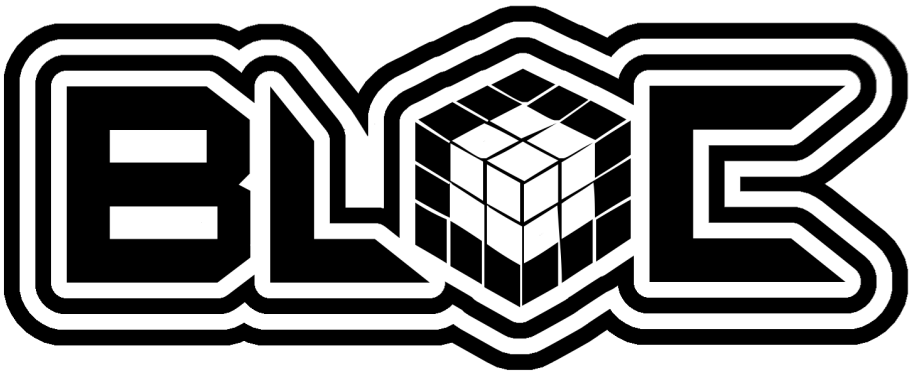

Rapport de 1re Soutenance « The ByteCoder »

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

Table des matières

1 Avant-propos **3**
1.1 Les Variables du Jeu . . . 3

2 Les Scripts **3**
2.1 L'interpréteur . . . 3
2.2 Objets scriptés . . . 5
2.3 Les événements . . . 8
2.4 Gestions des Collisions . . . 8
2.5 Gestion des niveaux . . . 10
2.6 Gestions des Ralentis . . . 10

3 Outils **12**
3.1 L'importance de l'instanciation . . . 12
3.2 La gestion des contrôles . . . 12
3.3 Les outils de déboguage . . . 13
3.4 Les couleurs TSL . . . 14
3.5 La caméra . . . 14

4 Les Graphismes **16**
4.1 L'Univers Graphique . . . 16
4.2 Le design du protagoniste . . . 17
4.3 Un ennemi coriace : le Dalek . . . 18
4.4 Petit mais efficace : le bot FlashPlayer . . . 18

5 Les Ennemis **18**
5.1 Le cœur de l'intelligence artificielle : l'Enemy Manager . . . 19
5.2 Un modèle pour chacun : l'EnemyModel . . . 19
5.3 L'Algorithme principal : l'Enemy . . . 21
5.4 Une arme à part entière : FarWeapon . . . 21

6 Les Effets Graphiques **22**
6.1 Les Particules . . . 22
6.1.1 « Particle » . . . 22
6.1.2 « ParticleModel » : le socle de l'effet . . . 22
6.1.3 L'effet : « ParticleManager » . . . 23
6.2 Le feu . . . 24
6.2.1 « FireParticle » . . . 24
6.2.2 « LittleFire » . . . 24
6.2.3 « FireManager » . . . 24
6.3 « SpawnEffect » . . . 26
6.4 L'Animation de la Mort . . . 27
6.4.1 « DeathParticle » . . . 27
6.4.2 « DeathManager » . . . 27

7 L'Éditeur de Niveau **28**
7.1 Son principe 28
7.2 L'Interface 28
7.3 Le Fonctionnement 31
7.3.1 Les fichiers « .lvl » 31
7.3.2 Sauvegarde des différents blocs 32
7.3.3 Sauvegarde d'une carte de références 32

8 Les Menus **33**
8.1 Edition d'un menu depuis un fichier 33
8.2 Configuration des graphismes 35
8.3 Configuration des touches 35
8.4 Résumé de la classe bouton 35
8.5 Fond Dynamique 36

9 Expérience personnelle **37**

1 Avant-propos

Avant de commencer la programmation d'un jeu de plateforme, il est primordial de réfléchir aux méthodes qui permettent de rentabiliser au maximum le temps de travail. La flexibilité du code est l'assurance de pouvoir effectuer facilement des modifications sur l'ensemble du projet sans avoir à tout recommencer.

1.1 Les Variables du Jeu

En conséquence, ce projet utilise un système de variables de jeu centralisées. Une variable de jeu est une valeur accessible par tous les composants du projet, elles peuvent par exemple représenter la gravité du niveau, le zoom de la caméra, la limite d'images par secondes, etc. Pour optimiser le temps d'accès aux valeurs de ces variables, ce système utilise une table de hachage. Ce dernier permet de savoir où se trouve la valeur d'une variable avec seulement son nom, sans avoir à parcourir toutes les valeurs d'un tableau jusqu'à trouver la bonne.

Les variables de jeu sont accompagnées d'un système de commandes qui permettent d'une part de les modifier, mais aussi de gérer dynamiquement les paramètres et les éléments de jeu au cours d'un niveau. L'objet qui gère les variables de jeu et les commandes est appelé le « VarManager ».Les commandes peuvent être exécutées directement par l'intermédiaire d'une console, ou par des scripts. Les scripts permettent de dynamiser un niveau, par exemple en faisant apparaître des ennemis lorsque le joueur déclenche un évènement.

2 Les Scripts

2.1 L'interpréteur

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 chaines de caractères. Les lignes sont interprétées de la première à la dernière (c'est plus pratique). 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 apparait 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 apparaitra comme ralenti, voir bloqué. Pour résoudre ce problème, chaque script s'exécute en calcul parallèle (Thread). 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).

2.2 Objets scriptés

De plus, l'interpréteur de script permet d'animer des objets dynamiques qui sont mis à jour en temps réel en se mettant à jour en exécutant des commandes. Le schéma ci-dessous explique le fonctionnement des objets scriptés :

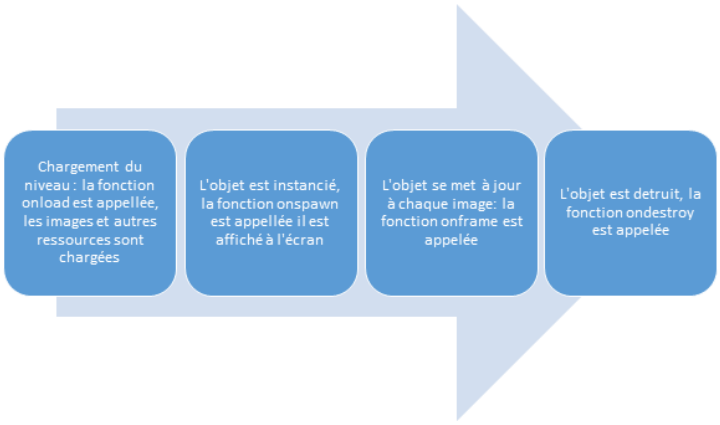

FIGURE 1 – Représentaion de la vie d'un objet scripté

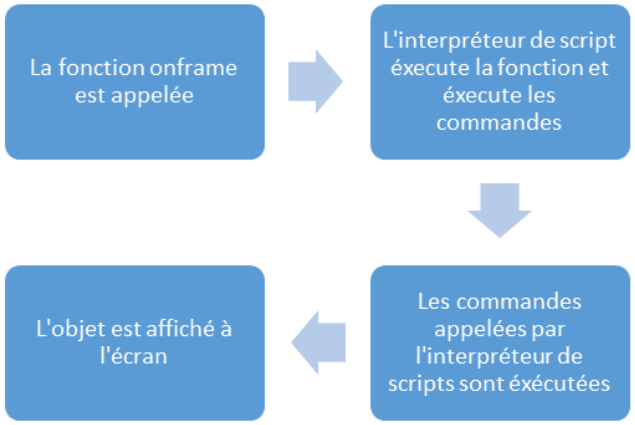

FIGURE 2 – Processus de mise à jour d'un objet scripté

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 celles qui sont liées à l'objet.

FIGURE 3 – Processus d'éxécution d'une commande

2.3 Les événements

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

— Quatre entiers qui représentent un rectangle, lorsque le joueur touche ce rectangle, l'évènement est déclenché.
— Un entier qui représente le nombre de fois que l'événement peut être déclenché avant de disparaître, 0 pour infini.
— Une chaine de caractères, c'est la commande à exécuter lorsque l'évènement est déclenché, par exemple la commande « exec » qui permet d'exécuter 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 apparaissent lorsque l'on passe un certain point, créer des effets de caméra, etc. Les évènements se créent avec la commande : « event » en utilisant la syntaxe suivante :

Event MinX MinY MaxX MaxY Cmd count. Avec Minx, MinY, MaxX et MaxY représentant les quatre points d'un rectangle. La fonction échoue si le rectangle est invalide, ou si des arguments sont manquants.

2.4 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 le pixel 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 être soit vide, soit 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 collision 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 4 – Ici, la collision est un segment, les étincelles seront affichées au centre du segment.

FIGURE 5 – Dans ce cas, la collision est représentée par deux segments, les étincelles seront donc envoyées à l'intersection des deux segments.

FIGURE 6 – La collision est un point, les étincelles seront affichées à partir de ce point.

Néanmoins, les objets du jeu ne peuvent pas 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.

2.5 Gestion des 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. Pour plus de détails référrez vous à la partie sur l'éditeur de niveau.

2.6 Gestions des Ralentis

Le programme effectue un rendu graphique 60 fois par seconde. L'objectif était de pouvoir ralentir le jeu (pour accentuer le côté épique de certains passages du jeu) sans diminuer le nombre d'images par seconde. Le rendu graphique s'organise de la manière suivante :

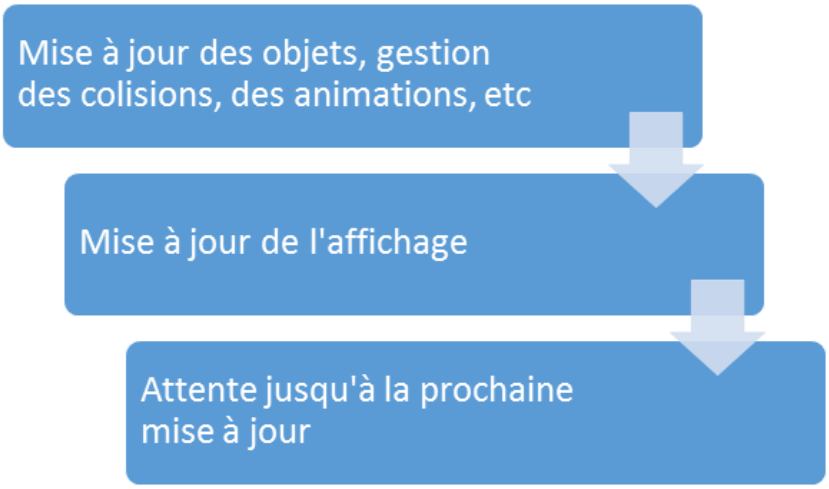

FIGURE 7 – Schéma représentant le proccesus de mise à jour de l'affichage

Pour pouvoir effectuer des ralentis, la solution suivante a été appliquée :

Une variable de jeu de type flottant nommée « timescale » donne la vitesse du jeu par rapport à 1. Une variable de type flottant nommée « framecount » initialisée à zéro compte le nombre d'images en fonction de timescale.

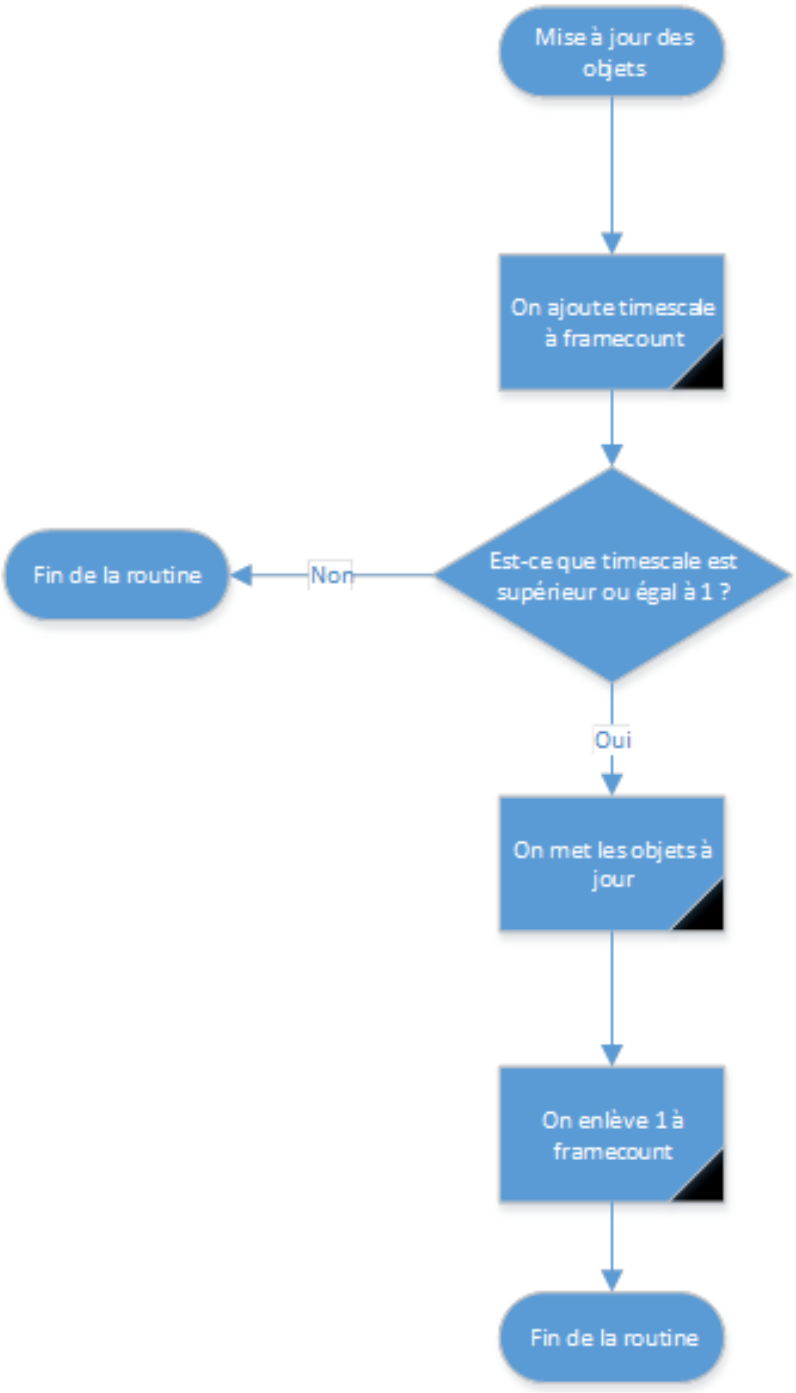

FIGURE 8 – Schéma représentant les mécanismes changeant la vitesse du jeu

3 Outils

3.1 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.

3.2 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 chainée d'enregistrements du type suivant.

```
Enregistrement AssocTouche
    Touche touche
    Booleen relachement
    /* Définit si la touche doit etre relachee
       pour etre de nouveau interpretee */
    Booleen relachee
    /* Definit si la touche a ete relachee */
    Chaine cmd
    /* La commande associee 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 chainée permet l'ajout et la suppression d'associations en jeu.

3.3 Les outils de déboguage

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.

3.4 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 * π.
— 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.

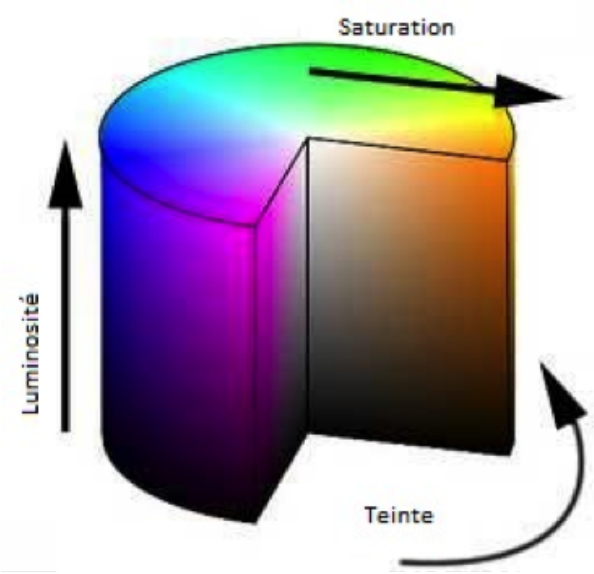

FIGURE 9 – Schéma du TSL

3.5 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).

4 Les Graphismes

Les graphismes dans un jeu video sont sûrement un des aspects les plus importants. En effet, ils servent autant le scénario que les mécaniques de jeu. Quoi de plus ennuyant que des graphismes n'ayant aucun rapport avec ce que l'on fait. L'univers représentant les différentes couches de l'informatique, nous avons opté pour des décors évoquant les composants internes des ordinateurs.

4.1 L'Univers Graphique

Pour représenter l'avancée du joueur dans l'univers, il devra progresser à travers des décors se rapprochant des plus bas niveaux de l'informatique. En effet pour cela dans les premiers chapitres de l'aventure le joueur parcourra des mondes rappelant le web, avec des références aux technologies Flash, ainsi que les mises en page en HTLM5 et CSS. Pour cela nous avons choisi de définir ces chapitres avec des couleurs vives et claires. Ensuite le joueur traversera des mondes de plus en plus sombres, par exemple les mondes suivants évoqueront des logiciels un peu moins intuitifs mais toujours d'assez haut niveau. La patte graphique la plus adaptée paraît être une ville plus ou moins futuriste, avec des teintes métalliques rapellant des oeuvres connues de science-fiction et d'univers d'autres jeux vidéos.

Ainsi il paraît logique que plus le joueur avancera plus il descendra dans les entrailles de cette agglomération, pour représenter le niveau se rapprochant du développement, une ambiance plus industrielle se rapprochant d'usines et de complexes technologiques va être mise en place. C'est pour cela que les graphismes de ces niveaux seront axés sur des réseaux de canalisation, et plus tard sur des mécanismes basés sur des engrenages pour les niveaux les plus sombres, qui évoqueront les parties les plus bas niveaux tel que l'assembleur.

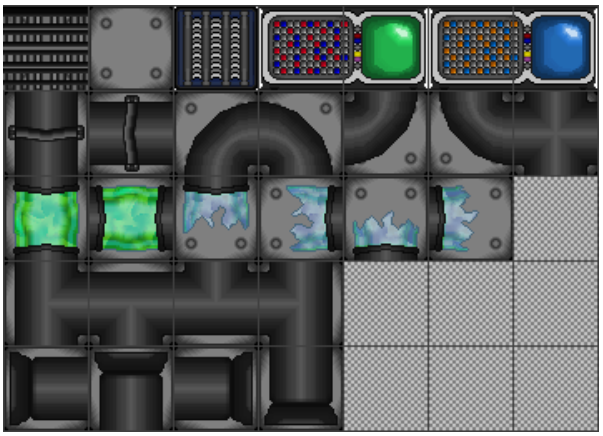

FIGURE 10 – Voici un aperçu d'un des divers univers graphiques que nous allons utiliser pour représenter les différents mondes.

4.2 Le design du protagoniste

Nous avons décidé de façon quelque peu inatendue d'opter pour un avatar représentant un ninja. Au début, cette représentation du personnage devait être temporaire, mais au fur-et-à-mesure du développement du jeu et des différentes phases de test, nous nous sommes attachés à ce personnage. Cela est certes un tant soit peu en désaccord avec notre univers, mais cela nous permet de faire une référence à des jeux tels que Ninja Gaiden ou Megaman, dont les différentes animations du personnage sont insprirées.

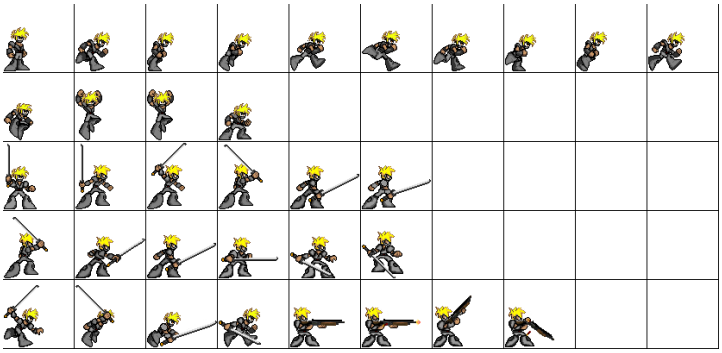

FIGURE 11 – L'actuelle planche de sprites du personnage principal, qui évoluera sûrement au cours du développement.

Voulant rajouter différentes mécaniques de jeu cette représentation permet, avec de nouvelles animations, mouvements et autres techniques qui donneront au joueur un ensemble d'outils lui donnant la possibilité de parcourir aisément les différents niveaux du jeu tout en rendant cela le plus fluide possible.

4.3 Un ennemi coriace : le Dalek

Le premier ennemi emblématique du jeu est lui tiré d'une license connue de tous : « Doctor Who », l'équipe étant fan de cette série, nous étions ravis à l'idée d'implémenter un ennemi tel que celui-là.

Un des problèmes de cet adversaire est de le rendre vivant. En effet il aurait été dommage de n'avoir que quelques cases d'animation, c'est pour cela qu'il a fallu lui rajouter un élément visuel qui permettra au joueur de savoir si le Dalek l'a repéré ou non. Cet élément étant la lumière rouge sur la tête du robot, qui scintillera de manière plus importante quand le joueur sera à proximité.

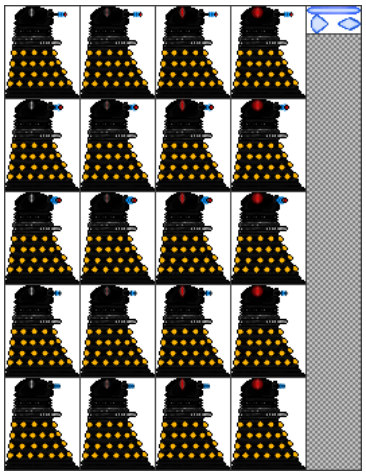

FIGURE 12 – La planche permettant d'animer l'ennemi dalek en jeu.

4.4 Petit mais efficace : le bot FlashPlayer

Dans l'optique d'un ennemi proche de l'archétype du joueur, l'intégration d'un ninja, cette fois-ci hostile, pourrait, en étant relié au scénario, être prometteur. Il semblait logique que ce nouvel ennemi soit aux couleurs du module FlashPlayer, qui souvent pose quelques soucis, comme le fera ce petit ninja.

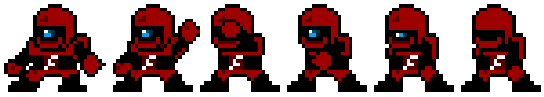

FIGURE 13 – Quelques animations que nous avons pour cet ennemi.

5 Les Ennemis

The Bytecoder est un jeu de plateforme dans lequel le héros est confronté à de nombreux obstacles. L'environnement hostile dans lequel il s'aventure est peuplé d'ennemis n'ayant qu'un seul but : le ralentir le plus possible et, si possible, le tuer.

Mais ces ennemis ne doivent pas être ennuyants, ni répétitifs... Sous peine de rendre le jeu lent ou monotone. Ainsi nous avons décidé d'implémenter un système structuré et hiérarchisé, afin de générer et de « faire vivre »cette Inteligence Artificielle que représente les ennemis. Voici son fonctionnement détaillé.

5.1 Le cœur de l'intelligence artificielle : l'Enemy Manager

Tout comme le système de variables centralisées qui peut changer au cours de la partie, l'EnemyManager permet une gestion dynamique et centralisée de l'IA. Celui-ci est représenté par une classe, et une unique instance de cette dernière est créée au cours du jeu, suffisant à la création, l'évolution et l'affichage de tous les ennemis.

Dans celui-ci, une liste dynamique représente les ennemis, qui sont eux-mêmes des objets que nous détaillerons plus tard. Ainsi, les méthodes de mis à jour des positions et de dessin des ennemis sont appelées, grâce à une boucle, dans des méthodes aux noms explicits. Il suffit donc d'appeler ces fonctions hors de la classe pour mettre à jour et dessiner tous les ennemis.

Chaque type d'IA a ses caractéristiques propres. Avant de pouvoir en ajouter grâce à la méthodes *spawn()*, on doit instancier un modèle qui viendra s'ajouter à un dictionnaire de EnemyModel. Ce dernier est une classe indiquant les caractéristiques d'un type d'ennemi. Voyons en détail le fonctionnement de celui-ci.

5.2 Un modèle pour chacun : l'EnemyModel

Tous les ennemis ne se valent pas. Certains sont plus puissants que d'autres, certains peuvent bouger, d'autres non. Certains sont spécialisés dans les attaques à distances, d'autres dans les attaques de corps-à-corps.

Afin de pouvoir décider les caractéristiques des différents types d'ennemis, nous avons choisis une méthode à la fois simple et flexible : les fichiers de modèles. Dans ceux-ci, sont listés les différentes variables et constantes qui définiront l'EnemyModel, ainsi que emplacements des sprites à charger à l'instanciation de la classe.

De ce fait, pour créer le modèle, il faut d'abord allouer une instance puis aussitôt charger un méthode *load()*, qui retourne un booléen pour savoir si tout s'est bien passé ou non. Cette méthode analyse le fichier .enemy dont le nom est passé en paramètre, afin d'ajouter des variables de type « nomdumodele.variable »à la table de hachage du Varmanager. Les sprites de l'ennemi et des armes qui le composent sont également chargés dans cette méthode.

Une fois que *EnemyModel.Load()* a retourné la valeur vrai, le modèle est utilisable dans l'EnemyManager avec EnemyManager.Spawn(string model, int ix, int y) pour ajouter un nouvel ennemi à la liste. Celui-ci sera créé dans la classe Enemy qui se chargera de l'évolution de l'ennemi jusqu'à ce que le joueur le tue.

Un extrait du fichier de configuration ninja.enemy :

```
staticenemy 0 // Si l'ennemi doit être statique ou non
melee 1 // Si l'ennemi peut taper au corps-à-corps
meleedistance 70 // La distance minimale pour frapper au corps-à-corps
meleewait 20 // Le temps d'attente entre les attaques au corps-à-corps
pv 100 // Ses points de vie
deathwaittime 0.0002
attackDistance 350 // La distance minimale pour attaquer à distance
weaponspeed 5 // La vitesse de l'arme
rotation 30 // La rotation de l'arme
throwFrequence 80 // La frequence de jet d'arme
...
```

5.3 L'Algorithme principal : l'Enemy

C'est dans cette classe que les propriétés et les méthodes de l'ennemi sont déclarées. Le constructeur prend en paramètre le Player (le joueur), le VarManager (le gestionnaire de variables), le LevelManager (le niveau), sa position au moment de sa création, et bien sûr l'EnemyModel, qui lui sert de « modèle ». Les textures de l'ennemi et de l'éventuelle arme à distance sont aussi chargées à partir de l'EnemyModel.

C'est aussi dans le constructeur que les booléens indiquant sa nature sont initialisés en fonction des variables de « type <EnemyModel.name>.variable. »Ces booléens sont les suivants :

- *staticenemy*, qui indique si l'ennemi peut se déplacer dans le niveau ou non
- *melee*, qui indique si il peut attaquer au corps-à-corps ou non
- *distanceattack*, qui indique si le l'ennemi peut attaquer à distance ou non
- *canjump*, qui indique si l'ennemi peut sauter afin d'éviter les obstacles

Ces variables permettront de déterminer les actions que les ennemis doivent faire, grâce à des structures de contrôle conditionnelles.

La méthode la plus importante de cette classe est la méthode *Enemy.Next()*, qui appelle d'autres méthodes internes à la classe, permettant à l'ennemi de se déplacer, tel que *Enemy.MoveX()* qui détermine le déplacement horizontal de l'ennemi et *Enemy.MeleeAttack()* qui permet de calculer la force et la fréquence des attaques de corps à corps.

Les armes à distances sont des objets stockés dans une liste déclarée dans la classe Enemy. Ceux-ci sont créés dans la méthode Enemy.Next().

5.4 Une arme à part entière : FarWeapon

Le constructeur de l'arme prend en paramètre sa vitesse, sa position de départ, le Player, le LevelManager, le VarManager, sa texture ainsi qu'une variable indiquant sa vitesse de rotation.

La direction du l'arme est calculée grâce un angle, lui même calculé à partir de l'arc tangente de la différence en y entre l'ennemi et le Player sur celle en x. On applique donc à cet angle le cosinus pour trouver le vecteur vitesse en x et le sinus pour trouver le vecteur vitesse en y. Cette arme continuera sa trajectoire jusqu'à toucher l'ennemi ou bien un mur.

6 Les Effets Graphiques

6.1 Les Particules

Un jeu en deux dimensions n'implique pas forcément un environnement statique et sans animations. Pour le ByteCoder, nous voulons rendre l'univers le plus vivant possible. Et pour ce faire, nous avons décidé d'implémenter un moteur à particules qui sert de base à la majorité des effets simples du jeu.

Ce moteur à particules se résume en trois classes, reprenant le principe de Managers guidant la conception du jeu. Voici une explication détaillée de leurs rôles et de leur fonctionnement.

6.1.1 « Particle »

Dans notre jeu, les effets, comme les étincelles, sont un ensemble de particules réagissant à certaines règles, qui, unis, vont créer quelque chose de reconnaissable par l'œil du joueur. Cet objet, le *Particle*, prend plusieurs arguments dans son constructeur : sa position, sa couleur, sa vitesse, une variable représentant la gravité et une autre sa durée de vie.

A chaque mise à jour, la particule se déplace en x d'une distance égale au cosinus de l'angle pris en paramètre et en y d'une distance variable. Au moment de l'instanciation de la particule, cette distance vaut le sinus de l'angle, mais à chaque rafraichissement on ajoute à cette distance la variable de gravité. Ainsi pour que la particule s'envole il suffit de lui donner une gravité négative et pour qu'elle tombe une gravité positive. De plus à chaque rafraîchissement, la transparence alpha de la particule diminue.

L'alpha diminue de (1 div (la durée donnée en paramètre)). Lorsqu'il est arrivé à 0, la méthode de rafraîchissement de la particule renvoie un booléen faux : la particule peut être détruite.

6.1.2 « ParticleModel » : le socle de l'effet

Les particules composant l'effet doivent avoir des règles bien délimités, pour éviter de se retrouver un effet avec des particules de toutes les couleurs et partant dans tous les sens. C'est le rôle de cette classe qui va gérer les différentes variables de chaque effet.

Dans cette classe, toutes les variables sont chargées à partir d'un fichier de configuration : l'angle minimal et maximal, la gravité, la vitesse minimale et maximale des particules, le nombre minimal et maximal de particules qui peuvent être créés dans l'effet ainsi que leurs durées.

Après la création de la classe, il est possible à tout moment de changer ces propriétés grâce à une fonction qui prend une ligne de commande en paramètre. Ainsi, un effet pourra changer de nature pendant son exécution.

6.1.3 L'effet : « ParticleManager »

Un facteur aléatoire permet d'avoir un effet réaliste, en faisant qu'aucune (ou presque) particule n'ai les mêmes caractéristiques : la direction, la vitesse, parfois même la couleur.

Deux méthodes importantes permettent d'appeler les méthodes des particules de la liste. En premier lieu, Next() qui appelle la méthode de rafraîchissement de la particule, tout en testant le booléen renvoyé par celle-ci. Rappelons-le, celui-ci indique si la particule a un alpha égal à 0 ou non. Si oui, Next() va supprimer la particule de la liste.

La deuxième méthode importante est Draw(), qui va appeler les méthodes de dessin de chaque particule.

6.2 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.

6.2.1 « 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 jours des positions et de la transparence, qui diminue. Elle renvoie un booléen indiquant si son alpha est nul ou non. La deuxième dessine la particule.

6.2.2 « LittleFire »

Cette classe représente les flammèches qui composeront l'effet de feu. Le constructeur prend en paramètre :

- la plage de l'abscisse dans la quel 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 don plus naturelle, des particules.

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

6.2.3 « FireManager »

Cette classe est le pilier central de l'effet de feu. Elle gère toutes les flammèches qui le composeront. 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 celle de dessin, qui appelle celles des flammèches.

6.3 « SpawnEffect »

L'effet « SpawnEffect »comprend deux classes : « SpawnEffectManager »et « SpawnParticle ». La classe « SpawnParticle »définit les attributs pour une particule et le « SpawnEffectManager »permet le déplacement de ces particules en fonctions de leur vitesse sur l'axe des x et des y. L'objectif étant de réunir chacune de ces particules pour les rassembler au point de création du personnage. Cela donne donc un effet d'assemblage des particules pour recréer l'image du personnage. La texture des particules se base sur l'image 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 particule et ce SpawnEffectManager en appelant la méthode « Dispose »de ce dernier, ce qui entraînera la destruction des particules.

6.4 L'Animation de la Mort

On ne peut pas rire de tout, et encore moins de la mort. C'est un sujet sérieux qui demande du respect. Lorsque le personnage du joueur ou même un ennemi meurt, il ne peut pas juste disparaître. Il lui faut une mort digne.

Univers numérique oblige, cet effet doit rester sobre (donc pas de bains de sang . . .). Nous avons donc décidé d'implémenter un effet de morcèlement de l'image du personnage qui meurt. Une décomposition en particules qui se désintègrent doucement, comme une animation inverse de celle à l'apparition des personnages.

Cet effet est composé de deux classes : une qui r eprésente le fragment d'image, et l'autre qui gère tous ces fragments.

6.4.1 « DeathParticle »

Cette classe prend en paramètre l'angle qui déterminera sa vitesse en x et en y, une texture, la gravité, une variable représentant sa durée de vie et une autre le temps d'attente avant que la particule ne parte. Elle est composée de deux méthodes.

La première, de mise à jour des positions, vérifie si la variable de temps d'attente vaut 0. Si non, elle diminue cette variable et retourne un octet signifiant que la particule est en attente. Si oui, elle diminue la variable de durée de vie et applique la translation et la gravité. La deuxième méthode, est une méthode de dessin classique.

6.4.2 « DeathManager »

Le constructeur de cette classe prend en paramètre :

— la texture finale du personnage mourant
— sa position
— sa taille
— sa durée
— l'origine, en coordonnées cartésiennes, de l'impact fatal
— la gravité (si les particules doivent s'envoler ou tomber)

Un TextureBuilder, instancié dans le constructeur, permet de découper la texture en sous-textures qui seront passées en paramètres aux particules.

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.

FIGURE 14 – Trois différentes phases de la décomposition d'un Dalek

7 L'Éditeur de Niveau

7.1 Son principe

Voulant créer des niveaux de plus en plus alambiqués pour essayer les dernières mécaniques implémentées, il était nécessaire de prendre de plus en plus de temps pour créer ces niveaux, à l'aide d'un bloc-note pour les élaborer et les sauvegarder. Au bout d'un moment la création d'un éditeur de niveau s'est présentée comme une obligation. En effet la syntaxe étant de plus en plus lourde au fur-et-à-mesure des ajouts, tels que les objets scriptés ou les événments, le développement d'un outils permettant d'automatiser ces actions. De plus cela rend possible d'élaborer des niveaux plus rapidement tout en ayant un visuel plus proche de celui du jeu.

7.2 L'Interface

Il était nécessaire de créer un outil simple à utiliser et complet. Pour cela le choix de s'orienter vers le type de fenêtres utilitaire que sont les « Windows Form »fut un exellent moyen d'y parvenir, en effet, en couplant la puissance de cette interface à un affichage XNA directement incorporé dans la fenêtre il fut possible d'obtenir un éditeur à la hauteur de nos attentes.

L'interface est découpée en plusieurs sections ayant chacune une utilité différente comme visible ci-dessus.

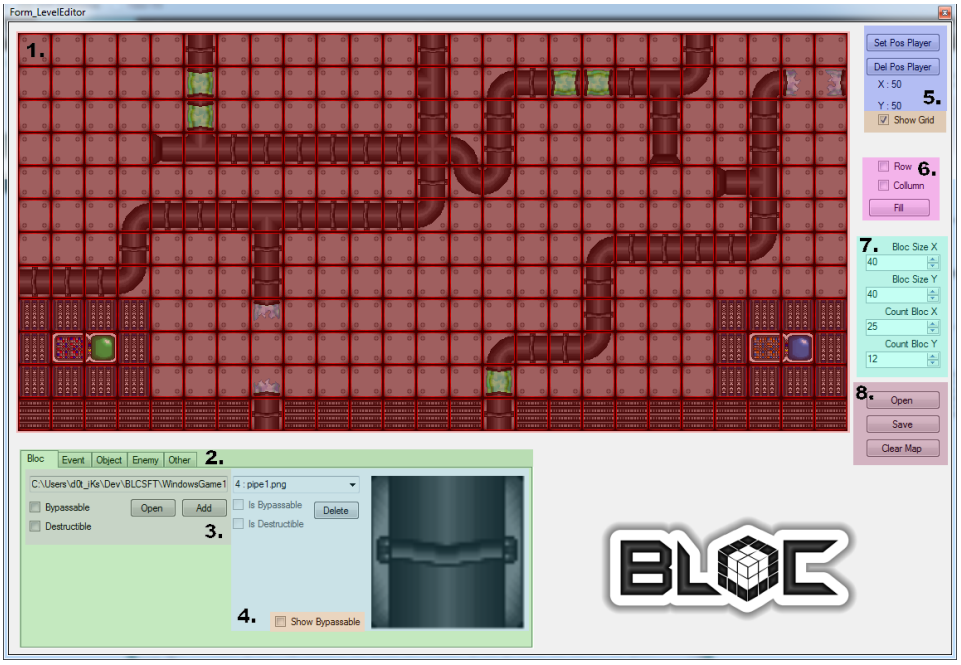

FIGURE 15 – Interface de l'éditeur de niveau.

1. L'APERCU Cette partie permet d'obtenir un visuel le plus proche possible du jeu, de plus il a fallu rapidement rajouter un système de défilement à l'aide d'une caméra de façon similaire au jeu lui-même, car la création de niveaux de plus en plus grands devenait un problème.

2. LES ONGLETS Cette partie est découpée en 5 onglets. Chacun de ces onglets ayant pour but de rajouter des éléments bien distincts dans le niveaux.

Le premier est la base de l'éditeur permet de rajouter des blocs construisant le niveau.

Le deuxième permet quant à lui de rajouter des événements déclencheurs comme expliqué plus haut dans le rapport.

Le troisième permet lui de rajouter des objets scriptés comme éxpliqué plus haut dans le rapport également.

Le quatrième donne la possibilité de rajouter des ennemis à partir d'un modèle pré-existant.

Le dernier et cinquième onglet permet de rajouter un fond d'écran au niveau, ceci étant utile si certaines zones ne contiennent pas d'éléments fixes (ce qui n'est pas le cas de ce niveau), ainsi que de rajouter une musique de fond qui se lancera dès le début du niveau.

3. L'AJOUT Cette troisième section existe de façon similaire dans les quatres premiers onglets en effet elle permet de sélectionner, selon l'onglet, la texture des blocs, des scripts, des modèles d'objets ou des modèles d'ennemis.

4. LA SELECTION Cette partie permet, après avoir rajouté les différents éléments nécessaires à la conception d'un niveau, de les sélectionner, d'obtenir un aperçu et ensuite de les placer dans le niveau. Concernant les événements, il est aussi possible de modifer les scripts qui leurs sont liés. Il est aussi possible de supprimer les différents éléments si au

final ils ne sont pas utilisés.

5. **LE DÉPART** Ces deux boutons permettent en effet de choisir où le personnage va être généré au début du niveau, en sachant que le joueur sera tout de même soumis à la gravité si l'on choisit de le faire apparaître dans les airs, il est aussi possible de supprimer cette position qui, par défaut, est aux coordonnées 10 ;10.
6. **REMPLISSAGE** Cette section de l'éditeur permet la conception de grands niveaux de façon plus aisée, en effet cette dernière permet de remplir des zones entières de la carte de jeu, des lignes, des colonnes ou même tout le niveau avec le dernier élément sélectionné.
7. **PROPRIÉTÉ** Cette zone munie de quatres indicateurs numériques permet d'augmenter la taille du niveau ou de la diminuer, pour cela il est mis a diposition un moyen d'agrandir ou de rétrécir la taille des blocs, ou alors d'augmenter ou de diminuer leur nombre (il est impossible d'atteindre des valeur en dessous de 1 pour tous les indicateurs).
8. **EDITION** Ces trois derniers boutons permettent d'importer, d'exporter ou de réinitialiser un niveau.

On distingue aussi deux zones en orange sur l'interface, celles-ci permettent de mettre en évidence dans le premier cas les blocs que le joueur pourra traverser ou bien les différents évenement, objets ou ennemis selon l'onglet et dans le second cas, d'afficher une grille permettant de bien placer les blocs lors de la création d'un niveau.

7.3 Le Fonctionnement

7.3.1 Les fichiers « .lvl »

Pour sauvegarder les différents niveaux nous avons eu recours à un fichier avec une extention spécifique, les « .lvl ». Ces fichers permettent de garder de façon externe au jeu presque toutes les informations relatives au niveau. Sur le schéma ci-dessous on peut distiguer plusieurs zones, chacune ayant une importance capitale :

FIGURE 16 – Fichier « .lvl »ouvert dans un bloc note.

COMMANDES (bleu) : Les commandes font appels aux différentes commandes de la console déjà expliquées plus haut (se référer à la gestion des Variables).

IDENTIFIANTs (orange) : Ces identifiants permettent,après les avoir enregistrés, de les utiliser en utilisant ces identifiants sans devoir rapeler toutes leur propriétés.

RÉFÉRENCES (vert) : Les références sont la partie principale des différents éléments enregistrés, en effet, pour les blocs, nous allons charger une texture et pour les ennemis un modèle.

APPEL DE VARIABLES (violet) : Comme expliqué plus haut l'utilisation des identifiants permet de façon plus rapide de modifier le niveau, par ailleurs la carte d'un niveau peu contenir 78 identifiants différents, ce qui laisse la possibitlié de créer des niveaux impressionnants et variés.

MODIFICATION (rouge) : La modification ou l'initialisation des différentes propriétés de chaque élément tel que : la possibilité de le traverser pour un bloc, la position pour un ennemi ou bien le nombre et la taille des bloc de façon générale dans le niveau.

7.3.2 Sauvegarde des différents blocs

Pour stocker les différents blocs dans l'editeur de niveau nous devions sauvegarder en mémoire tous ces derniers. Le problème c'est que nous voulions utiliser le moins d'emplacement mémoire possible. Pour une technique similaire à celle de l'utilisation des identifiants dans les fichiers est utilisée.

En effet à chaque fois que l'utilisateur rajoutera un bloc pour construire son niveau, l'éditeur le sauvegardera de manière unique. Pour cela nous avons utilisé un dictionnaire de blocs nous permettant d'avoir un référencement et un accès simplifié aux différents éléments dont l'utilisateur a besoin.

```
private Dictionary<char, GraphicBloc> bloc_list;
```

L'utilité du dictionnaire étant expliquée précédemment, nous pouvons juste ajouter que ce dernier répond parfaitement à nos attentes, autant en termes de rapidité que de simplicité.

7.3.3 Sauvegarde d'une carte de références

Maintenant que les différents blocs sont accessibles via un identifiant qui est en mémoire un *char*, caractère, pour sauvegarder les différentes positions des blocs, nous pouvons utiliser une Liste à deux dimentions qui représentera la grille que nous pouvons voir dans le visuel de l'éditeur de niveau.

L'utilisation d'une liste à deux dimensions est justifiée par la nécessité de pouvoir à tout moment agrandir ou diminuer la taille de la carte durant l'édition. Ne voulant pas détruire et reconstruire un tableau à chaque changement de taille, l'utilisation d'une structure dynamique s'est imposée d'elle-même.

```
List<List<char>> map;
```

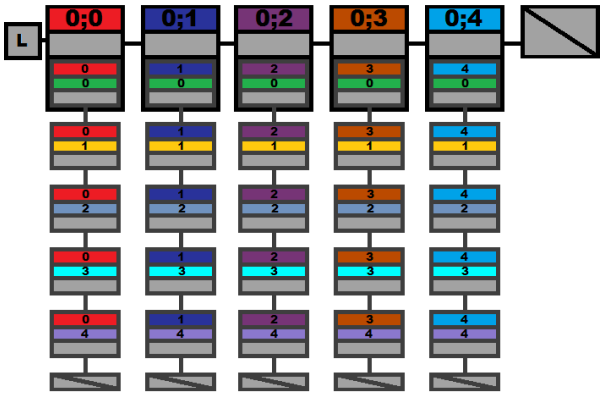

FIGURE 17 – Représentation de la liste à deux dimension utiliser pour la carte.

8 Les Menus

Les différents menus sont contrôlés depuis le menu principal comme montré sur le diagramme suivant :

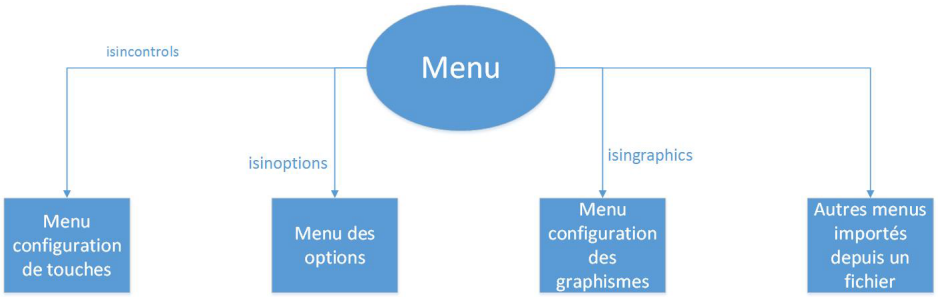

FIGURE 18 – Gestion des menus.

8.1 Edition d'un menu depuis un fichier

Certains menus sont créés depuis un système de fichier portant l'extension « .menu ». Le principe étant de créer des menus dynamiques via cette configuration. Le fichier est donc lu ligne par ligne et le jeu interprète chaque ligne puis met à jour les composantes de chaque bouton.

FIGURE 19 – Voici comment se présente un menu fait à partir d'un fichier

```
set buttoncount 4
background black.png
bindkey escape "multiple (inmenu 0) (bind escape (inmenu 1) 1 1)"
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 0 52 158
setcolor 1.color 0 52 158
setcolor 2.color 0 52 158
setcolor 3.color 0 52 158
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 20 – Cette image montre le résultat de l'interprétation du fichier « pause.menu »

— Avec la première ligne, on définit le nombre de boutons utilisés dans le menu.
— La seconde ligne sert à définir le fond à charger pour ce menu.
— La troisième ligne permet de définir de manière globale une touche qui fera une certaine action (ici quitter le menu grâce à la touche « échappement »).
— Les huit lignes qui suivent servent à fixer la position des boutons de façon relative, c'est-à-dire en pourcentage par rapport à la résolution du jeu.
— Les quatre lignes qui suivent définissent le texte à afficher lorsque le bouton sera chargé.
— La couleur du bouton est ensuite définie, suivie de la couleur qui sera dessinée lorsque le bouton sera sélectionné.
— Enfin, une commande ou un script est attribué à chacun des boutons, ainsi, lorsque le joueur appuiera sur entrée, cela exécutera le script ou la commande.

Le fichier sera donc parcouru et pour chaque bouton qui correspond, son texte lui sera attribué, sa couleur, sa commande ainsi que sa position. Ensuite, si la ligne correspond à une directive d'affichage comme le fond d'écran, la fonction vérifiera si l'image existe.

8.2 Configuration des graphismes

Pour la gestion des résolutions,une fonction récupère les résolutions disponibles pour l'utilisateur.Une fonction récupère les résolutions disponibles pour l'utilisateur et les renvoie sous la forme d'un tableau. Le menu de configuration des résolutions va donc récupérer ces valeurs pour pouvoir ensuite modifier la résolution du jeu et adapter la position des boutons dans les menus en cours. D'autres fonctionnalistés ont été ajoutés comme la possibilité d'activer et de désactiver le plein-écran, l'anti-crénelage mais aussi de changer la qualité du feu.

8.3 Configuration des touches

Grâce à un menu de configuration des touches, le joueur a la possibilité de modifier les touches qu'il veut utiliser, en sachant que ces paramètres sont enregistrés pour les prochaines sessions, ainsi qu'à chaque modification.

8.4 Résumé de la classe bouton

La classe bouton est la base de tous nos menus. En effet, chacun des menus a besoin de créer des boutons puis créera une liste de ces boutons. Ils seront ensuite affichés grâce à la méthode « Draw ».Ainsi, un bouton a plusieurs attributs résumé dans ce diagramme :

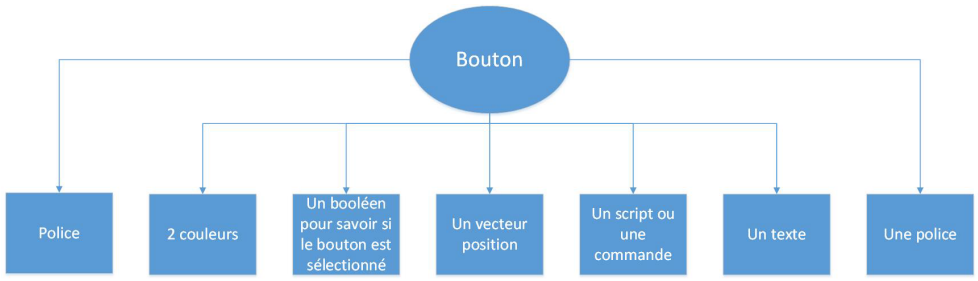

FIGURE 21 – Attributs d'un bouton

8.5 Fond Dynamique

Dans le menu principal, nous avons intégré un fond d'écran dynamique. En effet, à partir d'un FontManager, qui charge une police depuis une image. Le principe de ce gestionnaire est de détecter les bords rouges de l'image qui définissent un caractère.

FIGURE 22 – Un exemple de caractères pris en charge par le FontManager

Le FontManager va donc récupérer chacune des lettres qui se situent entre les cases rouges. Cette image contient chacun des caractères dont le menu aura besoin pour le fond dynamique. Il est donc possible, à partir de ce gestionnaire, de récupérer une image pour chaque caractère.Ces caractères seront ensuite repris par l'effet qui gère ce fond dynamique. On a donc le résultat suivant :

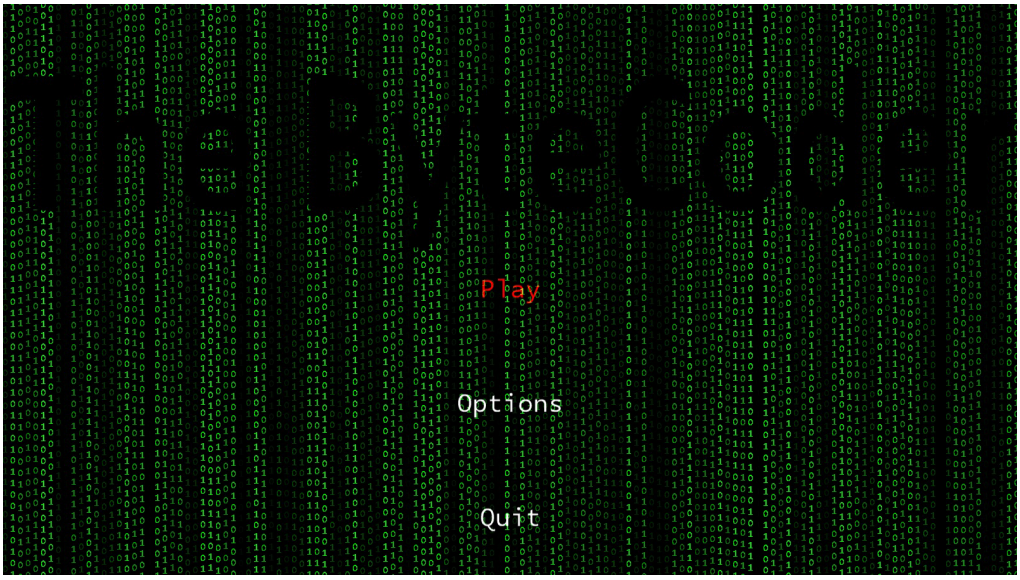

FIGURE 23 – Fond dynamique obtenu

Chacune des colonnes est définie par une vitesse de déplacement aléatoire ainsi qu'une transparence aléatoire

9 Expérience personnelle

Pierre « BLUESCREEN »BOULAY

Après plusieurs mois de travail en équipe sur ce projet, je suis très satisfait de constater que nous avons réussi à obtenir un résultat qui répond à nos attentes. Même s'il reste encore beaucoup à faire, nous avons pu constater la puissance du travail en équipe. Les idées et inspirations de chacun donnent chaque jour une nouvelle dimension au jeu. Notre motivation nous permet de passer beaucoup de temps sur le projet, et c'est pour moi un véritable bonheur de voir le jeu prendre forme, brique par brique. Il est encore trop tôt pour dire comment sera le résultat final, mais je sais que nous pouvons réussir à aller jusqu'au bout de nos rêves. Et même s'il nous arrive de ne pas être du même avis, nous avons jusqu'ici toujours su résoudre les problèmes et prendre les bonnes décisions.

Serge « Kh4L »Panev

Après quelques mois à travailler sur notre jeu, j'ai beaucoup appris en programmation et en travail de groupe. Les différents points de vue des membres du groupe nous permettent d'avoir une approche complète des problèmes techniques qui ont pu appaître.

Ce projet est pour moi une première, dans le sens où je n'avais jamais participé à un travail si important, dans lequel la rigueur et la persévérance sont de mise. J'ai dû passer de longs moments à réfléchir sur une optimisation de mes algorithmes afin de rendre le jeu le plus optimal possible. Ainsi, j'ai pu mobiliser mes connaissances en algorithmique, en mathématique et physique.

Finalement, les objectifs que nous nous étions fixé pour cette première soutenance ont été atteints. Et même si ce n'est que le début, notre volonté de faire le meilleur jeu nous motive toujours plus.

Paul-Henry « Dr_c0w »PERRISSEL

S'occuper des menus a été une expérience enrichissante. En effet, il a fallu lire les fichiers, s'assurer que toutes les variables que l'on veut récupérer sont bien présentes dans le fichier et gérer les cas où ces valeurs n'existent pas. Cela m'a permis de découvrir des fonctionnalités du C# que je ne connaissais pas, où du moins qui n'avaient pas encore été abordées en TP, tout en s'assurant que quoiqu'il arrive le résultat sera stable.

Pour cette première soutenance, il a également fallu gérer des menus fait directement en C# puisque les fonctionnalités recherchées n'étaient pas compatibles avec des menus créés à partir de fichiers. C'est le cas du menu de configuration des touches ainsi que le menu de configuration des graphismes. En effet, il n'est pas possible de changer une touche paramétrée pour une certaine action pour qu'elle soit assignée à une autre tâche voire qu'elle n'ai plus d'influence sur le jeu. De même pour les changements graphiques, il n'était pas possible de créer un menu à partir d'un fichier qui permette de changer la résolution. La seule chose en commun à ces deux méthodes est la classe bouton qui est et restera très utile tout au long du projet.

Alessandro « d0t_iKs »PISU

Depuis la création du groupe j'ai toujours pensé que nous avions les atouts pour concevoir un jeu complet et plaisant à jouer. Aujourd'hui à l'heure de la première soutenance je peux affirmer que nous sommes en bonne voie. Certe nous avons nos différents mais nous avons toujours réussi à trouver une solution propre et qui convenne à tous.

Concernant mon experience au cours de ces premiers mois je trouve que mon niveau tant en terme de programmaion que de travail en équipe à nettement crû. J'ai aussi de plus en plus d'idées pour l'avenir du projet tant en termes de mécaniques de jeu que de graphismes pour les différents mondes.