ABIMN

Rapport de soutenance

12 Mars 2014

Laure *Lyloox* **Daumal** (*daumal_l*)
Viviane *DragonDatas* **Lair** (*lair_v*)
Nicolas *Pvut* **Lesquelin** (*lesque_n*)
Alexandre *Tsunami* **Manuel** (*manuel_d*)

Rapport de soutenance 1

12 mars 2014

Table des matières

1 Introduction **2**
1.1 Petite introduction 2
1.2 Recap 2
1.2.1 Première soutenance : 3
1.2.2 Accompli jusqu'à présent 3
1.2.3 A faire pour la suite 3

2 Choix de C# et XNA **4**

3 Architecture **6**

4 Gestions des évènements **10**

5 Surcouche moteur graphique **12**

6 Tile mapping **14**
6.0.4 Choix de la représentation graphique 14
6.0.5 Les classe Cell/Map pour le TileMapping 15

7 Ressources graphiques **17**

8 Moteur Physique **21**

9 Intelligence artificielle **23**

10 Les crises du developpement **24**

11 Conclusion **25**

Chapitre 1

Introduction

1.1 Petite introduction

L'équipe de F0rM4g3 est heureuse de vous présenter, pour la première fois, son projet : Abimn

Depuis la validation du cahier des charges, les Développeurs ont étudiés pour créer ce monde hors de l'espace et du temps qu'est l'Abîme. Après deux mois de travail et le sacrifice de centaines de chatons pour peupler de nouvelles âmes, les maisons et les landes de notre univers, notre Héros au passé encore inconnu commençait à se matérialiser, particule par particule aux portes de la Mort.

Malgré un départ plus tenant de la tortue que du lièvre et plus d'une dissension au sein du groupe, nous ne pouvons vous montrer que les premiers pas de notre bébé. Pour rappel, Abimn est un Jeu de Rôle aventure 2D qui permettra au joueur d'incarner, dans des graphismes style "rétro", un fantôme à la recherche de son passé. L'histoire sera globalement linéaire mais des possibilités d'intéraction et quelques choix permettront au joueur de forger sa propre experience de jeu (et non totalement imposée). En évoluant dans le monde, combattant des ennemis et interagissant avec d'autres habitants, son Karma basculera vers le bien ou le mal et il accèdera au Paradis ou aux Enfers

1.2 Recap

Reprenons le tableau de prévision que nous avions fourni avec notre cahier des charges et aujoutons un tableau représentant l'avancement actuel de notre projet :

1.2.1 Première soutenance :

1.2.2 Accompli jusqu'à présent

	Lyloox	DDatas	Pvut	Tsunami	Total
Site Web	X	-	-	-	0%
Editeur	-	-	-	X	0%
Menu principal	X	-	-	-	50%
Editeur de personnage	-	X	-	-	0%
Interface d'exploration	X	X	X	-	60%
Menus ingame	-	-	X	-	70%
Interface de PNJ	X	-	-	-	0%
Interface de combat	-	-	X	X	30%
Intelligence artificielle	-	-	-	X	50%
Système de sauvegarde	-	X	-	-	0%
Ressources graphiques	X	X	X	X	30%
Ressources sonores	X	X	X	X	0%

1.2.3 A faire pour la suite

	Lyloox	DDatas	Pvut	Tsunami	Total
Site Web	X	-	-	-	100%
Editeur	-	-	-	X	100%
Menu principal	X	-	-	-	50%
Editeur de personnage	-	X	-	-	70%
Interface d'exploration	X	X	X	-	70%
Menus ingame	-	-	X	-	100%
Interface de PNJ	X	-	-	-	30%
Interface de combat	-	-	X	X	80%
Intelligence artificielle	-	-	-	X	70%
Système de sauvegarde	-	X	-	-	0%
Ressources graphiques	X	X	X	X	60%
Ressources sonores	X	X	X	X	50%

Actuellement les menus ingame ont pris un peu d'avance, ils sont entièrement gérables à la souris ce qui ne devait à la base pas être le cas. Nous prévoyons pour la prochaine soutenance d'avoir fait le site web, l'éditeur de cartes, de commencer à implémenter des PNJ et du son, et enfin de faire évoluer l'intelligence artificielle et l'interface de combat.

Chapitre 2

Choix de C# et XNA

De par les chartes du projet, nous ne pouvions choisir qu'entre deux langages de programmation : C# et Ocaml.
Pour choisir définitivement entre les deux outils (et trancher les désaccords qui rongeaient notre esprit d'équipe) nous avons effectuer quelques recherches sur les bibliothéques grahiques.
Passons en revue les différentes bibliothèques graphiques que nous avons trouvé pour ces deux langages :
OCaml :

1. SDL
2. Ocsfml
3. ocaml-sfml
4. librairie graphics par l'INRIA

C# :

1. SDL.NET
2. Binding SFML pour C#
3. XNA

On voit déjà rapidement que la SDL et la SFML, qui sont deux bibliothèques graphiques qui sont portées en OCaml et en C#. Concernant la librairie graphics native d'OCaml que nous avons eu l'occasion de tester en TP, nous avons décidé de la laisser de côté car elle est très bas niveau et difficile à maîtriser par des novices en programmation.
Nous avons trouvé que le C#, par rapport à l'OCaml, est plus simple à prendre en main et plus proche d'une conception entièrement objet. Nous avons donc choisi le C# pour être notre heureux élu, tout d'abord pour des questions de confort.
Ajoutons à celà que pour développer en C#, Microsoft Visual Studio nous accompagne déjà bien plus que par exemple Emacs pour l'OCaml. Un environnement de développement du niveau de Visual Studio n'existant pas pour l'OCaml, nous avons encore penché en faveur du C#.

Revenons au choix de la librairie graphique. Il nous reste la SDL, la SFML et XNA. Parlons un peu d'XNA. Nous avons là une bibliothèque parfaitement en harmonie avec le langage car elle été développée spécifiquement pour le framework .NET Celle ci est entièrement objet et respecte donc bien la philosophie du C#. La SDL, elle, est réputée pour être une bibliothèque de nature impérative (elle a été créée pour le langage C à l'origine qui lui n'est absolument pas orienté objet). Celle ci étant une bibliothèque très bas niveau, nous avons également décidés de la laisser de côté.
La SFML elle, est belle et bien orientée objet, seulement, après un long choix d'au moins dix secondes, nous avons fini par préférer XNA pour plusieurs raisons :

1. Le support des élèves en Spé qui ont majoritairement utilisé XNA pour leur projet de Sup

2. La documentation sur msdn

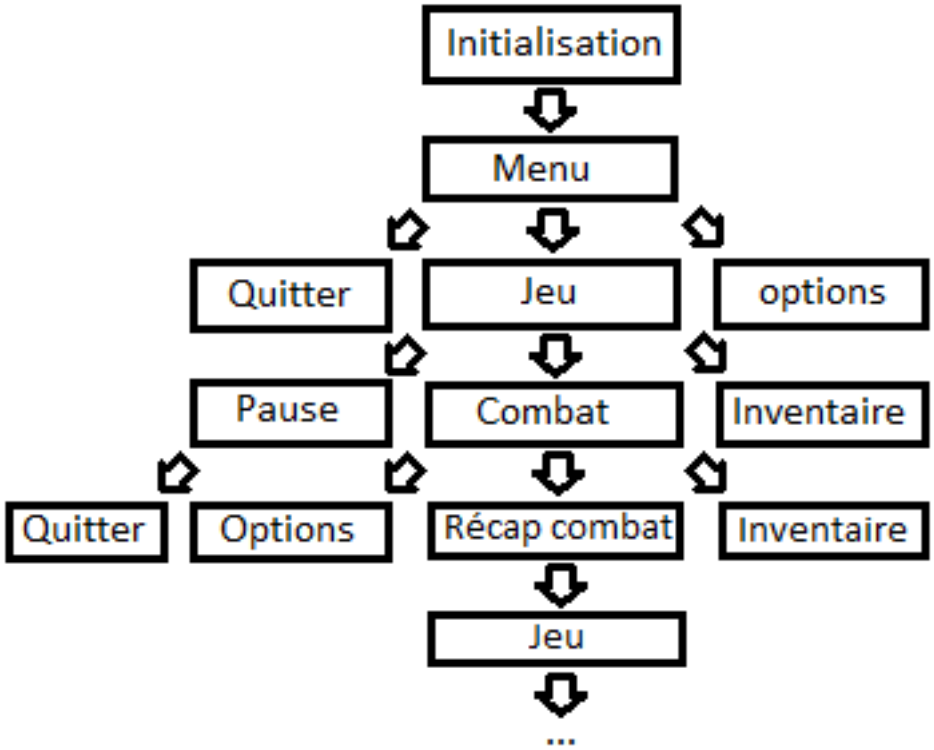
3. Les différentes optimisations pensées pour le milieu du jeu vidéo par Microsoft

4. L'expérience acquise avec cette bibliothèque lors des TP d'Informatique Pratique

Chapitre 3

Architecture

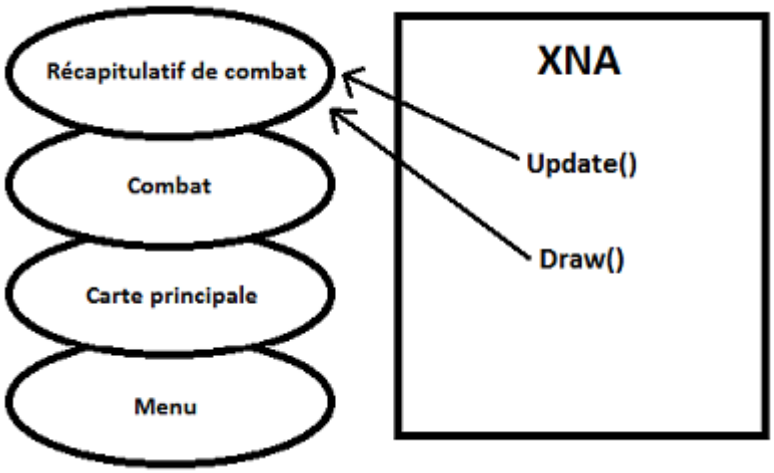
Premièrement, nous avions envisagé une architecture de code plutôt vieille à cause d'un habitué du bas niveau que nous ne nommerons pas. Tout était une approche impérative du problème.
Le programme commençait par faire des initialisations générales puis appelait une fonction qui en appelait une autre qui en appelait elle même une autre, etc.....En bref, voici quel était le schéma de fonctionnement du jeu :



Cependant, cette architecture empêchait clairement d'utiliser la puissance

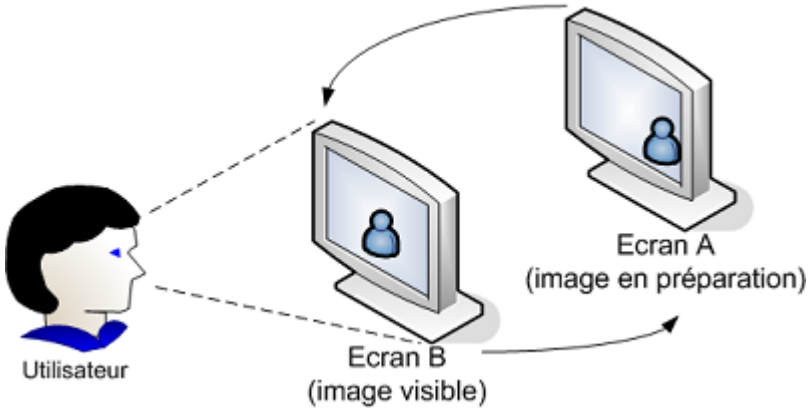
d'XNA. En effet, il est ainsi bien difficile de dissocier calculs de jeu, du dessin des entités à l'écran alors qu'XNA bénficie d'optimisations intéressantes suu ce point de vue (telles que la diminution de FPS quand le jeu perd le focus)
C'est pourquoi nous avons décidé de changer de système pour gérer tpit ceci différemment, de manière plus adaptée au fonctionnement d'XNA.

Tout est basé sur une pile. C'est l'élément au sommet de cette pile qui définit quel est l'élément du jeu à faire fonctionner à un instant t. Comme un schéma vaut mieux qu'un long discours, voici de quoi illustrer mes propos :



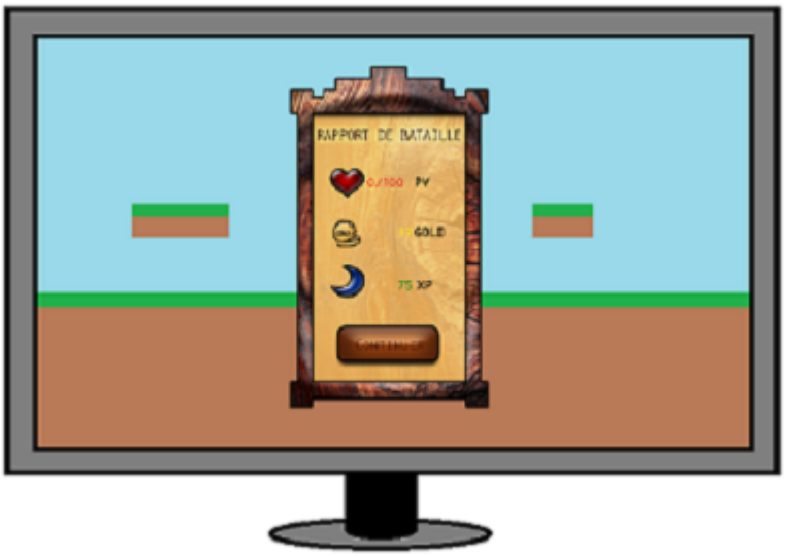
Cette pile est l'équivalent de la pile d'appell de fonctions dans la première architecture. Voyons cependant ce qui change. Premièrement, chaque pierre de la pile est en fait liée à un ensemble de fonctions qui correspondent (comme par hasard) aux fonctions de l'architecture classique d'un jeu fait avec XNA.

Chaque pierre est donc composée d'une fonction Update, d'une fonction Draw et d'une fonction Initialize. Cette dernière est appelée lors de la pose de la pierre concernée sur la pile. Quand à Update et Draw, seules sont appelées celles de la pierre du haut de la pile. Il est en effet inutile de continuer de dessiner le menu alors que nous sommes en plein jeu.
Seulement un problème s'est alors posé : XNA utilise une technique appelée le double buffering.

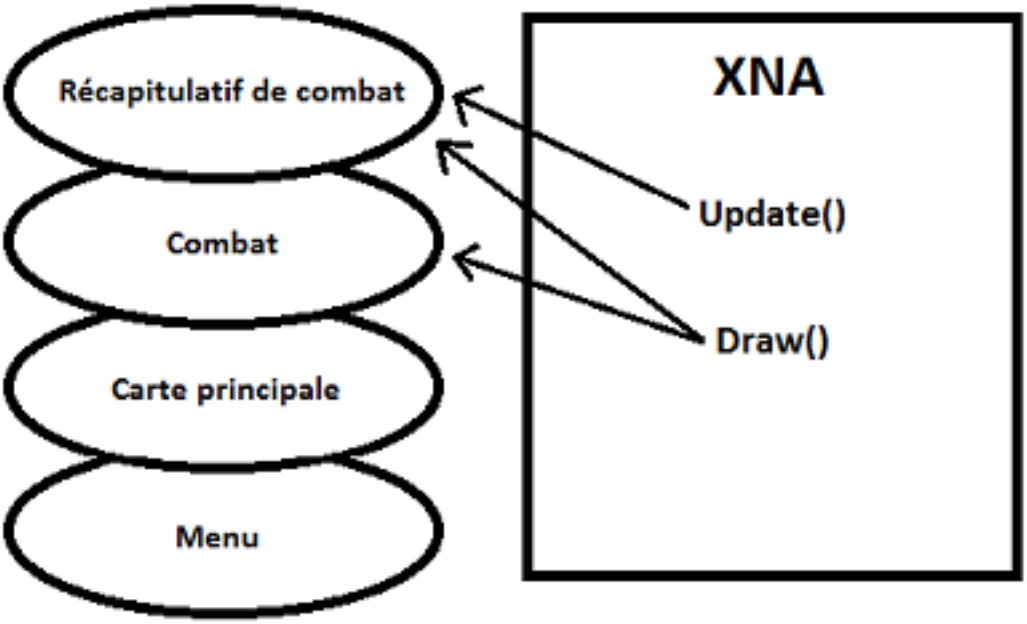


Le principe est d'avoir deux écrans en mémoire. Le premier (écran A) n'existe qu'en mémoire contrairement au deuxième (écran B) qui représente l'écran en préparation. Quel est l'intérêt de cette technique ? La phase de dessin d'une frame peut être assez longue. Ainsi, pour ne pas dessiner sous les yeux de l'utilisateur et ainsi faire scintiller son écran, on préfère préparer tous nos dessins dans un endroit caché en mémoire (écran A) pour ensuite, à l'affichage demandé de la frame, échanger les deux écrans. L'échange d'écrans étant extrêment rapide, c'est invisible pour l'utilisateur et il aura ainsi l'impression de voir la frame se construire.

Cette technique a un petit défaut pour nous . XNA, à l'échange des deux écrans, commence par effacer l'écran qui se retouve à l'arrière afin de nous permettre de partir d'une base de dessin propre. Seulement dans le cas du récapitulatif de combat par exemple , on aimerait garder la dernière frame du combat dessinée derrière le récapitulatif. Celui-ci n'occupe pas la totalité de l'écran comme on peut le voir ci-dessous :



Il en devient ainsi nécessaire de remonter d'au moins un élément dans la pile pour dessiner ce qui se trouve en dessous de notre élément flottant. On se retrouve donc avec le schéma suivant :



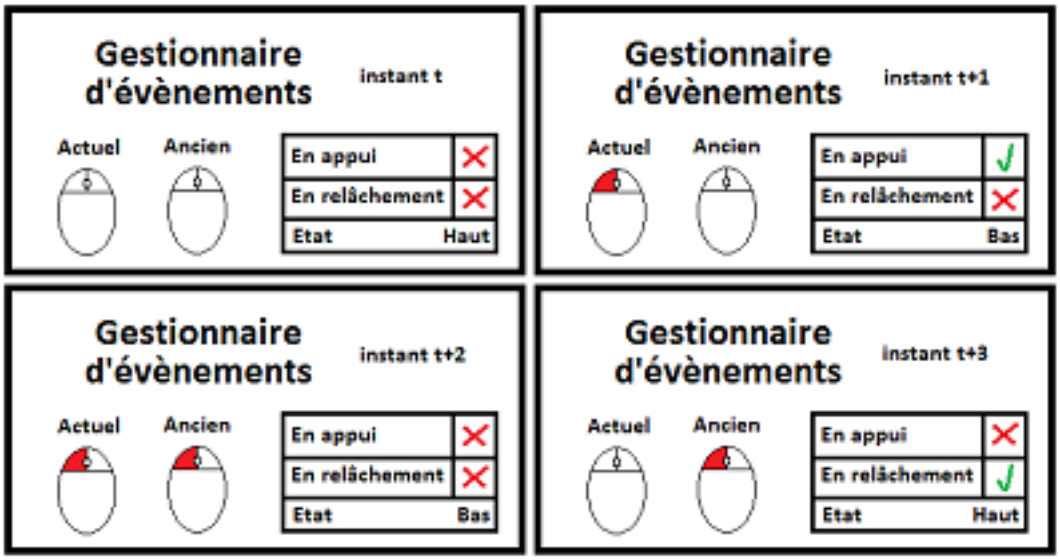
Bien évidemment on ne veut qu'un aperçu de ce qui se trouvait en dessous du récapitulatif. On ne veut pas que le combat continue de se dérouler. C'est pourquoi seule la méthode Draw est appelée sur la pierre Combat et pas la méthode Update (on voit ici l'intérêt de séparer le dessin du calcul).

Chapitre 4

Gestions des évènements

Concentrons-nous un peu sur la méthode Update. C'est là que l'on va gérer les entrées du clavier et de la souris pour que nottre programme interragisse en fonction. XNA fournit les outils nécessaires à flasher l'état du clavier et de la souris. Le problème est que ce flash est lent. Nous avons donc décidé de ne faire cette opération qu'une seule fois par appel à Update.
Le but est de permettre à tous les prgrammeurs du jeu de pouvoir accéder à un flash récent des périphériques d'entrée de données. On a donc un objet fermé qui se charge de la gestion de tout ceci.
Détaillons son fonctionnement.
Afin de savoir si une touche vient d'être enfoncée à un instant t, il faut non seulement qu'elle soit enfoncée à l'instant t, mais également qu'elle ait été en position haute á l'instant t-1. C'est pourquoi il est nécessaire de garder en mémoire deux flash. Le flash de l'instant t et le flash de l'instant précédent.

On se rapproche très fortement de la psychologie d'un double buffering, mais à l'envers. Ce qui est conservé dans les treffonds cachés de la mémoire, c'est l'ancien flash, et la future frame.
Le principe est évidemment le même pour les boutons de la souris. Le schéma ci-dessous illustre le fonctionnement de la souris pour son clic gauche.



Le flashage des périphériques d'entrées ne se fera qu'une seule fois par appel de la fonction Update par XNA, et non pas dans chaque pierre de la pile (heureusement !)
Le modèle est ainsi au maximum de son optimisation

Chapitre 5

Surcouche moteur graphique

Afin de gérer proprement les graphismes avec XNA, nous avons créé une surcouche au moteur. Sont ainsi nées trois classes dont nous nous servons dans tout le programme.

Classe Tile :
Chaque objet instancié à partir de cette classe représente une image liée au projet. Pour des questions d'optimisation, nous avons décidé de charger la totalité des images une et une seule fois au lancement du jeu. Celles ci n'étant pas trop nombreuses, nous n'observons pas de ralentissement au lancement du programme. Cela a pour avantage de n'avoir chaque image qu'une seule fois en mémoire et non une image par entité en jeu (ce qui pourrait mener à une duplication d'images s'il y a plusieurs entités de même nature).
Toutes nos Tiles sont rangées dans des tableaux globaux afin de pouvoir s'en servir depuis toutes les briques du jeu. Elles sont initialisées et chargées au lancement du programme.

Classe Entity :
Toute entité en jeu est représentée par un objet de type Entity ou d'un type enfant. A chaque entité est associé une tile (par référence bien sûr , on ne duplique pas les tiles), une position et un vecteur vitesse. Ceci rend chaque entité en jeu très maléable. L'idée est qu'à chaque Update, la position varie en fonction du vecteur vitesse associé.

Classe Button :
Les boutons (qu'on retrouve dans tous les menus, écrans de confirmation, etc) sont des entités tellement à part que nous avons décidé d'en faire une classe à part entière. Tout d'abord, ceux-ci ne bougent pas à l'écran (pas besoin de vecteur vitesse donc) mais en plus, ils disposent tous d'une gestion

à la souris grandemment simplifiée par les méthodes de cette classse.
Trois tiles sont associées à chaque bouton. Une pour le bouton dans sa position normale, une quand il est survolé par la souris et une quand le bouton est enfoncé.
Toutes les tiles étant passées par référence, la non-utilisation d'effet de survol/enfoncement est parfaitement optimisée car un espace mémoire pour une seule image seulement sera nécessaire à notre programme dans ce cas.

Chapitre 6

Tile mapping

6.0.4 Choix de la représentation graphique

L'un des premiers choix auquel à été confronté notre équipe lors de la création de notre projet à été le choix du type de carte que nous voulions utiliser et le type d'exploration pour notre personnage L'alliance des deux devait être suffisament agréable pour le joueur, avec une prise en main facile et surtout possédant un niveau de code à la portée de tous les membres du groupe. Plusieurs choix s'offraient à nous :

1. Tout d'abord, savoir si nous devions faire le jeu en 2D ou 3D. La jeu en 3D aurait, dans l'absolu, donné un bien meilleur rendu et aurait donné au côté exploration un aspect peut-être plus attractif. Mais compte tenu des possibilités de notre équipe en création artistique et manipulation d'image, rendre les graphismes suffisamment fluide et fourni pour faire des paysages agréables et immersifs compatible avec le type d'univers onirique que nous recherchions, aurait été tr'es complexe pour ne pas dire pratiquement impossible. Par égard pour les yeux de nos futur joueurs, nous avons donc très vite rayé cette possibilité

2. Restait encore à choisir entre la 2D et la 2D isométrique. Des jeux trés récents ont fait le choix de revenir a un style de graphisme en 2D isométrique, preuve s'il en est du plébiscite de ce système,utilisé dans un nombre incalculable de jeu, aussi bien des jeux de console que des jeux d'ordinateur RPG tel que *le légendaire Baldur's Gates*. Mais l'idée de tabler sur un décalage d'utiliser le côté extrèmement rétro du graphisme 2D plat pour trancher sur l'univers (pouvant accueillir aussi bien fantastique que science fiction) était également a envisager.

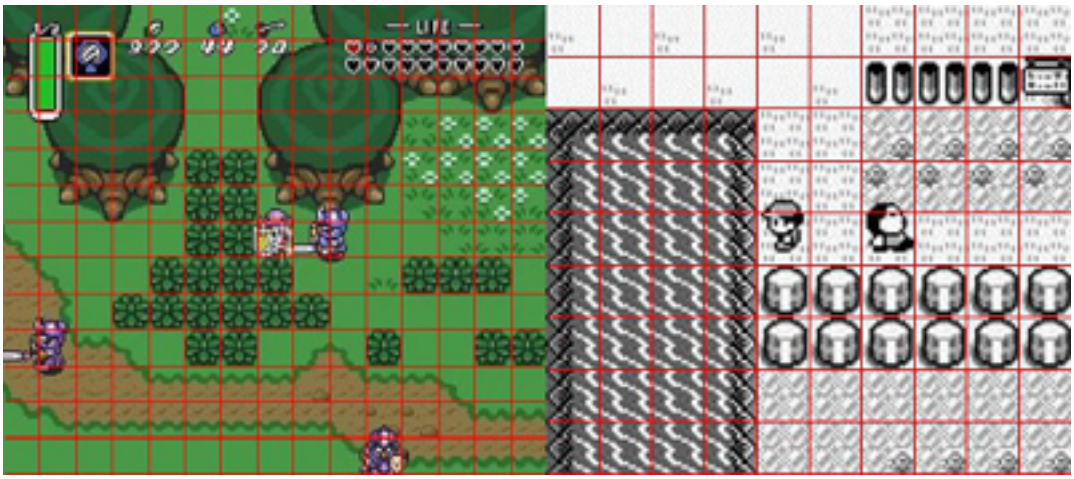
3. Là dessus est venu le problème du second mapping : l'écran de Combat l'Utilisation de la 2D, isométrique ou non, n'allait pas avec le dynamisme que nous voulions donner á nos séquences de combat. Nous avions décidé de faire des affrontements en temps réel et non au tour par tour, plus statique, strategique et calculatoire que purement ludique. Comptant sur l'effet addictif des combats pour redynamiser l'ensemble du jeu, une idée nous est venue : pourquoi ne pas utiliser une instance de map totalement différentes. Nous avons donc décider de présenter les combat de profil, comme un MarioBros, alors que l'exploration et les actions sociales seraient gérés depuis une carte plus conventionnel et trés siimple. Les deux seraient en pixel art et 2D pour garder le côté décalé.

6.0.5 Les classe Cell/Map pour le TileMapping

Le plus simple pour créer la carte d'exploration principale était le tile mapping. Détaillons ensemble un peu cette technique.
Le principe d'avoir une mao constituée d'un tableau de cases Chaques case est iddentifées par un couple de coordonnées. Il devient aunsi très simple de faire une gestion de collisions entre le personnage et son environnement. Le personnage ne fait que se déplacer de case en case, dans les cases qui veulent bien l'accepter.
Cette technique permet :

- Une gestion simplifiée des collisions
- une économie non négligeable de mémoire d'un point de vue quantité d'images
- Un import/export simplifié de cartes dans des fichiers
- L'existence d'un éditeur puissant
- Une carte fractionnée, et donc affichable par partie

Afin d'avoir une grande carte, le héros se déplace dans une carte qui bouge selon ses mouvements. Grâce au Tile mapping, seule la partie visible de la carte est affichée. Le seul défaut de cette technique est un manque de liberté graphique dans la carte du jeu. Néanmoins, cette technique est très utilisés encore de nos jours dans de très nombreux jeux vidéos en deux dimensions. L'utilisation de cette technique est flagrante dans des jeux tels que les premiers Pokémons ou les premiers Zelda.


— LIFE —

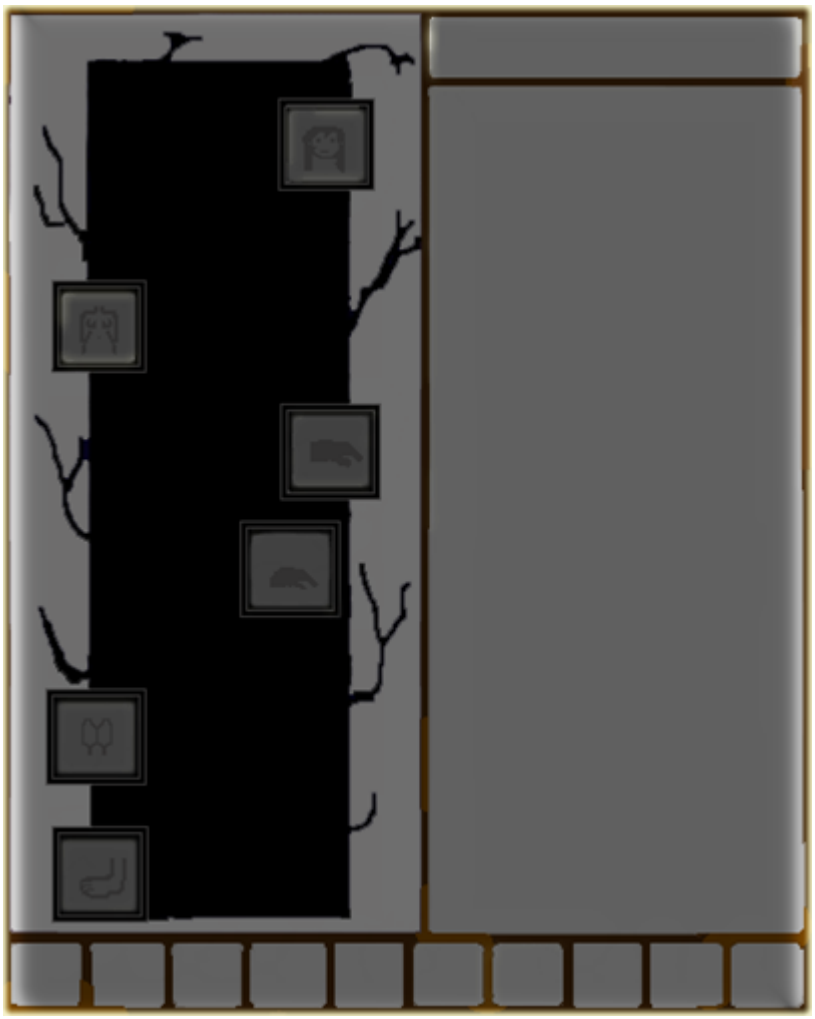
Chapitre 7

Ressources graphiques

Afin d'avoir un Rpg un tant soit peu intéressant et facile à prendre en main, il vaut mieux avoir les ressources adéquates. notre jeu fonctionnant en 2D, nous aurons besoin au moins d'un fond (qui est le décor, communément appelé "map"), et de personnages qui s'y déplacent.

1. Décors.
   DDatas à réalis2 la carte, qui s'affiche en tant que fond durant la boucle de Jeu. Pour celà, elle a dessiné differents types de terrains (principalement du gravier et de la pelouse ) Ainsi que des éléments particuliers comme des buissons ou des fontaines possédant tout deux des propriétées physiques

Pvut a aussi déssiné le fond qui servira de base pour l'inventaire :



Avec la possibilité d'équiper un chapeau, une armure, une arme à chaque main, des jambières et des chaussures.
En bas se trouve une barre de raccourci clavier pour accéder plus rapidement aux objets.

2. Sprites
   Les "sprites" représentent un élément de notre jeu. Par exemple, ces dessins de Pvut sont des sprites :

    ,

   Et les renards de Lyloox sont aussi des sprites :


,
Mais ce qui donne réellement un intérêt à ces sprites, c'est qu'ils soient animés. En effet, si l'élément est amené à se déplacer (ce qui est un minimum indispensable pour un personnage que l'on souhaite jouer), on préfèrera que ses déplacements soient fluides afin de donner un aspect réel à ceux-ci. Nous avons ici besoin de sprites afin de donner cette animation et éviter de donner l'impression que le personnage se téléporte tout simplement.

Voici 3 sprites de héros, déssinés puis colorisés par Lyloox :

1.  2.  3.  ( avant colorisation)

C'est l'enchainement (1,2,3,2,1) de ces sprites qui donnera un aspect animé et fluide au dplacement vers la droite du personnage.

Déplacement vers la droite du héros secondaire :

1.  2.  3. 

Afin de gérer tous les déplacements possibles, il a aussi fallu prévoir les directions haut et bas.
Voici les sprites secondaires et non-animé de ces deux déplacements :

Chapitre 8

Moteur Physique

EN combat, on voit la scène de côté, ce qui permet au personnage de faire des sauts. Le problème de réalisme du saut se pose alors. On ne procèdera pas comme une entité classique dans ce cas donc.
Un saut lié à une courbe lineéaire serait trop irréaliste, et à ce moment là plusieurs options s'offrent à nous :

- Implémenter un moteur physique externe pour calculer les sauts
- Calculer les sauts avec des fonctions mathématiques du second degré
- Implémenter la seconde loi de Newton

La première solution a vite été mise de coté lorsque l'on a cherché des moteurs. La pluparts sont plus adaptés à des calculs en 3D, et ils sont de toute façon bien trop complets et complexe pour s'en servir seulement pour faire un saut.
Il nous restait alors le choix des deux derniers. La seconde loi de Newton a l'avantage de nous fournir une parabole réaliste sans que l'on ait à chercher l'équation de la dite parabole.
Nous avons donc, bien sur retenu la troisiìeme solution.
Afin d'y parvenir, il faut utiliser le gameTIme donné à la fonction Update de la pierre du combat pour faire un saut en fonction du temps.
On trouvera en utilisant les axes paramétriques les formules suivantes :

$$x_(t) = v_x \times t$$

$$y_(t) = v_y \times t - \frac{gt^2}{2}$$

Ici t repr='esente le temps en milisecondes écoulés depuis le début du saut, g

la constante gravitationelle que nous définirons en dessous de 9.81N/kg pour permettre à notre personnage de sauter plus haut (en diminuant la gravité donc)
v représente le vecteur vitesse initial que nous calculerons comme suit :
$v_x = v_0 \times \cos\theta$
$v_y = v_0 \times \sin\theta$
avec v_0 un vecteur qui représentera la force dans les jambes du personnage et θ l'angle de saut que nous définirons à 60^o en mouvement et à 90^o à l'arrêt.

Chapitre 9

Intelligence artificielle

Les armes de tir à dostance n'étant pas implémentées, l'ennemi a tout intérêt à foncer vers le personnage pour l'attaquer. L'ennemi est donc capable pour l'instant de se diriger vers le personnage, et éventuellement de faire un saut pour l'atteindre. La recherche du plus court chemin a été abandonnée pour cette soutenance pour des raisons de lenteur et de complexité algorithmique. Les sauts permettant ici des mouvements complexes, la recherche devenait trop longue sans optimisation importante de l'algorithme. Nous avons donc décidé de rendre l'IA plus simple mais plus fluide pour cette soutenance.

Chapitre 10

Les crises du developpement

Un problème que nous avons découvert assez rapidemenet et auquel nous ne nous attendions pas á été de garder notre projet à jour de la même façon sur tous les ordinateurs de chaque membre du groupe.

En effet, le travail de développement en groupe, bien que divisant grandement la charge de travail et permettant de confronter les points de vues et les idées d'optimisations, réserve des surprises inconnues aux codeurs soliraires. Bien qu'ayant répartie le travail de façon assez cloisonnée depuis la mise en place de l'Architecture, il est indispensable qu'au bout du compte le Projet comporte toutes les modifications nécessaires aux ajouts de chacun. Les développeurs n'avançant pas toujours au même rythme ni au même moment , passer le projet de clé en clé ou par e-mail est vite devenu très lourd. EN plus d'être long, nous nous perdions dans les différentes versions qui étaient aprfois mal référencées ou obsolètes.

Une mesure rapide et efficace a été utilisé : les transfert de fichers de Skype. En plus de permettre des échanges rapides datés et chronologiquement clair pendant les séances de travail communes, la fonction de partage d'écran aidait également à clarifier certaines manipulations nécessaire au changement du code global (Constante et variable globales).

S'il n'a pas été nécessaire d'utiliser un logiciel spécifique pour le début du code il nous apparaît maintenant que se former rapidement à Git serait nettement profitable, non seulement pour travailler ensemble plus facilement mais aussi pour garder des sauvegardes régulières. Il est donc fort probable que nous passions à ce logiciel de version d'ici la prochaine soutenance.

Chapitre 11

Conclusion

Le travail effectué pour cette soutenance nous a permit d'établir une première architecture assez claire et assz bien cloisonnée pour établir les bases des éléments principuw du jeu. Pourtant, tel un squelette, il manque encore de corps et surtout de l'âme au projet (un comble pour un monde de fantôme) : le héro et la quête qui sont en cours de développement . À l'avenir, nous devront affiner les déplacemetns, le système de combat et les options mais également implémenter de nouveaux contenus pour pouvoir transporter le joueur dans notre univers.