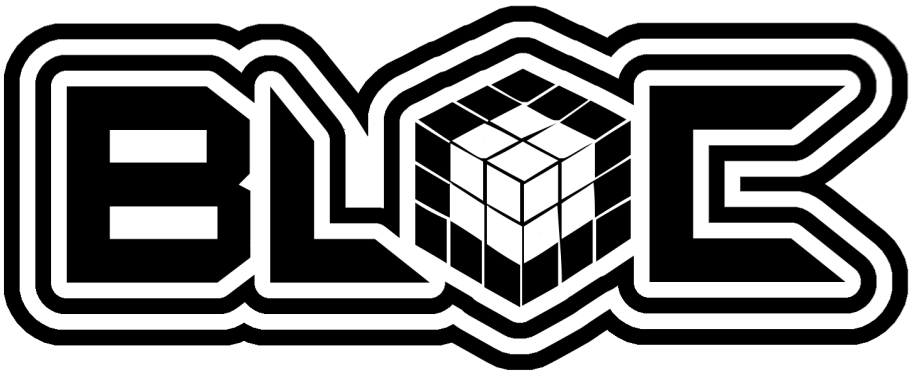


Rapport de 2e Soutenance « The ByteCoder »

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 Introduction

Après plusieurs mois de développement, notre projet commence à prendre forme et le résultat devient de plus en plus clair. Les composants deviennent de plus en plus élaborés, les mécaniques de jeu de plus en plus travaillées, donnant au fur et à mesure, un aspect plus soigné à notre jeu. C'est pour nous un plaisir de voir un projet évoluer et suivre nos idées. Même si son élaboration n'est pas encore terminée, le résultat que nous avons actuellement est un bon aperçu de la version finale. Ce document présente le nouveau contenu ajouté au ByteCoder ainsi que les méthodes utilisées pour aboutir à un tel résultat. Ainsi, nous allons présenter les différents moyens mis en œuvre afin de réaliser tout ce nous avions prévu, des fondements du jeu à l'expérience utilisateur, en passant par le scénario et les différents médias utilisés dans ce jeu.

2 La structure de rendu

2.1 Rendu graphique des différent 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 dans quel ordre les objets doivent être affichés.

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 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 la quantité de dégâts que l'objet va recevoir.

2.2 Ajout et Suppression des objets

Pour optimiser l'ajout et la suppression d'objet, la liste des objets est stockée dans une liste doublement chaînée. Le principe d'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 palier 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 chainée pour être mis à jour et dessinés, et, d'autre part stockés 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 doublement chaînée 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 l'élément du tableau par l'élément suivant dans la liste chainée, 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 – Fonctionement de la routine AddObject()

3 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). Si le joueur a chargé assez longtemps, l'arme tire un projectile lorsque la touche de chargement est relâchée. L'image ci-dessous représente cette animation de chargement

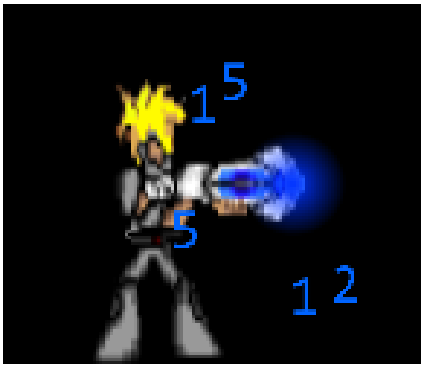
centering 

FIGURE 2 – 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 nombre 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 VitesseR) 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 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 3 – Trainée de chiffres

4 Le multijoueurs

La possibilité de jouer à plusieurs ajoute une nouvelle dimension aux mécaniques de jeu de ce projet. Mais « multijoueurs »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'objet 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.

4.1 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 lui transmettre les commandes de l'utilisateur, si le joueur n'est pas fini, la valeur de son index dans le tableau est nulle.

Les entrées utilisateurs sont gérées par des commandes. Comme expliquées dans le précédent rapport, 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 sur 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.

4.2 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 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 si 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 connaitre 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 : $\sqrt{(x1+x2)^2+(y1+y2)^2}$.

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 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 être représentée par trois cas distincts :

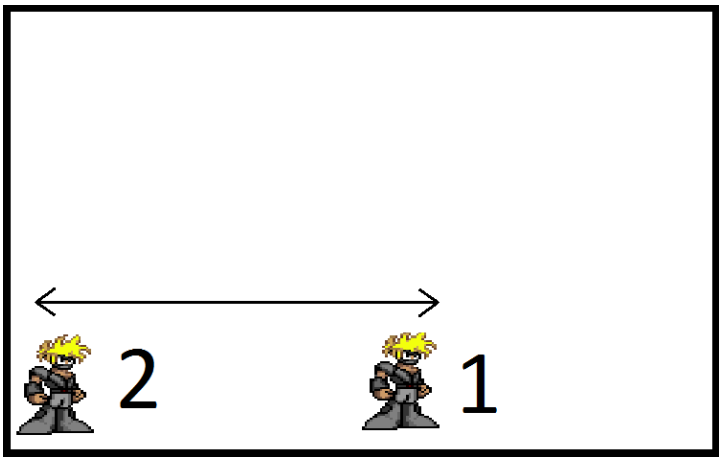


FIGURE 4 – Premier cas

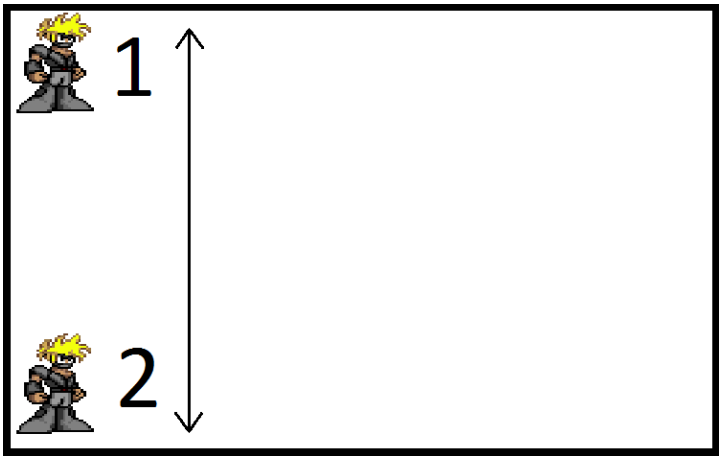


FIGURE 5 – Deuxième cas

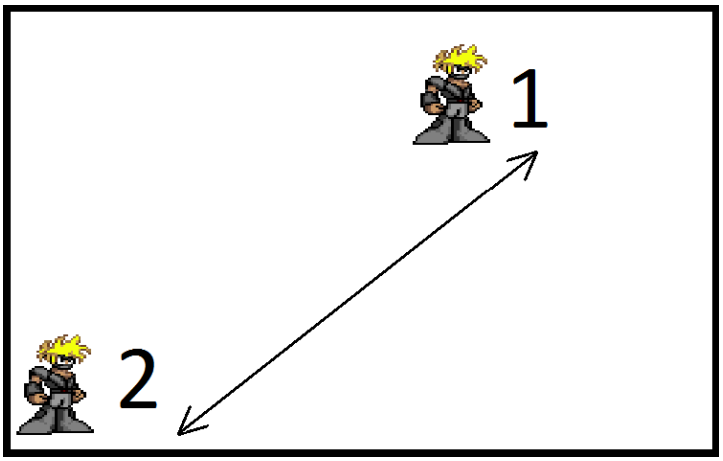


FIGURE 6 – Troisième cas

Ces trois schémas montrent qu'à distances égales, 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.

5 Les ennemis

5.1 Une arme à distance plus intelligente

Jusqu'ici, les armes à distance des ennemis étaient lancées par ceux-ci en direction du joueur, sans vérifier ce qui sépare les deux protagonistes.

Nous avons donc implémenté une méthode dans la classe de l'ennemi qui permet d'analyser les objets qui séparent l'ennemi de sa proie. Cette fonction, à l'aide de l'angle calculé qui permet de savoir dans quelle direction se situe l'ennemi, a pour structure principale une boucle qui vérifie chaque position entre les deux entités. Si l'une de ces positions est occupée par un objet appartenant au niveau, cette méthode renverra un booléen « faux »empêchant ainsi l'ennemi de tirer, et par conséquent rendant le comportement de celui-ci plus intelligent.

5.2 Plus de joueurs : un choix pour l'ennemi

Avant l'intégration du mode multijoueur dans notre jeu, l'ennemi n'avait qu'un seul but : tuer Bytecoder, le personnage incarné par le joueur. Toutes ses actions étaient focalisées sur ce dernier.

Mais avec l'apparition de plusieurs joueurs, nous avons dû repenser le comportement de l'ennemi. Ainsi nous avons créé un système de gestion de "joueur cible". Celui-ci est déterminé en fonction de la distance qui sépare chaque joueur de l'ennemi. Ce dernier prend comme cible principale le joueur principale.

5.3 Des interférences électromagnétiques : Interference Effect

Lorsque le joueur attaque un ennemi, peu importe de quelle manière, il lui inflige des dégâts de quantité variable. De cette façon, les points de vies de l'ennemi diminuent, et quand ceux-ci sont inférieurs ou égaux à 0, il meurt et ne représente plus un danger pour le joueur.

Ainsi le joueur doit pouvoir savoir si son attaque a atteint l'ennemi, et s'il lui a infligé des dégâts. Dans beaucoup de jeux, ce retour est représenté par changement de couleur de l'ennemi pendant quelques secondes ou bien, dans les jeux plus évolués, par un jet de sang jaillissant de l'ennemi.

Or l'univers de notre jeu fait que les ennemis sont numériques et comparables à des hologrammes produits par la machine. De ce fait, nous avons choisi de représenter les dégâts subits par l'ennemi par une distorsion de l'image de celui-ci. Cette distorsion se doit de rappeler celle produite par une vidéo à distance ayant des interférences électromagnétiques.

Le code

Pour ce faire, nous avons créé une classe qui gère l'effet à partir de son instanciation jusqu'au temps qui lui est imparti.

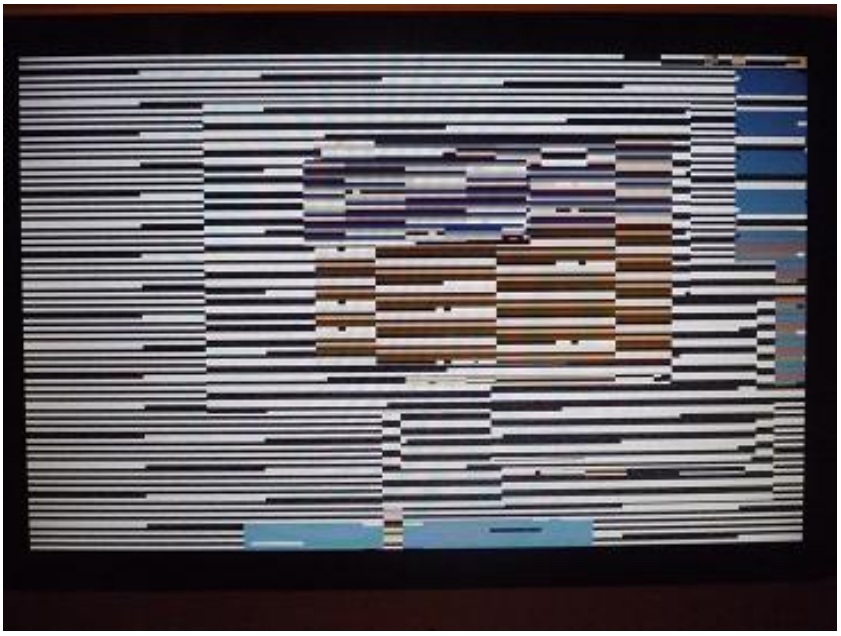


FIGURE 7 – Un écran d'ordinateur brouillé

Tout commence dans EnemyModel, la classe qui permet de garder en mémoire les propriétés et les images d'un type d'ennemi. En effet, la distorsion de l'ennemi est faite grâce à un tableau de plus petites images découpées à partir de l'image principale, qui sont en fait des bandes horizontales, découpées grâce à notre outil interne de traitement d'image : le TextureBuilder.

Ce tableau est en fait bidimensionnel, permettant d'avoir deux découpes : celle de l'ennemi tourné vers la gauche, et celle de l'ennemi tourné vers la droite.

A son instanciation, l'ennemi déclare un InterferenceEffect et un booléen qui vaudra « faux ». Ainsi, lorsque l'ennemi est frappé par le joueur, la valeur « vrai »sera affectée au booléen, qui empêchera ainsi le dessin ainsi que les mises à jour de l'ennemi. A partir de ce moment, la classe Enemy vérifiera si l'InterferenceEffect est arrivé à sa fin, en appelant une méthode de ce dernier. Si l'InterferenceEffect est fini, il est détruit et l'Enemy retrouve son comportement normal et est dessiné. Intéressons nous maintenant à la classe en elle-même et son fonctionnement interne.

La classe InterferenceEffect

Le constructeur de cette classe prend en paramètre :
— le tableau bidimensionnel de textures
— la position en x et en y de l'ennemi au moment où il a été frappé
— la hauteur et la largeur totale de l'ennemi
— la direction (droite ou gauche) au moment de l'impact
— la durée de l'effet

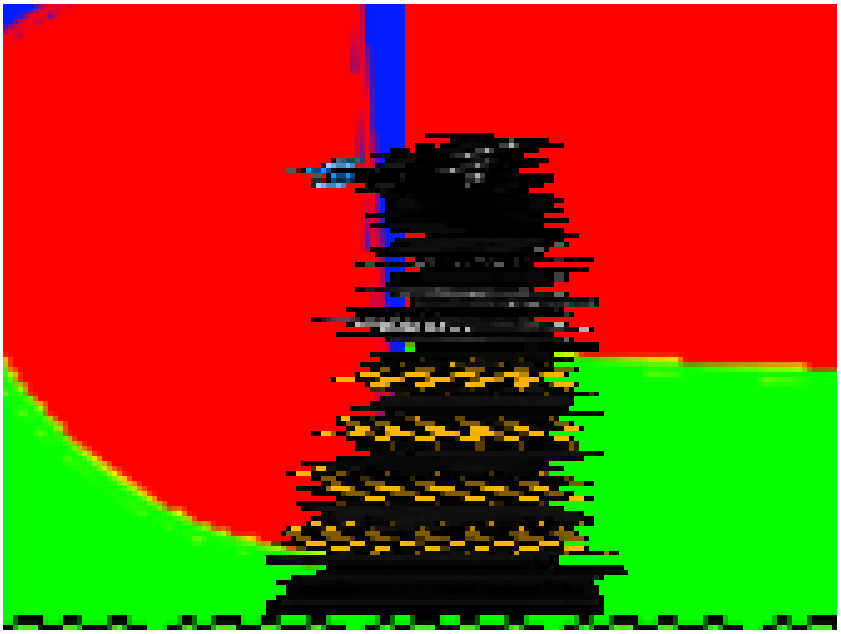


FIGURE 8 – L'effet appliqué sur un ennemi : le Dalek

A son instanciation, la classe va créer un tableau unidimensionnel (l'image droite au gauche de l'ennemi, déterminée grâce à la variable passée dans le constructeur) d'une classe annexe, appelée SubTexture, à partir du tableau d'images. Les propriété de celui-ci sont une texture, un booléen représentant le sens de la translation horizontale, la variation en abscisse par rapport à l'abscisse original et la variation en abscisse par rapport à l'abscisse original maximal qu'il peut avoir. Ce sont ces deux dernières variables qui vont permettre la translation horizontale des sprites et donner l'impression de brouillage de l'image.

La variation en abscisse par rapport à l'abscisse original maximale est aléatoire, ce qui permet de rendre l'effet instable et donc plus naturel.

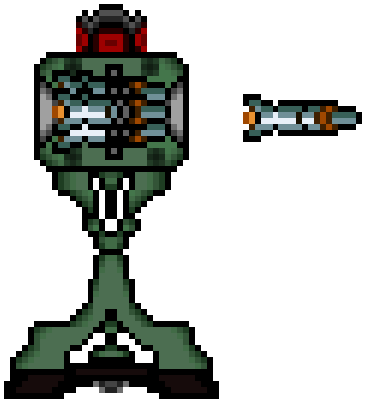
A chaque rafraîchissement du jeu, on appelle la fonction Next() de l'effet. Dans celle-ci, on décrémente la variable qui représente le temps, puis on vérifie si elle vaut 0, si oui l'effet est terminé, il peut être détruit et le fonctionnement normal de l'ennemi peut reprendre. Sinon, on incrémente ou décrémente, en fonction de la direction de la SubTexture, la variable représentant la variation en abscisse par rapport à l'abscisse original. Enfin on dessine chaque SubTexture, avec la petite variation en x.

6 Les nouveaux ennemis

6.1 L'ennemi central du premier monde : Apache

La création de cet antagoniste s'est basée sur deux éléments : le premier le plus visible, l'univers amérindien, que nous avons jugé intéressant à intégrer dans notre univers. Le second est le serveur Apache qui s'intègre avec le reste de notre univers. Au cours de son apparition, le joueur devra faire face à de multiples attaques, telles qu'une pluie de flèches qu'il devra esquiver à l'aide de déplacements ou des décors présents sur la carte.

6.2 Le lanceur



Un autre ennemi que le joueur pourra rencontrer durant son périple est le lanceur, ce lanceur mettra en jeu des ogives qui seront scriptées de telle sorte à ce qu'elles suivent le joueur sur une certaine longueur.

7 Les Menus

Nous avons choisi de modifier les menus au niveau du design. C'est pourquoi nous avons rajouté un effet appelé HyperSpace. Nous avons également décidé d'ajouter un effet aux boutons dans certains menus. Celui-ci donne une impression de zoom lorsqu'un bouton est nné et un effet de dézoom lorsque le bouton n'est plus sélectionné, le changement de couleur défini lors de la création du bouton s'applique également.

Nous avons également dû rajouter quelques menus. En effet, avec la gestion des manettes et l'arrivée du mode multijoueur, il était nécessaire de supporter le paramétrage de certaines touches pour certaines actions.

7.1 Menu de configuration de touches

Nous avons ici aussi choisi de laisser le choix à l'utilisateur des touches qui lui conviennent le mieux. Cependant il y avait une contrainte : il fallait pouvoir récupérer le « PlayerIndex », cela correspond à l'identifiant de la manette, celui-ci se situe entre un et quatre et varie en fonction de l'ordre de branchement de chacune des manettes. Pour récupérer cet identifiant, il y a deux possibilités qui se sont offertes à nous :

— Le joueur a déjà configuré des touches sur la manette, auquel cas il suffit de récupérer cet identifiant et de le comparer à la manette.
— Au moment du premier changement, on regarde l'identifiant de la manette que le joueur utilise lorsqu'il appuie sur le bouton

7.2 Menu de configuration du multijoueur

Après avoir gérer le clavier et les manettes pour plusieurs joueurs, il fallait aussi pouvoir configurer une partie en multijoueur en renseignant plusieurs paramètres : le nombre de joueurs et le type de séparation entre chacun des joueurs (Split, Zoom, AutoSplit). Une fois ces paramètres renseignés, le menu va ensuite demander la création des joueurs selon un certain fichier de configuration. Suite à cela, les joueurs peuvent choisir le terrain sur lequel ils veulent jouer grâce au LevelSelector.

7.3 Menu de sélection des niveaux

Lorsque l'on sélectionne un niveau, il faut pouvoir naviguer entre chacun d'entre eux, tout en ayant un apeçu de ce que chacun représente. C'est pourquoi le niveau précédent et le niveau suivant (quand il y en a un) sont également représenté, mais avec une représentation plus petite que le niveau sélectionné.

Il a donc fallu mettre au point deux effets. Le premier permet au joueur de choisir le niveau suivant, l'image correspondant au niveau sélectionné va donc être réduite puis déplacée vers la gauche, l'image correspondant au niveau de droite s'aggrandit et passe au centre, enfin, on affiche l'image du niveau qui suit à droite du niveau sé lectionné. Le second effet va permettre de déplacer les images dans le sens inverse et donc d'avoir l'image correspondant au niveau précédent au centre avec à droite et à gauche respectivement le niveau précédent et le niveau suivant.

8 Un menu à la vitesse de la lumière : l'HyperspaceEffect

L'hyper-espace 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 H2G2.

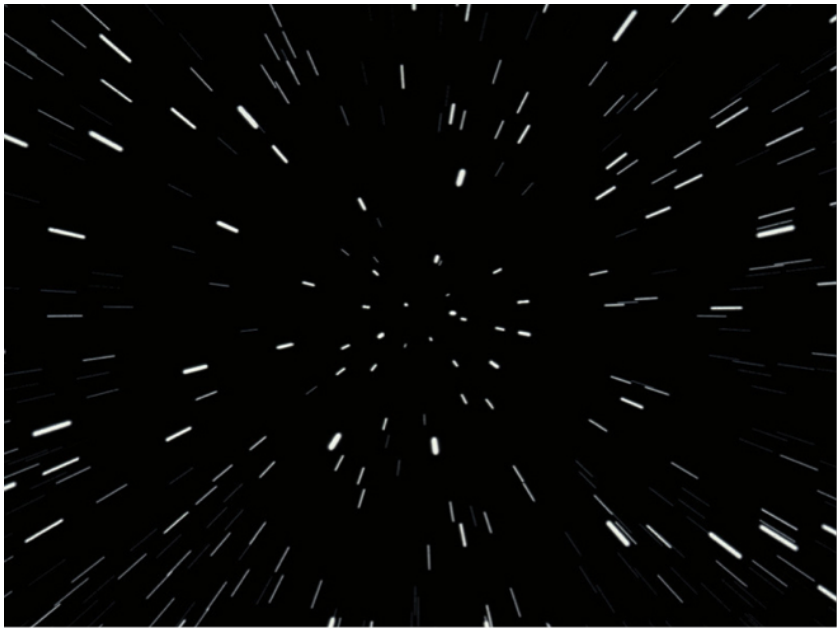


FIGURE 9 – Hyperespace dans StarWars

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 peut s'appliquer aux électrons voyageant dans les circuits de l'ordinateur.

8.1 Un objet pour chaque étoile : l'HyperspaceParticle

Chaque particule est en fait une file d'une autre sous-classe, composé d'un Rectangle et un flottant compris entre 0 et 1, représentant la transparence de la texture à dessiner. La particule prend en paramètre un angle, à partir du quel elle va calculer la vitesse, puis l'accélération de la particule, grâce à la taille de l'écran.

Voici les propriétés de la particule :
- Une file d'AlphaRect : Rectangle et alpha
- La taille maximale qu'une sous-particule peut avoir
- La taille maximale de la file
- Un flottant représentant l'alpha
- Un flottant représentant l'augmentation de l'alpha
- Une vitesse en abscisse et une en ordonnée

— Une accélération en abscisse et une en ordonnée

A 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 Next() renvoie un booléen valant "vrai", sinon on renvoie un booléen valant "faux".

8.2 L'HyperspaceEffect : gérer les particules

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 Next(). Si cette dernière renvoie la valeur "faux", la particule est retirée de la liste et détruite.

Finalement on obtient cet effet :

FIGURE 10 – L'HyperspaceEffect en action dans un menu

9 Un arrière-plan dynamique : le ScrollingBackground

Pour The Bytecoder, nous avons décidé de créer un jeu vidéo en 2 dimensions, estimant que ce genre de graphisme offre une meilleure expérience dans un jeu de plateformes. Ainsi, tout le jeu se déroule sur un seul plan. Dans l'optique de rendre le jeu le plus dynamique possible nous, avons pensé introduire un système de fond d'écran dynamique, composé de plusieurs plans défilant à des vitesses différentes. Celui-ci aura pour but de reproduire un effet bien connu par les yeux : lorsqu'on se déplace, plus un élément ou un paysage est éloigné, le plus lentement il se bougera par rapport à nous.

Ce paysage réaliste sera géré par une classe : le ScrollingBackground, qui gérera le chargement, les mise à jours et le dessin de cet "arrière-plan défilant".

9.1 Le chargement

Afin de le rendre plus réaliste, le fond est composé de plusieurs plans, eux mêmes composés de tuiles. Ces dernières sont en fait des images couvrant la hauteur de l'écran. Chacune d'elles est faite de manière à ce que la transition de l'une à l'autre soit imperceptible.

La classe prend en paramètre une chaîne de caractère indiquant le type d'arrière-plan à charger. Ce dernier porte le nom du monde correspondant (en effet l'ambiance et donc le fond d'écran sera caractéristique de chaque groupe de niveau, appelé monde). Le ScrollingBackground prend un second paramètre crucial pour le déplacement des tuiles : une instance de la classe Camera.

Pour charger les tuiles, le fichier de configuration du monde correspondant est ouvert (il porte le nom du monde suivi de l'extension *.wrld*). Il est ensuite analysé afin d'en déduire le nombre de plans (pouvant aller de 2 à « ce que la mémoire d'un ordinateur peut supporter »), le nombre de tuiles de chaque plan et location des images des tuiles. Ensuite chaque image est stockée dans un tableau à deux dimensions : une pour les plans, l'autre pour les textures en elles-mêmes.

9.2 Les mises à jour

Chaque tuile est en fait une classe appelée BackgroundSlab, dalle de l'arrière-plan en anglais, qui est composée d'une texture et d'un flottant, représentant la position en abscisse de la tuile. La partie centrale du fond d'écran est un tableau de listes de BackgroundSlab : une liste par couche.

La méthode Next(), qui gère les mises à jour de l'arrière-plan, déplace toutes les tuiles d'un dixième de la translation horizontale de la caméra fois le niveau de la couche (0 étant celle du fond) plus 1, dans le sens contraire de la translation de la caméra (*(- camera.xtranslation / 10) * layerdepth*).

Cette méthode vérifie aussi si les tuiles de chaque couche couvrent l'intégralité de l'écran. Si une est en dehors de l'écran elle se voit retirée de la liste. Au contraire, si une partie de l'écran n'est pas couverte par une tuile, on en rajoute une, à l'endroit non couvert, avec une texture aléatoire choisie parmi celles du plan concerné. L'algorithme vérifie en fait si la tuile qui se trouve la plus à gauche de l'écran sort de l'écran et de même pour celle la plus à droite.

9.3 Le dessin

La fonction de dessin du ScrollingBackground parcours du tableau de liste de BackgroundSlabs afin de dessiner méthodiquement chaque couche en commençant par celle le plus au fond. Sur la figure suivante, on peut observer un ScrollingBackground à 3 couches dans le fond. Chaque couche est représenté par une couleur différente.

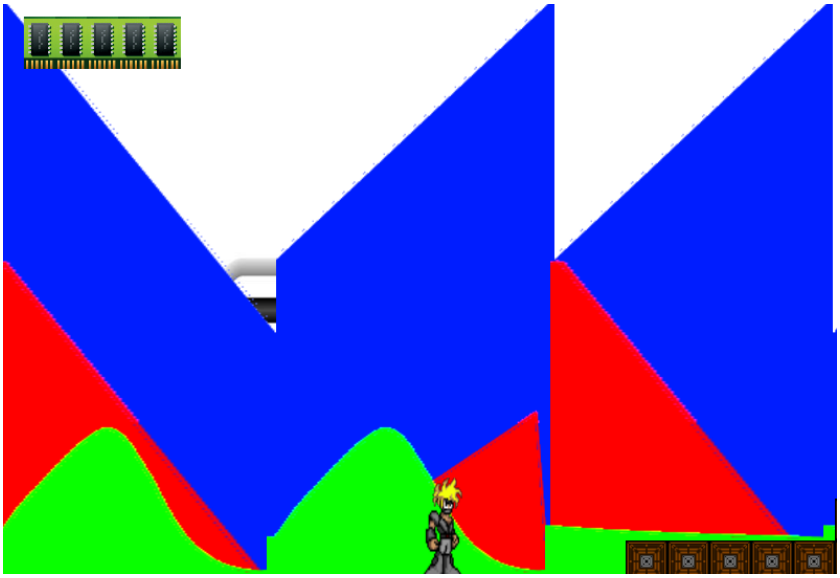


FIGURE 11 – Le ScrollingBackground de test

FIGURE 12 – Le ScrollingBackground du monde "Desert"

10 Le HUD

Le HUD, « HEAD-UP DISPLAY », est un outil qui affiche les informations dont le joueur a besoin. Cela comprend sa barre de vie, ses munitions, ses armes. Dans le cas du multijoueur, il sera possible de repérer son écran grâce à son numéro qui sera affiché. Actuellement, le HUD comprends le numéro du joueur et sa vie.

L'affichage de la vie se calcule par la différence des points de vie maximals du joueur et des dégâts qui lui ont été infligé. Pour rappeler à nouveau l'univers, on a choisi une barrette de RAM pour représenter cette valeur. Il a également fallu adapter la barre de vie dans les cas

où les points de vie restants représentent une valeur non multiple de dix, une barre de RAM représentant actuellement dix points de vie, pour cela on ne dessine qu'en partie cette barre. Par exemple, si le joueur a 45 points de vie restant, 4 barres vont être dessinés, puis la moitié d'une barre va être ensuite ajoutée.

11 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.

12 Amélioration de l'éditeur de niveau

12.1 L'interface

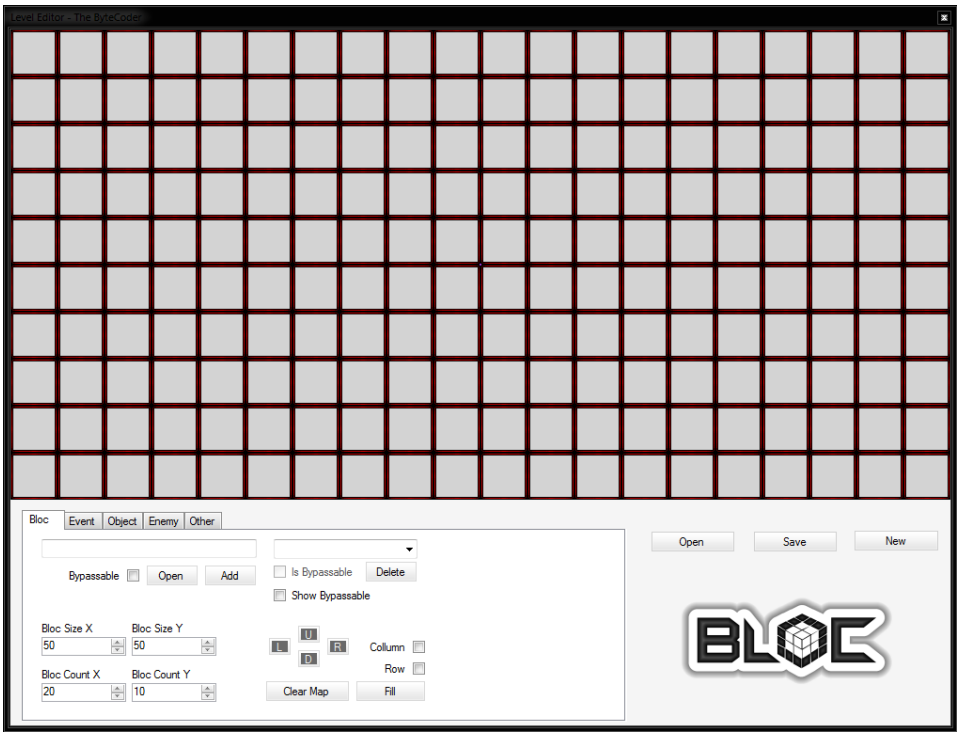


FIGURE 13 – Nouvelle interface de l'éditeur de niveau

Pour faciliter la conception des niveaux, la zone d'affichage a été agrandie. De plus, plusieurs fonctionnalités ont été rajoutées aux sections déjà existantes. Par exemple, pour faciliter le réagencement d'un niveau au cours de sa création, une croix directionnelle permet à l'utilisateur de déplacer tous les blocs de la carte dans la direction choisie. Un nouveau bouton a aussi fait son apparition, ce dernier permet de réinitialiser totalement le niveau.

12.2 L'onglet « Événement »

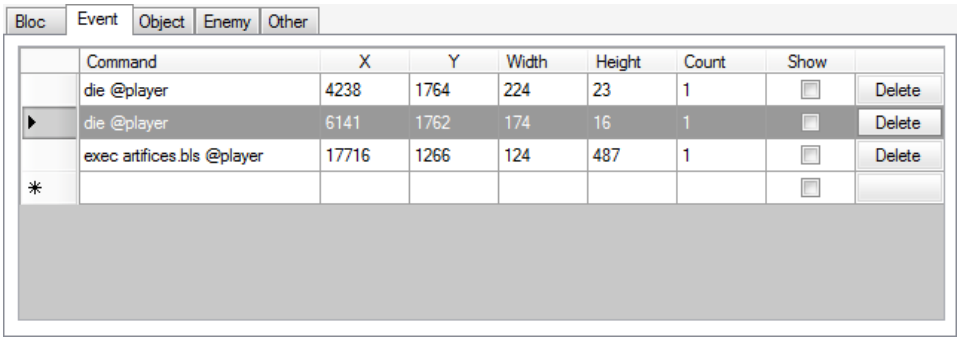


FIGURE 14 – Onglet permettant la mise en place d'événement dans le niveau

Pour créer un événement il suffit de remplir une nouvelle ligne. Il faut ensuite choisir directement sur la zone affichage où l'on veut le placer. Cela permet lors de la création d'un niveau, de fluidifier l'ajout d'événements, ce qui facilite la création. Après avoir placé un événement sur la carte, il est toujours possible de le modifier. Par exemple si l'utilisateur remarque qu'un élément est soit mal positionné ou mal orthographié, il peut modifier cela en sélectionnant le champs erroné et en le corrigeant. L'utilisateur possède aussi le choix de spécifier le nombre d'éxécution lors de le partie, ce champ est par défaut initialisé à 1. L'avant-dernier champ de cette grille permet d'afficher ou non les événements sur la carte, ce dernier est validé par défaut.

12.3 L'onglet « Objet »et l'onglet « ennemi »

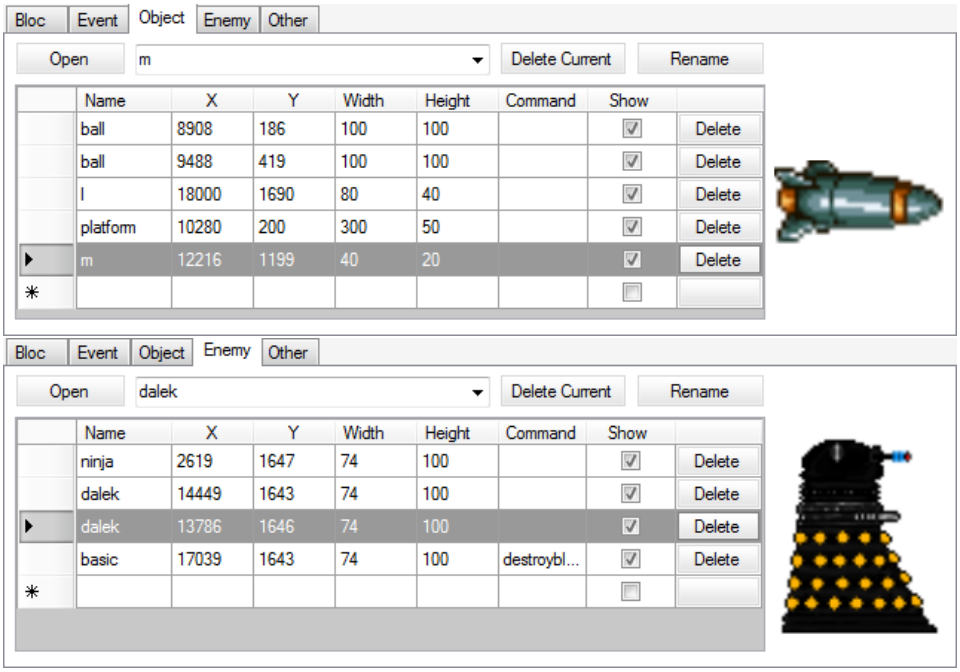


FIGURE 15 – Onglets permettant la mise en place d'objets scriptés et d'ennemis dans le niveau.

Ces deux onglets permettent de rajouter des objets scriptés ou ennemis, à la carte qui fonctionnent de façon similaire aux événements.Il faudra néanmoins ajouter un modèle objet ou d'ennemi dans la partie. Après avoir chargé les modèles, l'utilisateur peut sélectionner dans la liste l'élément qu'il veut rajouter et doit ensuite choisir la position de cette entité. Ici, pour changer le nom d'un objet ou d'un ennemi, on doit changer le nom du modèle correspondant.cela affectera toutes les entités déjà présentes ainsi que ceux qui seront ensuite ajoutés. La position peut également être changée, mais les valeurs concernant la taille sont invariantes et ne sont affichées qu'à titre indicatif. Par ailleurs un champ supplémentaire permet l'autre de la destruction de l'objet d'exécuter une commande, ce qui pourrait par exemple ouvrir une porte. Tout comme les événements, les objets et les ennemis peuvent être affichés sur la carte, si ces derniers possèdent une texture, celle-ci sera affichée.

12.4 L'onglet « Autre »

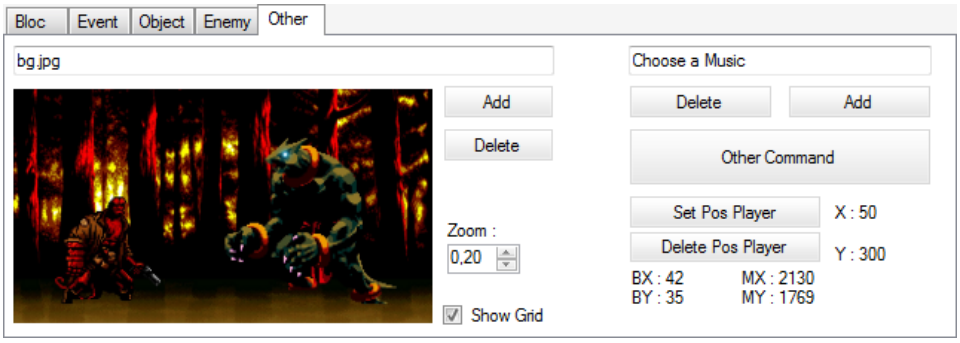
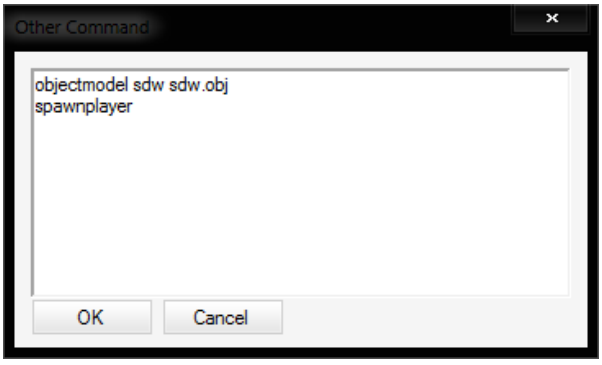


FIGURE 16 – Onglet permettant de changer tous les éléments ne rentrant pas dans les autres onglets.

Le dernier onglet permet, de choisir, par exemple, un fond d'écran parmi ceux proposés ou permet à l'utilisateur d'en ajouter, d'afficher la grille pouvant aider à la conception du niveau ou de choisir les coordonnées où le personnage apparaîtra au lancement de la partie. Cet onglet permet aussi d'afficher certaines informations pratiques pour l'utilisateur comme par exemple la position actuelle, en pixels et en nombre de blocs, de son curseur sur la carte, ce qui peut être utile lors de l'écriture de scripts spécifiques au niveau ou de commandes dépendantes du jeu pour être incorporées à ce niveau. Ces commandes peuvent tout de même être rajoutées avec l'éditeur via le bouton « Other Command », lors de son activation ce bouton va ouvrir la fenêtre visible ci-dessous :



Cette fenêtre permet d'afficher les commandes que l'éditeur ne peut pas interpréter, par exemple une commande dépendante de la taille du joueur, en effet ccette information n'est disponible qu'en jeu. Ces commandes, si existantes, n'étant pas interpréter par l'editeur ne pourront être corrigées ou signalées fausses.

13 Scénario

L'histoire se déroule dans un ordinateur dans lequel un virus a corrompu la sécurité. La plupart des programmes sont fortement affaiblis ou corrompus, à l'exception du ByteCoder qui a réussi à se protéger. C'est pourquoi ce dernier a pour mission de rétablir la stabilité du système. Il va devoir s'aventurer dans les différentes couches de celui-ci pour parvenir à ses fins...

La toute première couche étant le Web, il va d'abord rencontrer des ennemis tels que des ninjas lui lançant des shurikens, ou encore des représentants de FlashPlayer qui le poursuivent, ainsi que des rockets qui ne désirent que la mort du héros. Le ByteCoder va donc devoir survivre à tous ces opposants commandés par des sbires du Processus, celui-ci étant la cause de l'instabilité du système. A l'issue de ce premier monde, il va évidemment devoir affronter Apache, celui-ci étant une figure emblématique du monde Internet.

14 Conclusion

Ce projet étudiant est pour nous plus qu'un simple devoir. Il représente la possibilité d'exprimer notre créativité, de partager nos idées avec les membres de notre groupe, mais surtout de réaliser ces mêmes idées par nous-mêmes. C'est une occasion d'apprendre, d'une part la programmation et d'autre partle travail en équipe, mais aussi d'apprendre aux autres, de se mettre d'accord sur les méthodes de travail et de se rendre compte de la puissance qu'a une équipe bien organisée et soudée.

A travers ce rapport, on peut voir l'évolution de notre jeu. Cette deuxième soutenance marque en effet l'arrivée de nouveau modes de jeu, ainsi que l'approfondissemnt de certains designs, mais également l'arrivée de nouveaux éléments comme les sons, le HUD ou encore les attaques à distance. Le scénario est également en train d'évoluer, en effet, il prend forme au fur-et-à-mesure que le temps passe, et le jeu commence également à s'en rapprocher.