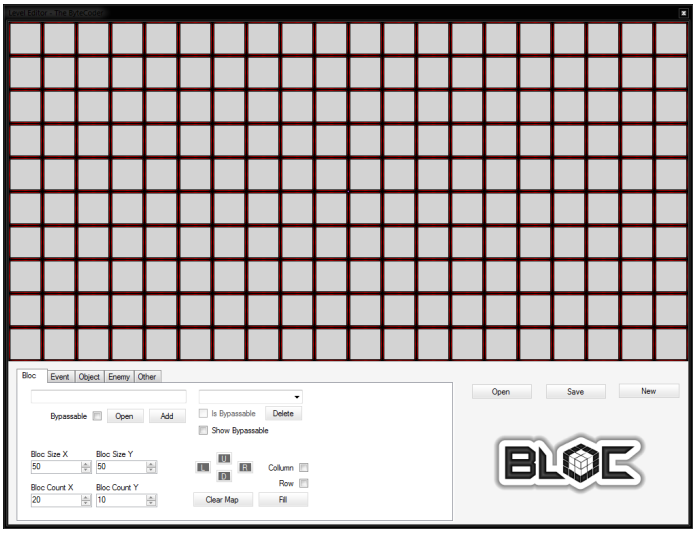

FIGURE 24 – Interface principale de l'éditeur de niveau.

12.2.2 Son fonctionnement

Pour utiliser cet éditeur, il suffit de rajouter dans les différents champs proposés les fichiers à importer. Cela étant fait, il ne reste plus qu'à placer ces éléments sur la carte de jeu. Au cours de la création, il sera toujours possible d'ajouter de nouveaux éléments ou d'en supprimer.

Après avoir conçu la carte de vos rêves, il ne reste qu'une chose à faire, la sauvegarder pour pouvoir l'utiliser dans le jeu.

12.3 Les outils de débogage

Les outils de débogages sont des moyens de faciliter l'identification et l'analyse de bogues. Ce projet étant un projet de groupe, il est possible qu'un membre découvre un problème qui provient d'un autre membre. Pour cette raison, il faut qu'il puisse lui envoyer un rapport qui permettra d'identifier et de corriger le problème.

Ce système de rapports de problème se traduit par la mise en place d'une console, qui affiche des informations sur les valeurs des variables, les commandes exécutées, affiche les éventuelles erreurs et enregistre tout ce qui est écrit dans un fichier texte. La console permet également d'entrer des commandes.

12.4 Les couleurs TSL

Sur un écran, les couleurs sont affichées en fonction de trois composantes, rouge, vert et bleu (couleurs RVB). Ce type de définitions de couleur ne permet pas directement de faire des dégradés, qui sont utiles par exemple pour représenter des flammes (avec un dégradé du jaune vers le rouge). La solution à ce problème est d'utiliser les couleurs TSL, qui sont représentées par les trois composantes suivantes :

— La teinte : Représente l'angle en radians sur le cercle trigonométrique, compris entre 0 et $2 * \pi$.

— La saturation : Représente la différence entre la composante la plus élevée et la composante la plus faible de la couleur RVB équivalente.

— La luminosité : Représente la moyenne entre la composante la plus élevée et la composante la plus faible de la couleur RVB équivalente.

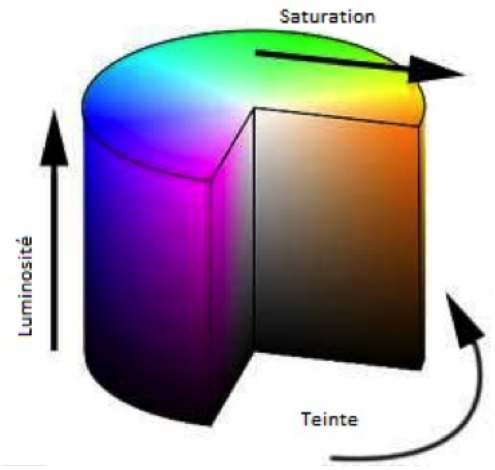

FIGURE 25 – Schéma du TSL

12.5 Le système de traduction

Dans ce projet, l'utilisateur peut choisir la langue. Cela s'applique aux menus et aux dialogues. Pour stocker les chaines de caractères, nous avons choisi d'utiliser le format XML (*Extensible Markup Language*). Chaque langue est stockée dans un fichier. Les langues disponibles sont le Français et l'Anglais, on dispose donc d'un fichier « fr.xml »et d'un autre fichier « en.xml ». Les fichiers sont des représentations de tables de hachage avec en clé le nom de la phrase et en donnée la phrase elle-même. Pour faciliter la rédaction de phrases, nous avons développé un outil qui nous permet de créer et de modifier ces fichiers XML à l'aide d'une interface graphique.

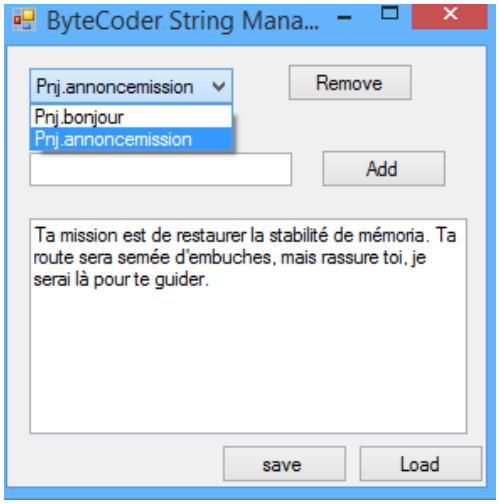

FIGURE 26 – L'outil de modification de fichiers XML

12.6 Optimisation des performances

13 La gestion des ressources

Les ressources sont les éléments qui prennent le plus de place en mémoire. Pour cette raison, il faut s'assurer de ne charger chaque ressource qu'une seule fois et de la supprimer seulement si aucun composant du jeu n'en a besoin.

Les textures sont les ressources les plus utilisées par les composants de ce projet. Un système de centralisation et d'organisation des textures a donc été mis en place. Toutes les textures sont gérées par un seul objet. Cet objet contient :

- Une table de hachage avec en clé le chemin vers la texture et en donnée une structure contenant la texture et un entier représentant le nombre d'utilisateurs de cette texture.
- Une autre table de hachage avec en clé une texture et en donnée le chemin vers le fichier.
- Une texture par défaut, la texture d'erreur.

Pour obtenir une texture, on utilise la fonction LoadTexture, qui prend en paramètre une chaîne de caractères(le chemin vers le fichier). On regarde ensuite dans la première table de hachage si cette chaîne de caractères existe en tant que clé. Si oui, on ajoute un au nombre d'utilisateurs de cette texture et on retourne cette texture. Si non, on essaye de charger la texture, si le chargement réussi, on crée une nouvelle structure,

avec cette texture récemment chargée et 1 comme nombre d'utilisateurs. On ajoute ensuite cette instance de structure à la première table de hachage. On ajoute ensuite un nouvel élément à la deuxième table de hachage avec comme clé la texture et comme valeur le chemin vers le fichier. Si le chargement de la texture échoue, on affiche un avertissement dans la console et on effectue les mêmes étapes avec la texture d'erreur.

Lorsqu'un composant du jeu n'a plus besoin d'une texture, il appelle la fonction DisposeTexture, qui prend en paramètre une texture. La fonction vérifie que cette texture est bien associée à une valeur dans la deuxième table de hachage. Si elle ne l'est pas, la fonction affiche un avertissement dans la console et s'arrête là. Si elle est bien présente, on utilise cette table de hachage pour obtenir le chemin du fichier que l'on utilise pour obtenir le nombre d'utilisateurs en utilisant la première table de hachage. On décrémente cet entier. On regarde ensuite sa valeur, si celle-ci est de zéro, cela signifie qu'aucun composant du jeu n'a besoin de cette texture, on peut donc la supprimer.

Seul le thread principal est autorisé à charger ou à supprimer des ressources. Il n'est donc pas possible qu'une texture soit supprimée pile en même temps qu'elle soit chargée (ce qui provoquerait une erreur).

13.1 L'importance de l'instanciation

L'instanciation est une technique permettant d'optimiser les performances d'un élément qui apparaît plusieurs fois dans un programme. Par exemple, un niveau contenant 100 ennemis qui n'apparaissent pas en même temps. L'ennemi est constitué d'un certain nombre d'images et de variables qu'il faut charger pour qu'il puisse apparaître à l'écran. L'instanciation consiste à ne charger qu'une seule fois toutes les images et variables de cet ennemi et à tout garder en mémoire. Lorsqu'un ennemi apparaît, il n'est donc pas nécessaire de recharger toutes les ressources, il suffit de lui passer les références des images et des variables. Ce système permet d'économiser la taille en mémoire d'un objet instancié plusieurs fois.

Dans ce projet, l'instanciation se traduit par un système de modèle qui, lors du chargement du niveau va charger tous les composants nécessaires à l'instanciation de l'objet associé. Cette méthode est utilisée par les ennemis, les objets scriptés et les particules.

14 Communication

14.1 Le site web

Lorsque nous avons entrepris de créer le site web du jeu, nous nous sommes demandé à quoi il pourrait servir. Après une réflexion commune, nous avons décidé de diviser le site en différente sections ayant chacune un sujet : l'actualité, la découverte du jeu, nous contacter ou encore télécharger le jeu.

Nous avons réalisé le site à l'aide des langages classiques du web, le trio HTML/CSS/-Javascript pour la partie client et le PHP pour la partie serveur. Pour le serveur nous avons décidé d'héberger nous-même le site sur nos serveurs personnels. Le site est accessible sous le domaine de blocsoft.net.

Le site est composé des pages suivantes accessibles n'importe où dans le site :

— La page d'accueil, composée de la vidéo de présentation du jeu sur un écran dessiné en CSS ainsi qu'un bouton de téléchargement du jeu
— La page de présentation qui présente le gameplay et l'univers de notre au jeu au visiteur du site
— La page de téléchargement, où l'on peut télécharger le jeu ainsi que les trois rapports de soutenance, le cahier des charges et différents médias en rapport avec le jeu
— La blog, qui est régulièrement mis à jour, et où le visiteur peut suivre l'avancée du projet ainsi qu'être au courant du nouveau et futur contenu du jeu
— La page de contact, où on peut trouver des informations sur notre groupe ainsi que les moyens pour nous contacter

14.2 Les réseaux sociaux

La page d'accueil et la page de contact de notre site ont également des liens vers nos pages sur les deux réseaux sociaux Twitter et Facebook. On peut suivre l'actualité et l'avancement du projet en suivant notre profil Twitter ou bien en aimant notre page Facebook.

@blocsoftware ou twitter.com/blocsoftware pour Twitter
facebook.com/blocsoft pour Facebook

15 Conclusion

Nous voici donc arrivés à la fin du projet. Après un peu plus d'un semestre de travail continu, nous avons la fierté de pouvoir présenter le jeu complet. Au moment de la rencontre des membres du groupe, nous avions très vite réalisé que nos attentes étaient similaires. L'idée du ByteCoder s'était toute de suite imposée.

Aujourd'hui, nous avons le résultat que nous avions imaginé quelques mois auparavant. Le système de jeu dont nous avions rêvé a pris forme, et nous pouvons le faire découvrir à tous nos proches. L'univers et l'histoire, qui s'est doucement amenée à nous au cours du projet, sont également des éléments du jeu qui nous permettent d'exprimer nos idées initiales.

Tout ceci ne s'est pas fait sans difficulté, au cours du projet nous avons été confrontés de nombreux problèmes techniques et humains. Même si nos idées étaient similaires au début, nos différents points de vue se sont parfois croisés et il a fallu trouver des compromis. Mais si l'on se réfère à l'adage bien connu « la différence fait la force », on peut considérer la contribution personnelle de chacun comme bénéfique au résultat final.

Les différents problèmes techniques rencontrés ont également permis de nous faire progresser en programmation. Ce sont ces difficultés qui nous ont poussé à chercher les solutions les plus adaptées et qui nous ont donné l'occasion de mettre en pratique toutes nos connaissances algorithmiques et scientifiques. Ce projet a aussi été pour nous une occasion de nous améliorer en C# et d'apprendre à utiliser la bibliothèque logicielle qu'est XNA.

Nous avons eu la chance d'avoir une équipe soudée du début à la fin. Dès le début, nous nous sommes organisés en nous définissant tous un rôle au sein du groupe. Cela nous a appris à travailler en équipe, respecter une date précise et organiser son travail personnel avec le projet dans son ensemble. Ainsi, nous avons atteint les objectifs fixés au commencement d'une manière bien plus enrichissante que l'on aurait pu imaginer au départ.

16 Conclusions Personnelles

Serge « Kh4L »Panev

Ce projet a été pour moi une expérience très enrichissante autant sur le plan technique que sur le plan relationnel. Ayant déjà participé à de nombreux projets de groupe, je n'ai jamais eu l'opportunité de prendre part à un projet aussi complexe que ByteCoder. Grâce à ce jeu, j'ai exploité tout ce que j'ai appris cette année à Epita. Que ce soit une structure de donnée convenable à trouver pour un problème, un problème de physique, ou encore pour comprendre la documentation XNA, plus complète en anglais qu'en français.

Tout au long du projet, voir le jeu prendre forme, au fil des séances de code, a été un véritable plaisir. Malgré tous les problèmes techniques que nous avons rencontrés, il a toujours été très intéressant d'en discuter afin de trouver la solution la plus adaptée aux besoins. Au cours de l'année, j'ai beaucoup progressé en C#. En effet, en plus de l'algorithmique formelle et des travaux pratiques, le fait de devoir mettre en application des idées personnelles m'a beaucoup aidé dans l'apprentissage du langage. De plus j'ai dû faire de nombreuses recherches sur des sujets que je n'avais pas forcément envisagés dès le départ, comme la conception d'intelligence artificielle ou bien la programmation réseau.

La rencontre avec les autres membres du groupe s'est faite naturellement. Nos passions communes et notre vision similaire de ce à quoi un jeu doit ressembler ont fortement contribué à ce choix. Il a fallu dès lors répartir le mieux possible le travail dans le groupe, afin de mettre les points forts de chacun en valeur. Une fois ces règles établies, j'ai trouvé très intéressant de voir le projet se vêtir des talents de chacun.

Maintenant que le projet est terminé, je me rends compte que ce n'était pas un groupe de projet que nous avons eu, mais un groupe d'amis se donnant à cœur joie afin de créer le meilleur jeu possible. J'écris ces lignes avec une certaine nostalgie, peut-être anticipée, mais caractéristique d'une aventure qui s'achève. Tout en sachant que nous allons continuer ce projet et en avoir encore beaucoup d'autres.

Pierre « BLUESCREEN »Boulay

Lorsque nous avons formé notre groupe en début d'année, je n'avais aucune connaissance du langage C # et de la bibliothèque XNA. Je me rends aujourd'hui compte de tout ce que j'ai pu apprendre, sur le code lui-même, sur la manière de l'écrire pour qu'il soit efficace, mais aussi compréhensible par les autres membres du groupe. Je n'avais jusqu'alors jamais participé à un projet d'équipe de cette envergure, et aujourd'hui, alors que la conclusion approche, je me sens fort de l'enrichissement que ce projet m'a apporté à moi-même ainsi qu'à toute l'équipe.

Ensemble, nous avons appris que l'union fait la force. Je suis heureux de voir que notre équipe a réussi à respecter ses objectifs, et est devenue aussi soudée qu'efficace. Nous avons certes connu des difficultés et des désaccords mais nous avons toujours su les résoudre de manière réfléchie lors de nos réunions. Chaque problème que nous avons rencontré fut une leçon dont nous nous souviendrons dans nos projets respectifs à venir. Nous avons partagé nos connaissances ce qui nous a permis d'aboutir à un résultat final qui est l'image des idées que nous avons eues.

Au-delà du développement du projet en lui-même, j'ai appris qu'un jeu vidéo n'est pas seulement composé de code. Il faut aussi rédiger des rapports, répartir le travail, savoir quand commencer à le rédiger, prendre des notes tout au long du projet pour ne rien oublier tout au long de sa redaction, etc. Je me sens particulièrement enrichi par les trois rapports que nous avons écrits. LaTeXpermet de rédiger des documents de qualité et cela a pour moi été une formidable découverte.

En tant que chef de projet, ce fut pour moi un honneur de mener cette équipe à ce résultat. Je suis fier de voir à quel point nous sommes devenus efficaces et méthodiques pour atteindre nos objectifs et, même s'il nous reste encore beaucoup à apprendre, je garderai un agréable souvenir de ce projet. J'éprouve néanmoins un sentiment de mélancolie en vue de la dernière soutenance qui sera le chapitre final de cette formidable aventure qu'a été pour nous The ByteCoder.

Alessandro « d0t_iKs » Pisu

N'ayant jamais eu de réelle occasion de travailler sur un projet avec d'autres personnes, le développement de ce jeu fut très instructif.

La première chose que cette expérience m'a apportée doit être l'opportunité d'utiliser la bibliothèque de Microsoft, XNA, en effet au moment où le développement à commencé, cette solution allait bientôt être officiellement mise de côté. Or utiliser cet outil fut d'une grande aide dans mon apprentissage du C# et de la façon de développer avec une équipe.

Concernant le projet, je me suis principalement occupé de l'élaboration des différents éléments graphiques du jeu, ainsi que du développement de l'éditeur de niveau. Les graphismes furent sûrement le plus fastidieux et le plus long à faire. Pour chaque image, j'ai dû vérifier que les dimensions soient constantes pour chaque image, ce qui n'est pas intuitif. De plus représenter des mouvements différents avec aussi peu de pixels n'est pas aisé.

En ce qui concerne le travail de groupe ce fut, un véritable défi de matérialiser correctement toutes les idées que les autres membres du groupe avaient en tête. Le plus long à représenter a dû être l'ennemi scripté « Apache » qui a nécessité plus d'une trentaine d'images différentes.

La conception de l'éditeur de niveau s'est effectuée en plusieurs étapes, la première consista à créer un outil avec plusieurs fenêtres séparées et permettant de construire la base d'un niveau. La seconde fut de rassembler ces fenêtres pour obtenir un éditeur plus simple à utiliser. Enfin après avoir obtenu un logiciel avec une structure stable, il ne restait qu'à rajouter les dernières fonctionnalités.

Finalement, travailler en équipe sur un projet aussi conséquent et étalé dans le temps, m'a démontré que la plupart des problèmes rencontrés lors de cette période furent des problèmes humains, comme par exemple des quiproquos ou des avis non partagés. Dans tous les cas, il fallait trouver une solution ralliant tous les membres dans un temps asez court. Le résultat qui sort de cette expérience reste tout de même une meilleure cohésion du groupe et un exemple concret d'élaboration d'un projet.

Paul-Henry « Dr_c0w » Perrissel

Ce projet a été très intéressant pour moi. En effet, c'était le premier projet de groupe auquel j'ai participé, et c'était également l'occasion d'apprendre le C# ainsi que l'utilisation de XNA, la bibliothèque graphique que nous avons choisi d'utiliser.

D'un point de vue programmation, j'ai eu l'opportunité de découvrir le C#, avec notamment la gestion des erreurs qui y est plutôt facilité face à d'autres langages de programmation. Mais j'ai surtout eu l'occasion de découvrir le fonctionnement de la lecture de fichier. Il a fallu, par conséquent, gérer les soucis de données manquantes, mais surtout s'assurer que si une donnée est manquante, cela ne doit pas empêcher le jeu de fonctionner, et mon devoir constituait donc à limiter au maximum ces erreurs. J'ai également rencontré quelques soucis avec XNA de par le fait que c'était une bibliothèque nouvelle pour moi. Heureusement, mes camarades étaient dans le même cas que moi, et il a donc été plus facile pour tout le monde de s'entraider. Même si certains ont appris plus vite que d'autres, notre groupe est toujours resté soudé.

Toujours d'un point de vue programmation, cette année a été marquée par l'apprentissage du LaTeX. J'ai donc dû me familiariser avec cet outil performant pour la rédaction de nos rapports. Evidemment, nous y avons tous mis du nôtre afin d'être plus performant sur la rédaction de ces derniers.

Ce projet a également permis de découvrir l'univers du travail en équipe. Comme je l'ai dit plus haut, pendant ce semestre, nous avons formé un groupe où l'entraide était vraiment présente. Dès que l'un d'entre nous rencontrait une difficulté, il y avait au moins un autre membre du groupe qui était prêt à l'aider. Il y a eu certes quelques désaccords, mais cela n'a pas empêché qu'il y règne une très bonne ambiance entre nous quatre.

Au niveau du projet en lui-même, j'ai participé, entre autre, à l'élaboration du scénario. Cela a donc commencé par un premier *brainstorming* pour définir le cadre du jeu ainsi que la trame principale. A partir de ce moment-là, je me suis occupé de l'histoire. Nous avions déjà choisi les niveaux principaux et l'atmosphère des mondes que nous allions créer, j'ai donc permis de lier ces mondes ensemble, avec leurs boss respectifs et leurs ennemis emblématiques.

Ce projet a donc été pour moi quelque chose de très instructif, que ce soit au niveau de la programmation ou au niveau du travail en groupe. Il y a de fortes chances pour que nous travaillions à nouveau ensemble mais probablement sur une plateforme différente et dans un langage différent. En effet, l'ambiance du groupe étant très bonne, nous ressortons plus soudés que nous l'étions avant le début du projet.

17 Webographie

— YouTube [En ligne] Google [Consulté au mois de novembre]
Disponible sur : http ://www.youtube.com/watch ?v=g-JupOxwB-k
Tutoriel pour la bibliothèque XNA

— MSDN [En ligne] Microsoft [Consulté au moins de novembre]
Disponible sur :
http ://xbox.create.msdn.com/en-US/education/catalog/sample/winforms_series_1
Tutoriel pour XNA

— Stack Overflow [En ligne] StackExchange [Consulté toute l'année]
Disponible sur : http ://stackoverflow.com/questions/tagged/c%23
Forum pour la programmation C#

— AcdcEpita [En ligne] ACDC Epita [Consulté toute l'année]
Disponible sur : http ://perso.epita.fr/ acdc/
TP de C# d'Infosup

— Algo La Page [En ligne] Nathalie Bouquet [Consulté toute l'année]
Disponible sur : http ://algo-td.infoprepa.epita.fr/
Cours de Chistophe Boullay et Nathalie Bouquet

— MasterCorp [En ligne] Inc [Consulté au mois d'avril]
Disponible sur : http ://mastercorp.epita.eu/isup/cours/maths
Cours et annals de l'Epita

— Teinte-Saturation-Lumière [En ligne] Wikipédia [Consulté au mois d'avril]
Disponible sur : http ://fr.wikipedia.org/wiki/Teinte_saturation_lumi%C3%A8re
Page encyclopédique sur le système de couleur TSL