PEUVERGNE Evan – Chef de projet H1 G2 P2018

RIVIERE Magali – Responsable design

ALLOUIS Arnaud – Responsable communication

LEMAHIEU Lucas – Responsable développement

Rapport projet labyrinthe

Troisième trimestre 2013-2014

hétic
La grande école de l'Internet

Table des matières

L'analyse fonctionnelle générale 3
Choix importants lors de la conception du projet 4
Les fonctions principales 5
L'analyse fonctionnelle détaillée 8
Génération du labyrinthe 9
Les déplacements joueurs 11
Les lignes de vue et la lumière 11
Les attaques 12
Les barres de vie et de faim 13
Sauvegarde des données 14
Les monstres 15
Le multijoueur 16
Annexes 18
Technologies tierces employées 19
Bibliographie et webographie 19

L’analyse fonctionnelle générale

Choix importants lors de la conception du projet

Un des premiers choix a été de choisir entre **un point de vue** entièrement 3D, une vue isométrique dite 2.5D ou une vue de dessus en 2D. La vue 3D a rapidement été abandonnée car trop complexe et trop gourmande pour être mise en place dans les délais demandés. Nous avons finalement tranché en faveur d'une vue 2D, bien plus légère en ressources, mais permettant d'obtenir de bons rendus visuels avec des graphismes fait main. Mais cette vue nous a surtout permet de nous intéresser plus profondément à deux éléments majeurs de notre jeu : le gameplay et le multijoueur.

Notre **gameplay** a été basé sur un axe simple : un labyrinthe qui se respecte se doit d'être obscur, terrifiant et oppressant. En fonction de cela, nous avons décidé de mettre en place des lignes de vue très courte pour l'aspect obscur, d'inclure des monstres plus ou moins vicieux pour l'aspect terrifiant, et enfin pour l'aspect oppressant d'exclure une quelconque carte et d'ajouter une « dead-line » sous la forme d'une barre de faim et de vie. Mais ne joueur ne sera pas sans défense : il pourra se défendre grâce à des armes récupérée dans le labyrinthe ou achetée dans la boutique, qui sera mise en place ultérieurement.

Le **multijoueur** quant à lui a été développé de façon à être un atout majeur du jeu, qui permet au joueur de découvrir le jeu sous un nouvel angle. Ainsi plutôt que l'angle PvE (*Player versus Environment*) du mode solo, le multijoueur est axé sur un angle PvP (*Player versus Player*). Ainsi il n'y aura pas de coopération entre les joueurs (sauf alliance tacite, mais il n'y aura finalement qu'un seul vainqueur), mais une forte compétition pour la survie. Seul le premier arrivé à la sortie gagne la manche ou bien le dernier en vie. D'où le grand dilemme : courir pour trouver la sortie, ou se battre pour être le dernier survivant ?

Pour ce qui du **thème général** de notre jeu et de son aspect visuel, il s'est accordé avec le gameplay : sombre et quasi-glauque. Nous nous sommes alors orientés vers un univers dit *Steampunk* ou rétro futuriste, agrémenté d'une touche de médiéval pour renforcer le côté obscur. Tous les designs ont été réalisés sur ordinateur par notre responsable design, sous forme vectoriel permettant un rendu bien supérieur à une image .jpeg étirée ou compressée.

Illustrations d'univers *Steampunk* :

Dans la sélection du **modèle de labyrinthe**, le choix fut rapide : il nous fallait un labyrinthe dit parfait, où un chemin unique passe par toutes les cellules. Cela nous permet de complexifier l'aspect solo du jeu car un seul chemin est disponible, mais également d'assurer au(x) joueur(x) que la sortie générée aléatoirement sera dans tous les cas accessible.

Pour le **stockage** des différentes données, nous avons fait appel à des tableaux de données JavaScript à une, deux ou trois dimensions et à des tableaux JSON (JavaScript Object Notation est un format de données textuelles, générique, dérivé de la notation des objets du langage JavaScript). Le format choisit est fonction de la donnée à stocker, afin de choisir l'option la plus efficiente. Pour ce qui est du stockage des données utilisateurs sur le serveur, nous avons utilisé le système de gestion de base de données MySQL.

Les fonctions principales

Génération du labyrinthe

La fonction globale de génération du labyrinthe a été la première à être mise en place. Déjà complexe dans sa mise en œuvre, plusieurs filtres ont également dû être pris en compte :

- Génération dynamique du code HTML correspondant au labyrinthe
- Gestion de l'état « déjà visité » ou « pas encore visité » case par case
- Nécessité de stocker la présence ou l'absence de mur pour les collisions, les lignes de vue etc.
- Nécessité de stocker le chemin parcouru dans un « fil d'Ariane »
- Ne pas sortir du tableau lors de l'exploration/création du labyrinthe
- Arriver à un résultat viable et jouable à travers une bonne gestion de l'aléatoire de la direction choisie

Les collisions

Le stockage des murs lors de la génération du labyrinthe permet de pouvoir tester les collisions aussi bien pour le(s) joueur(s) que pour les monstres, ou encore la lumière. Bien qu'omniprésent, les tests de collisions sont basés sur un test très léger (la valeur stockée est-elle égale ou différente de 0 ?), et ainsi n'alourdissent pas le jeu.

Les déplacements joueurs

Les déplacements du ou des joueurs seront contrôlés par l'utilisateur, et uniquement lui. Pour mettre cela en place, nous avons inclus un écouteur d'évènements sur les touches directionnels (haut, droite, bas et gauche). En fonction de la direction, un test de collision est demandé, et s'il n'y a pas de collision, l'animation de déplacement se lance. On modifie également la variable « position joueur », pour bien lier la position du joueur à la case du labyrinthe correspondante.

Plus tardivement, la notion d'orientation du joueur a également été incluse afin de faciliter les attaques et le système de combat en général.

Les lignes de vue et la lumière

Pour renforcer l'aspect effrayant et claustrophobe du jeu, les lignes de vue sont primordiales : quel est l'intérêt d'un labyrinthe, si on voit l'intégralité de la carte ? Rien de plus qu'une gymnastique intellectuelle. Ici, les lignes de vue gérées de façon « réaliste » (on ne voit pas à travers les murs), et sont également dynamiques : il va soit falloir faire travailler sa mémoire, soit utiliser papier et crayon.

Les attaques

Les attaques permettent tout d'abord de survivre aux monstres, en les tuant en premier. Plusieurs armes sont implémentées, mais elles se subdivisent en deux grandes catégories : les armes de corps-à-corps et les armes à distance. L'attaque réagit également à écouteur d'évènement pour une touche précise.

Les barres de vie et de faim

La notion de mort et de game over est indispensable pour entretenir le suspense voir l'angoisse du joueur lors d'une partie. La mort peut intervenir de deux façons :

- mort violente (être tué par l'attaque d'un monstre voir d'un autre joueur en multijoueur) quand la barre de vie atteint zéro
- mort d'inanition (ce qui correspond à un chronomètre adapté au gameplay de notre jeu) quand la barre de faim atteint zéro
-

Sauvegarde des données

Pour la sauvegarde du jeu lors d'une partie locale, nous utiliserons le localStorage pour garder les informations nécessaires le temps de relancer cette partie.

Les monstres

Les monstres possèdent leurs propres caractéristiques (attaque, vie, vitesse de déplacement). Ils se déplacent aléatoirement dans le labyrinthe.

Le multijoueur

Système quasi-indépendant du reste du jeu, le multijoueur est un défi en lui-même. Il est géré grâce à des requêtes serveurs, qui doivent synchroniser les informations de tous les joueurs mais doit également les retransmettre en temps réel. Il peut pour le moment être considéré comme étant en phase béta.

L’analyse fonctionnelle détaillée

Génération du labyrinthe

Utilité : générer aléatoirement un labyrinthe parfait et en stocker les caractéristiques pour une utilisation ultérieure.

Entrées : nombre de case total, nombre de case par ligne

Sorties : labyrinthe parfait, tableau contenant l'intégralité des murs du labyrinthe

1ère étape : création d'un tableau HTML correspondant au format du labyrinthe. A chaque case du tableau créé, on associe dans un tableau JS à une dimension la valeur booléenne false (signifiant que la case n'a pas encore été explorée) ; et dans un second tableau JS à deux dimensions, on associe pour chaque case quatre valeurs, correspondant respectivement aux murs Nord, Est, Sud et Ouest. Pour l'instant, toutes les cases sont initialisées comme ayant 4 murs.

2ème étape : en partant de la cellule actuelle (pour la première case, il s'agit par défaut de la case 0), on teste chaque case concomitante pour sélectionner les cellules voisines possibles (et donc étant encore dans l'état booléen false). On stocke les voisins possibles dans un tableau éphémère, dans lequel on sélectionne aléatoirement le nouveau voisin. Si aucun voisin n'est possible, on passe directement à l'étape 4.

3ème étape : on ouvre le mur se situant entre les deux voisines (un mur pour chaque voisine), et on stocke la valeur de la case actuelle dans un tableau, qui servira de fil d'Ariane, puis on passe notre « curseur » sur la nouvelle voisine, qui devient ainsi la case actuelle. Le booléen de cette case devient true (= déjà visitée). On repasse à l'étape 2.

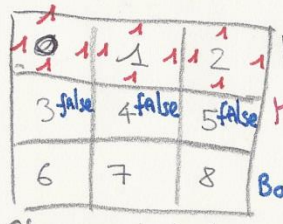
NB : lors du premier passage par cette étape, on active l'étape 5.

4ème étape : si aucun voisin n'est disponible, cela signifie que nous sommes arrivés à un cul de sac. On retourne donc sur nos pas, en sélectionnant comme case actuelle l'avant dernière valeur du tableau fil d'Ariane. Puis on supprime la dernière valeur du tableau fil d'Ariane. On repasse à l'étape 2.

5ème étape : écouteur d'évènement : si la valeur de la case actuelle est égale à la première valeur stockée dans le tableau servant de fil d'Ariane, on arrête les étapes 2, 3 et 4 et on passe à l'étape 6. En effet si on revient à la toute première cellule, cela signifie que le labyrinthe est entièrement généré, car toutes les cases ont été visitées.

6ème étape : affichage graphique du labyrinthe. On teste pour chaque case chacun de ses murs, et s'il n'y a pas de mur, on affecte pour ce mur la même couleur que le reste de la cellule : le mur disparait graphiquement.

Étape 1 : labyrinthe "vierge"



Numéro case

Murs (0 = vide, 1 = plein)

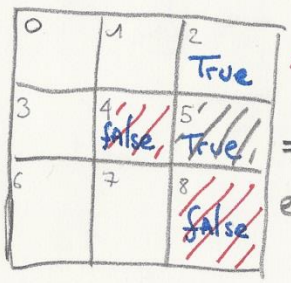
Booléen

Chaque est identique
case

Étape 2 : cherche voisins

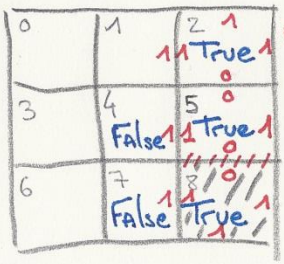
/// cellule actuelle

/// voisins possibles



=> on choisit aléatoirement entre 4 et 8.

Étape 3 :

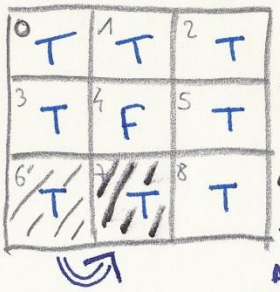


.... suppression mur

Murs (0 = vide, 1 = plein)

/// cellule actuelle (Maintenant True)

Étape 4 : retour en arrière



T : true

F : false

/// cellule actuelle dans un "cul de sac"

/// dernière cellule actuelle

-> Retour en arrière : la cellul
actuelle n'est plus 6 mais 7.
On y trouve un chemin vers 4

Les déplacements joueurs

Utilité : se mouvoir dans le labyrinthe

Entrées : position actuelle du joueur, position du labyrinthe en arrière-plan, direction de la touche entrée par le joueur, vitesse de déplacement du personnage

Sorties : en cas de déplacement viable, animation visuelle de déplacement et réécriture de la position actuelle du joueur.

Un constat simple : on ne peut déplacer le joueur et laisser le labyrinthe en arrière fond immobile ; ce doit être le labyrinthe qui bouge et le joueur qui reste fixe. Ainsi, il peut explorer l'intégralité du labyrinthe sans crainte de sortir de la fenêtre de l'utilisateur.

1ère étape : rotation joueur. Avant même de tester si le déplacement est possible ou non, on stocke la nouvelle direction du joueur (Nord, Est, Sud ou Ouest) et on change l'image du joueur pour un rendu visuel. Cette étape servira grandement pour les futures fonctions d'attaque.

2ème étape : test mur. On test sur la case du labyrinthe correspondant à la position actuelle du joueur le mur correspond à la direction choisie. Si présence d'un mur : on arrête la fonction. Si absence de mur, on passe à l'étape 3.

3ème étape : on ajuste la position du labyrinthe en fonction de l'inverse du déplacement du joueur. Par exemple pour un déplacement du joueur vers la gauche, on décalera le labyrinthe vers la droite. On stocke également la nouvelle position du joueur.

4ème étape : on appelle la fonction de lumière et de ligne de vue pour les réajuster à ce déplacement.

Les lignes de vue et la lumière

Utilité : permettre une gestion de la vue dynamique, réaliste et oppressante

Entrées : position actuelle du joueur

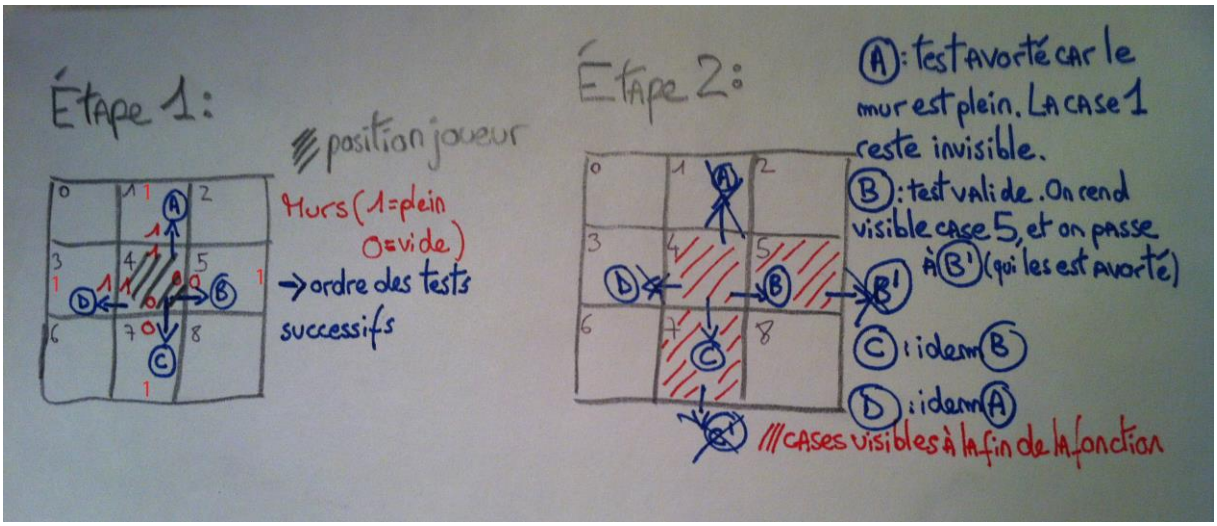
Sortie : uniquement visuelle

Par défaut, toutes les cases du labyrinthe sont invisibles et ne s'affichent donc pas.

1ère étape : on lance l'étape 2 pour le Nord puis l'Ouest puis le Sud puis l'Est. Une fois ces quatre directions testées, la fonction s'arrête.

2ème étape : on regarde le mur de la cellule correspondant à la position actuelle du joueur, dans la direction souhaitée. S'il y a un mur, on revient à l'étape 1. S'il n'y a pas de mur, on rend visible la cellule suivante, et on teste à son tour son mur.

Cette fonction est comprise dans une boucle while, qui arrête de tourner une fois qu'on a exploré 4 cases consécutivement.



Les attaques

Utilité : pouvoir se défendre contre les monstres

Entrées : la direction du perso, la portée de l'arme, les dégâts de l'arme

Sorties : si présence d'un adversaire, lui inflige des dommages

1ère étape : en prenant en compte la direction du joueur, on parcourt un nombre de cases correspondant à la portée de l'arme que l'on porte (1 pour les armes de corps à corps, plus pour les armes à distances). Si on croise un ennemi, on passe à l'étape 2.

2ème étape : on soustrait aux points de vie du monstre les dégâts correspondant à l'arme utilisée. On passe à l'étape 3.

3ème étape : si les points de vie du monstres sont inférieurs ou égaux à zéro, le supprime le monstre

Les barres de vie et de faim

Les barres de vie et de faim sont représentées algorithmiquement par une variable chacune.

Barre de vie

Utilité : déterminer la santé du joueur et ainsi déclencher un game over en cas de mort « violente »

Entrée : dommages infligés par l'adversaire (déclencheur)

Sorties : modification de la variable dédiée, modifications visuelles pour signaler le changement à l'utilisateur, déclenchement de la fonction game over si mort détectée

1ère étape : on soustrait les dommages causés à la variable vie. On affiche en fonction un visuel pour informer le joueur.

2ème étape : si la variable vie est inférieure ou égale à 0, déclenchement de la fonction game over.

Barre de faim

Utilité : déterminer la faim du joueur et ainsi déclencher un game over si le joueur meurt de faim. Cela correspond à un chronomètre : le joueur a un temps limité pour sortir de ce labyrinthe et passer au suivant, et cela permet également d'assurer le climat oppressant de notre jeu

Entrée : rien

Sorties : modification de la variable dédiée, modifications visuelles pour signaler le changement à l'utilisateur, déclenchement de la fonction game over si mort détectée

1ère étape : toutes les x secondes, cette étape est appelée. On passe à l'étape 2.

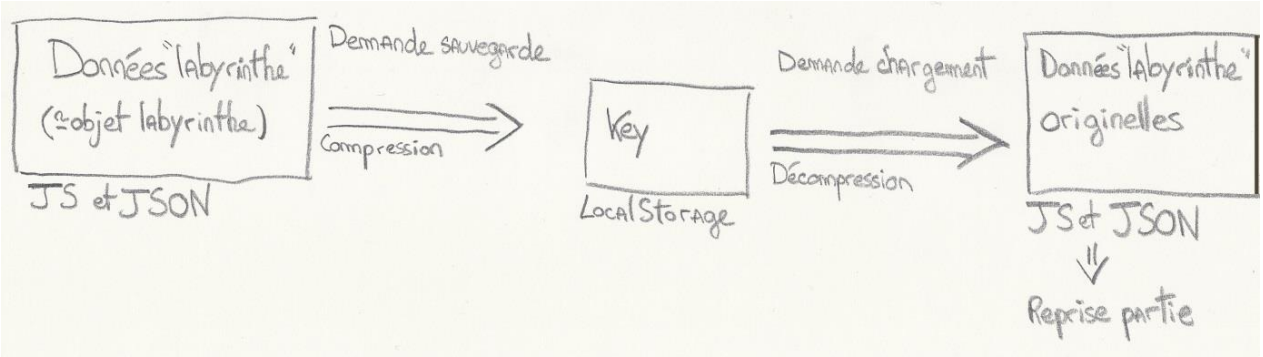
2ème étape : on soustrait un montant y à la variable faim. On affiche en fonction un visuel pour informer le joueur.

3ème étape : si la variable vie est inférieure ou égale à 0, déclenchement de la fonction game over.

Sauvegarde des données

Afin de sauvegarder la partie en cours dans le mode solo, nous utiliserons le localStorage qui permet de sauvegarder sous forme de Key valeur des ensembles d'objet. Le labyrinthe (et tous ses constituants) n'étant qu'un objet gigantesque il est très simple de sauvegarder l'état actuel et l'avancement de l'utilisateur et de lors de son retour sur le jeu de lui restituer.

Le localStorage pourrait être comparé à des cookies bien plus puissants (quelques Ko contre plusieurs Mo pour localStorage) et qui n'encombrent pas le trafic http. De plus, ils sont persistants : ils survivent à la fermeture d'un onglet ou d'une fenêtre d'un navigateur.



Les monstres

Les monstres représentent un dilemme sur le plan de la performance spatiale et temporelle. Chaque animation de déplacement d'un monstre consomme un nombre non négligeant de ressources, et le nombre de monstre est croissant avec les niveaux.

Pour résoudre ce problème, nous avons créé un algorithme qui empêche les animations de se lancer si elles se situent à plus de 17 cases de la position du joueur (hors de l'affichage fenêtre donc). Mais les monstres continuent tout de même à se déplacer : seule leur variable indiquant leur position est modifiée.

Utilité : permet de dynamiser le labyrinthe avec des monstres vivants. Augmente également la difficulté car le monstre peut se déplacer pour attaque le joueur.

Entrées : position actuelle du monstre, vitesse déplacement monstre

Sorties : animation ou non du déplacement, changement de la variable position du monstre

1ère étape : on teste un à un les murs de la case correspondant à la position du monstre. S'il y a un mur on passe directement au suivant, sinon on stocke la case suivante dans cette direction dans un tableau éphémère.

2ème étape : on choisit par un tirage aléatoire parmi les cases possibles

3ème étape : on détermine si le monstre se situe « dans l'écran » du joueur. Si oui on passe à l'étape 4, sinon directement à l'étape 5.

4ème étape : on lance l'animation de déplacement du monstre puis on passe à l'étape 5

5ème étape : on stocke dans la variable position du monstre sa nouvelle position

Le multijoueur

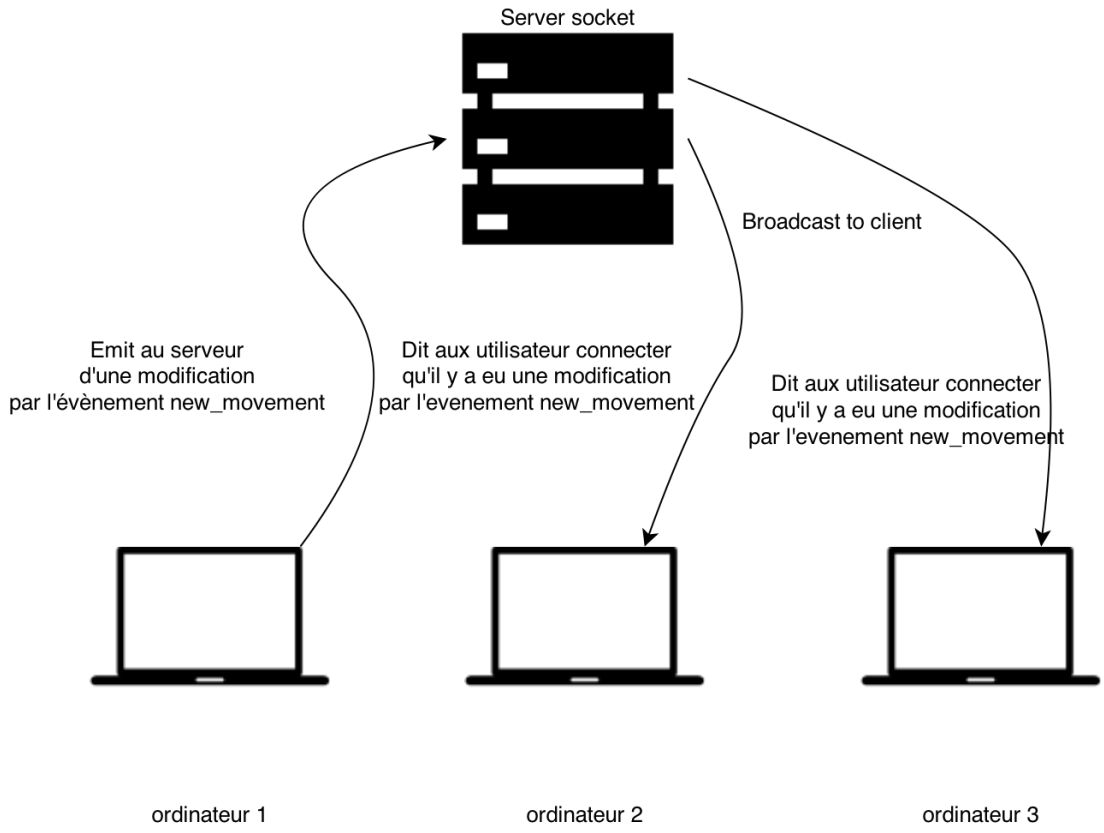
Pour l'aspect multijoueur nous avons utilisé socket.IO qui permet d'utiliser les différents sockets (« connecteur réseau » ou « interface de connexion » qui aide à recevoir et expédier des données) disponibles dans les différents navigateurs. De nombreux jeux vidéo utilisent cette technologie car cela évite de devoir réinventer la roue pour du realtime (temps réel). Elle nous permet (cette technologie) de pouvoir faire évoluer différentes personne sur un même labyrinthe et cela malgré le fais qu'ils n'utilisent pas le même navigateur. Nous avons donc un serveur nodeJS qui permet de faire marcher le serveur Socket.IO. Les sockets ont un fonctionnement simple mais qu'il faut arriver à cerner au début. C'est le principe de Client/Server poussée à son paroxysme. Le client peut envoyer un serveur une requête qui va être lue par le serveur qui va pouvoir lui-même envoyer un message juste à l'utilisateur ou le broadcaster à tous les utilisateurs qui sont sur une room. Parlons des room un peu. Le fonctionnement est plutôt simple. Les utilisateur peuvent se connecter à une room et pourront ainsi recevoir des messages qui ne sont destinés qu'à eux. Quand nous parlons de messages nous voulons bien-sûr dire data, des ensembles d'objets.

Le serveur peut cibler une seule room et ainsi envoyer aux bonnes personnes. Cela nous permet de créer différents partie qui sont en réalité sur plusieurs room et de broadcaster les différentes positions des joueurs au bon joueur dans la bonne room. S'il n'y avait pas le moyen de mettre les utilisateurs dans une room il ne pourrait pas avoir plusieurs parties. Le fonctionnent est vraiment celui-là, il y a une boucle infinie qui va écouter des évènements aussi bien côté client que serveur et il ne va pas pouvoir réagir dessus. Pour notre cas par exemple à chaque touche clavier un évènement est envoyer au serveur avec en paramètre la room. Le serveur reçoit l'évènement et broadcast (= envoie simultané) à tous les autre utilisateurs de la même room sur un certain évènement les data qu'avait envoyé l'utilisateur. Tous les clients excepter l'utilisateur expéditeur de ces data reçoivent les donnée sur l'évènement envoyer par le serveur. C'est une fonction de callBack qui renvoie un tableau d'objet de ce qu'avait envoyé le client au serveur.

Pour le projet du labyrinthe nous avons une surcouche en plus qui est Sails. Elle permet d'agglomérer d'autre technologie nodeJS et de créer un site complet. C'est le mélange de ExpressJS qui permet de gérer les routes et de créer un site dynamique. Pour nous cela nous permet de créer le système de login et de registrer que nous avons inclus. Il nous permet de gérer les pages de création d'item, d'ennemies, de renvoyer du JSON pour gérer notre propre API REST que nous avons conçu pour l'occasion.

Cet api que nous avons conçu nous permet d'imaginer un futur plus simple pour le jeu. Quelqu'un sans connaissance dans le développement pourrais concevoir lui-même des objets sans connaissance en JavaScript. Une interface claire et concise existe déjà dans l'administration pour cela. Il y a aussi la présence d'un puissant orm qui nous permet de faire des requête sur des bases de donnée quel que soit leurs type: MongoDB, Mysql, Redis.

Ce beau mélange nous permet de pouvoir avoir un jeu avec des fonctionnalités toutes pas intégrer de moduler le jeu rapidement et de lui rajouter du contenu et des fonctionnalités. La version solo pourrai par exemple par un simple système de login sauvegarder les parties directement sur notre base de données pour permettre à l'utilisateur de continuer une partie où qu'il soit. La grande modularité c'est ce qui permet de pouvoir rajouter des fonctionnalités au jeu rapidement. Pouvoir faire le plus d'ajout de fonctionnalité dans un produit comme un jeu est un point positif.



Annexes

Technologies tierces employées

Node.js

Node.js est un environnement très bas niveau, qui nous permet d'utiliser le langage JavaScript sur le serveur. Il nous permet donc de faire du JavaScript en dehors du navigateur. Node.js bénéficie de la puissance de JavaScript pour proposer une nouvelle façon de développer des sites web dynamiques. Il est nécessaire pour le fonctionnement de Socket.IO, entre autres.

Socket.IO

Socket.IO est l'une des bibliothèques les plus prisées par ceux qui développent avec Node.js. Pourquoi ? Parce qu'elle permet de faire très simplement de la communication synchrone dans une application, c'est-à-dire de la communication en temps réel. Cela est nécessaire pour la mise en plus du multijoueur.

JQuery

JQuery est un framework qui permet d'agir sur le code HTML, CSS, JavaScript et AJAX. Nous nous en servons entre autres pour les animations des déplacements des personnages.

Sails

Sails.js facilite la création d'applications personnalisées, de classe entreprise Node.js. Il est conçu pour imiter le modèle MVC comme Ruby on Rails, mais avec un soutien pour les applications modernes: API orientées données/objets avec l'architecture orientée évolutivité. Il est particulièrement bon pour la mise en place de chat, de tableaux de bord en temps réel, ou de jeux multijoueurs.

Bibliographie et webographie

- http://fr.wikipedia.org/wiki/Mod%C3%A9lisation_math%C3%A9matique_d%27un_labyrinthe
- http://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra
- http://fr.wikipedia.org/wiki/Algorithme_A*
- http://socket.io/#how-to-use
- http://nodejs.org/api/
- http://sailsjs.org/#!documentation