Année 2013 - 2014

RAPPORT DE PROJET – EI4 AGI

Zomb'IstiA

Jeu multijoueur temps réel en ligne

Projet réalisé par

Abdelmounaim BELGHALEM
Johan BRUN
Brendan GOUIN
Florian HARDY

Projet encadré par

Mehdi LHOMMEAU

ISTIA – Université d'Angers

Remerciements

Avant tout, il nous semble important de commencer ce rapport en remerciant M. Mehdi LHOMMEAU, notre encadrant, qui nous a suivi durant ce projet et qui s'est toujours montré disponible pour répondre à nos questions.

Nous souhaitons également exprimer notre reconnaissance à toutes les personnes ayant contribué de près ou de loin au développement de cette application.

Tout d'abord à Marc BEAUFILS, pour ces précieux conseils et avoir pris le temps de nous enseigner quelques astuces en développement web.

Nous souhaitons également exprimer notre sympathie à Damien DARROU et Guillaume FACHE, pour avoir pris le temps de tester la première version de notre plateau de jeu et nous avoir fait des retours constructifs; ainsi qu'à Justine JEANNIN, pour avoir grandement aidé à l'élaboration de la maquette de la deuxième version du plateau de jeu.

Pour finir, nous remercions Claudine STEFANT de nous avoir annoncé la bonne nouvelle du déploiement de notre application sur l'Internet lorsque nous étions attelés à cette tâche.

Table des matières

Introduction

Présentation du document

Ce document fut conçu dans le cadre du projet tuteuré de la 2ème année de cycle ingénieur dispensé pour le diplôme d'ingénieur par l'ISTIA d'Angers (Institut des Sciences et Techniques de l'Ingénieur d'Angers). Il fut rédigé en trois semaines et clôt le projet.

Ce document est le rapport récapitulant tout ce qui a été élaboré par notre équipe durant les 24 semaines de projet, sur une période allant du 11 Novembre 2013 au 25 Avril 2014. Il présente la façon dont nous avons géré ce projet, les technologies utilisées, le travail réalisé puis établit un bilan de ce projet.

Contexte d'écriture

Le présent document a pour but d'informer l'équipe pédagogique de toute l'organisation de notre projet, ce que l'on a produit, ce que l'on a appris, les outils utilisés, ainsi que les difficultés rencontrées.

Remarque : Ce document pouvant être par moment technique, un glossaire a été rédigé en fin de ce document afin d'expliquer certains termes. Ces derniers seront écrits en rouge afin de signifier au lecteur que ce terme fait partie du glossaire.

Présentation de l'équipe

Notre groupe est composé de 4 étudiants de la promotion des 2èmes année du cycle ingénieur (EI4), suivant le parcours Automatique et Génie Informatique (AGI) de l'ISTIA d'Angers. En voici les membres :

- Abdelmounaim BELGHALEM
- Johan BRUN
- Brendan GOUIN
- Florian HARDY

Nous étions encadrés tout au long de ce projet par M. Mehdi LHOMMEAU, enseignant-chercheur à l'ISTIA.

1. Présentation du projet

1.1. Contexte

Ce projet fut réalisé dans le cadre des projets tuteurés d'EI4. C'est un sujet que nous avons proposé car nous avions envie de créer notre propre jeu multijoueur. Nous avons initialement rédigé un cahier des charges, comprenant les fonctionnalités de l'application, ainsi qu'une maquette de l'interface client. Après quelques modifications et validation des spécifications avec notre encadrant, notre projet était lancé. Nous avions alors la tâche de créer un site web respectant ce cahier des charges, en utilisant les technologies préconisées.

1.2. Objectifs pédagogiques

Ce projet s'inscrit dans le cadre de notre formation en utilisant les connaissances et compétences en développement logiciel apprises durant les cours dispensés par l'IstiA, et a pour objectifs de nous faire découvrir de nouvelles technologies, de mettre en œuvre nos capacités en gestion de projet à travers un travail en équipe, ainsi que de renforcer nos compétence en développement logiciel, à travers l'utilisation de langages de programmation.

1.3. Idée générale du produit

Globalement, l'objectif du projet est de créer un jeu de rôle multijoueur par navigateur web (comme Google Chrome ou Mozilla Firefox), comme on peut en trouver une multitude sur l'Internet. Ce genre de jeu se range dans la catégorie des ORPG pour "Online Role Playing Game", pour "Jeu de rôle en ligne " dans la langue de Molière. Le jeu sera présenté sur un site web, afin que les joueurs puissent accéder au jeu à l'aide d'un navigateur web. Contrairement aux jeux pour consoles ou ordinateur, l'aspect graphique n'est pas la priorité (pas de moteur graphique), mais l'accent est mis sur le nombre de règles et de possibilités de jeu. Le jeu sera dit *temps réel* (une action a instantanément un impact sur le jeu et les autres joueurs) et *persistant* (le jeu continue de fonctionner quand les joueurs sont déconnectés).

1.4. Contraintes

Pour réaliser ce projet s'imposaient à nous des contraintes de temps, mais également des contraintes technologiques. En effet, le projet a commencé Novembre 2013 et la date finale était le 25 Avril 2014, date de la soutenance de projet, ce qui nous laissait un peu plus de 5 mois de travail. Théoriquement, nous disposions de cinq heures par semaine, un créneau horaire réservé par l'ISTIA, et dédié aux projets que nous avons bien entendu utilisé. Concernant les contraintes technologiques, nous avions initialement proposé à Monsieur LHOMMEAU, responsable des projets de notre promotion, de développer notre application en C# et en utilisant les WPF et WCF (technologies crées par Microsoft pour le développement d'applications). Cependant, comme nous avions déjà des connaissances avec ces technologies, nous avons décidé, d'un point de vue pédagogique, d'utiliser les technologies conseillés par l'enseignant, c'est à dire Node.js, HTML5, Canvas et MongoDB.

Remarque : Nous n'avons eu aucune contrainte financière, car nous disposions chacun des outils nécessaires au développement de l'application.

2. Technologies et outils utilisés

2.1. Langages de développement

2.1.1. HTML 5 - CSS 3

Afin de réaliser le site web, nous avons utilisé deux langages. Ceux-ci, bien qu'ayant des rôles différents, sont complémentaires.

- **HTML** (*HyperText Markup Language*) : créé en 1993, ce langage gère et organise le contenu du site. C'est un langage de balises (définies par < >, les plus connues sont <html> et <body>). Chaque balise à sa signification et son utilité (par exemple, écrire Mon mot affichera "Mon mot" en gras). C'est donc en HTML que nous avons écrit ce qui doit être affiché sur la page : du texte, des liens, des images...

Figure 1 - Logo HTML5

- **CSS** (*Cascading Style Sheets*, aussi appelées *Feuilles de style*) : créé en 1996, il gère l'apparence de la page (agencement, positionnement, décoration, couleurs, taille du texte…). Un fichier CSS est composé de plusieurs règles de présentation qui permettent d'appliquer des styles aux balises HTML.

Figure 2 - Logo CSS3

En résumé, le HTML définit le contenu. Le CSS permet, d'arranger ce contenu et de définir la présentation : couleurs, image de fond, marges, taille du texte…

L'un des avantages et nouveauté du HTML5 est sa balise appelée "Canvas". C'est un espace de pixels transparents de base, qui permet avec du code JavaScript de réaliser un bon nombre de fonctions graphiques, allant du simple tracé de courbe jusqu'aux animations. Cette balise est compatible avec la plupart des navigateurs Web récents. Dans le cas où le navigateur ne supporte pas Canvas, il est possible d'écrire un message d'erreur entre les balises que l'utilisateur pourra voir. L'inconvénient actuel de cette balise est qu'il n'existe pas d'outil graphique ou d'IDE (Environnement de Développement Intégré), ou encore d'éditeur WYSIWYG pour produire le code nécessaire sans créer soit même le code JavaScript. Cela vient du fait qu'il existe bien trop de contraintes technologiques (dépendant de librairies, d'images chargées, d'interactions…).

2.1.2. JavaScript

Figure 3 - Logo JavaScript

Souvent abrégé JS, le JavaScript fut créé en 1995 par Brendan Eich[1]. C'est un langage de programmation interprété, principalement utilisé dans les pages web dynamiques. C'est l'interpréteur du navigateur internet qui interprète le code afin de le faire fonctionner.

Dans le cas de notre application, nous avons utilisé sa possibilité d'être utilisé en tant que langage orienté objet. Un langage de programmation orienté objet s'inspire de ce qui nous entoure. Par exemple, nous pouvons disposer d'un objet voiture, qui est composé de 4 objets roues, 1 objet moteur… une voiture est également un objet véhicule (objet parent). Ces différents objets possèdent des caractéristiques spécifiques. Le langage fournit des objets de base comme des images, des dates, des chaînes de caractères… mais il est possible de créer soi-même des objets. Cela permet de de simplifier notre code, de le rendre plus clair (facile à lire), et d'avoir une manière de programmer plus intuitive.

[1] Brendan Eich étant membre du conseil d'administration de la fondation Mozilla, il a créé le JavaScript pour le compte de Netscape Communications Corporation.

Le JavaScript est un langage dit côté client, c'est-à-dire que les scripts sont exécutés par le navigateur de l'utilisateur. Cela diffère des langages de scripts dits côté serveur qui sont exécutés par le serveur web.

2.2. Plateformes logiciel

2.2.1. Node.js

Node.js a été créé par Ryan Lienhart Dahl[2] en 2009. C'est une plate-forme logicielle construite sur l'exécution JavaScript de Chrome (avec le moteur V8) pour construire facilement des applications de réseau rapides et évolutives.

Node.js utilise un modèle *non-bloquant* (par exemple, quand il demande une donnée à la base de données, le temps de recevoir la réponse, il va exécuter le code suivant et revenir une fois la réponse reçu), *événementiel* (son code n'est composé que de fonctions directement appelées par le client), ce qui le rend léger et efficace. Il est idéal pour les applications en temps réel requérant un grand nombre de données. La nouveauté de Node.js est qu'il rend désormais possible le développement de serveur à l'aide de JavaScript, qui n'est de ce fait plus seulement un simple langage coté client.

Figure 4 - Logo Node.js

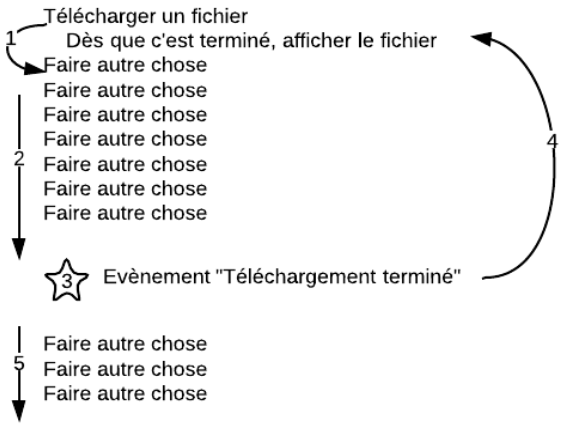


Figure 5 - Schéma programmation asynchrone

Node.js intègre une bibliothèque de serveur HTTP qui permet de faire tourner un serveur web sans avoir besoin de logiciel externe. Cela permet également de contrôler la façon dont le serveur web fonctionne (directement dans le code).

Développer une application avec Node.js est très simple. Premièrement, la création d'un serveur web fonctionnel est simple et ne nécessite que quelques lignes de code (une quinzaine, exemple en annexe 8). Ensuite, il ne reste qu'à configurer les *routes* (c'est à dire définir quelle page est à renvoyer au client selon l'adresse url demandée, exemple en annexe 10), et les éventuelles données à envoyer (exemple de code en annexe 11).

Pour assurer la communication client-serveur, Node.js utilise un système de socket (grâce à la libraire Node.js socket.io). On retrouvera un exemple en annexe 9.

Pour le reste, c'est la puissance du langage JavaScript qui s'exprime, un langage simple, enrichissable par des librairies externes et pouvant être utilisé en tant que langage objet. Node.js peut être enrichi, comme JavaScript, à l'aide de diverses *librairies* (appelées également *packages*), comme par exemple *Node-schedule* qui permet de programmer un événement à une date/heure précise.

[2] Dahl a été inspiré pour créer Node.js après avoir vu une barre de progression de téléchargement de fichiers sur Flickr. Le navigateur ne savait pas combien de fichier ont été téléchargés et a dû interroger le serveur Web.

2.2.2. MongoDB

Figure 6 - Logo MongoDB

Créé en 2009, MongoDB (inspiré du mot "humongous" qui signifie “énorme”) est un système de gestion de base de données NoSQL, orientée documents. Ces derniers contiennent les informations à stocker. Chaque document conserve les informations suivant un tableau associatif (clé => valeur).

MongoDB fonctionne avec un système de collections (équivalents des tables pour MySQL). Ces collections peuvent contenir plusieurs documents. MongoDB est très facilement exploitable avec Node.js par l’intermédiaire de la librairie Mongoose. La création d’une base de données est très rapide et complètement paramétrable.

Le NoSQL qui signifie “Not Only SQL” a été développé de façon à pouvoir gérer de très grands volumes d’informations. Le principe est de limiter les structures relationnelles de façon à pouvoir accélérer la capacité d'accès aux données. Les grands acteurs d’Internet – Google, Amazon, Facebook, Twitter ou Linkedin – conçoivent et exploitent des bases de données de type NoSQL.

2.3. Libraires et Frameworks

2.3.1 Socket.io

Cette librairie, incontournable dans le monde de Node.js, rend possible la communication entre un serveur Node.js et différents clients. Il est très utilisé dans notre application, notamment avec les *socket.emit()* et les *socket.on(),* qu’on retrouvera dans ce document.

2.3.2. CreateJS

Figure 7 - Logo CreateJS

CreateJS est un ensemble de librairies et d’outils pour la création d’interfaces riches et interactives. Ces librairies sont modulables (appelable directement depuis notre code) et peuvent être utilisées ensemble ou séparément selon les besoins des développeurs. Il utilise la technologie HTML5 et est principalement utilisé pour enrichir et améliorer le contenu des canvas. A ce jour il existe déjà bon nombre de modules : EaselJS, TweenJS, SoundJS et PreloadJS, Zoë et ToolKit.

Parmi ces outils, nous avons seulement utilisé EaselJS et PreloadJS. Nous n’avions pas besoin des autres modules car notre application ne contenait pas d’animations à proprement parlé (TweenJS), ni de sons (SoundJS), nous n’avons pas géré les spritesheets comme la plupart des jeux-vidéos (Zoë, découpeur d’animations en spritesheet), et nous n’avons pas utilisé la technologie Flash (ToolKit permet d’adapter Flash pour HTML5).

2.3.2.1. EaselJS


Figure 8 - Logo EaselJS

EaselJS fournit des solutions directes permettant de travailler avec des graphismes riches et interactifs avec le canvas HTML5. Il permet de créer des interfaces familières à celles développées en Flash. EaselJS utilise le JavaScript. Grâce à EaselJS, il est possible de manipuler facilement textes, images, images animées, ombres, filtres, matrices, conteneurs... Il gère également les événements et le rafraîchissement synchronisé.

2.3.2.2. PreloadJS


Figure 9 - Logo PreloadJS

PreloadJS est une bibliothèque permettant de pré-charger du contenu multimédia (des images, des sons, du code JavaScript, des données de texte, etc.). Il permet de charger plusieurs files d'attente simultanément et de gérer les multiples connexions. PreloadJS contrôle la vitesse de chargement du contenu, en choisissant quels éléments charger et à quel moment, afin de rendre le chargement optimal.

Chaque module est présenté sur le site CreateJS, avec différentes démonstrations, une documentation complète et détaillée, ainsi qu'avec des tutoriels faciles à suivre pour démarrer. Le code des démonstrations est fourni, il est donc simple de reprendre ce code et de l'adapter selon le fonctionnement souhaité. La communauté CreateJS est de plus en plus grande, de nombreux forums d'aide au développement existent.

2.3.3. Autres libraires Node.js

EJS	Librairie de template pour les pages Web.
Express	Framework pour application web, gérant les différentes pages
Nodemailer	Librairie permettant l'envoi de mail en utilisant le smtp d'une adresse mail publique (nécessite d'avoir un compte mail ouvert sur le site web propriétaire du smtp)
Node-schedule	Librairie permettant de programmer des événements dans un serveur Node.js

2.3.4. Bootstrap

Twitter Bootstrap, est une collection d'outils utiles à la création de sites web et applications web. C'est un ensemble qui contient des codes HTML et CSS, des formulaires, boutons, et autres éléments interactifs, ainsi que des extensions JavaScript en option. On y obtient, grâce à son utilisation, un site ergonomique et esthétique.

Figure 10 - Logo Bootstrap

2.3.5. jQuery

jQuery est une librairie JavaScript permettant de simplifiant la communication entre le html et le JavaScript sur une page HTML (et donc coté client). Il est très simple à utiliser, et de ce fait est intégré dans 80 % des 10.000 sites web les plus visités sur l'Internet.

Figure 11 - Logo jQuery

2.4. Outils d'aide au développement

2.4.1. Git

Git est un logiciel de gestion de sources, crée par Linus Torvalds[3], permettant un travail collaboratif entre plusieurs développeurs sur les même sources. C'est un petit logiciel, créé à l'origine pour Linux puis adapté à Windows, facile à mettre en place (l'installation d'un serveur Git ne nécessite que quelques commandes sur le serveur de développement, ainsi que pour le client sur nos machines personnelles). Les commandes sont simples à retenir et peu d'entre elles nous sont vraiment utiles. Exemple des plus utilisées:

Figure 12 – Logo Git

- git clone : permet de cloner le contenu d'un serveur Git.
- git commit –a : insérer nos mise à jour dans un « paquet ».
- git status : voir l'état des sources (voir si elles ont été enregistrées dans des commits, combien de commits qui n'ont pas encore été envoyés au serveur).
- git push : envoyer les mises à jour au serveur.
- git pull : télécharger les dernières mises à jour.

[3] Linus Benedict Torvalds est connu pour avoir créé en 1991 le noyau Linux dont il continue à diriger le développement. Il en est considéré comme le « dictateur bienveillant ».

- git clone : première à être utilisée sur nos postes afin de pouvoir cloner le dépôt Git créé sur le serveur.
- git checkout . : réinitialise l'espace de travail au dernier commit effectué ou téléchargé

Et pourquoi spécialement Git ? Il existe d'autres outils de gestion de sources, par exemple SVN. Mais Git était le seul outil de gestion déjà connu par l'un des membres du groupe. De plus, il existe sur le net GitHub, qui permet de créer un dépôt sur un serveur public.

2.4.2. Aptana

Figure 13 - Logo Aptana

Aptana est un outil de développement disposant de multiples outils pour la programmation Web. Il est basé sur Eclipse, logiciel très utilisé pour le développement en langage Java.

2.4.3. Sublime Text 2

Figure 14 – Logo Sublime Text

Sublime Text est un éditeur de texte permettant la coloration syntaxique suivant le langage en cours d'écriture par le développeur, ainsi que l'auto-complétion. Son interface simple et sa coloration efficace en fait un pratique éditeur de code. De plus, il peut être complété par différents plugins (par exemple, il est possible de rajouter un module pour la reconnaissance des différentes fonctions d'une librairie en cours d'utilisation et il propose du code à l'aide de l'auto-complétion).

2.4.4. MongoVUE

Figure 15 - Logo MongoVUE

MongoVUE est un petit logiciel permettant de consulter et d'éditer clairement les données contenues dans une base de données NoSQL. Normalement payant, les fonctions de consultation et d'édition sont malgré tous disponibles dans la version gratuite limitée.

2.5. Autres

2.5.1. Firebug

Figure 16 - Logo FireBug

Firebug est un outil de développement web, sous forme d'extension pour Mozilla Firefox qui permet de déboguer, modifier et contrôler le HTML, le CSS, le JavaScript et bien d'autres langages d'une page web.

Ce module a surtout été utilisé pour indiquer les erreurs programmation dans le canvas. Cet outil est simple et efficace, il indique la ligne où se situe l'erreur dans le code, il est ainsi plus simple et plus rapide de les corriger.

2.5.2. Putty

Figure 17 - Logo Putty

Putty est un logiciel permettant de se connecter à un serveur distant en SSH ou Telnet. Il surtout destiné aux plateformes Windows, ne disposant pas de commandes ssh nativement. Une fois connecté, ce petit logiciel émule un terminal permettant d'administration à distance le serveur.

2.5.3. Outils graphiques

Concernant la partie graphique du canvas, nous avons utilisé différents logiciels, notamment pour dessiner la carte du plateau de jeu, mais également pour éditer des images.

2.5.3.1. Tiled

Tiled (qui signifie en anglais “carrelé” ou “en mosaïque”) est un outil gratuit, développé par Thorbjørn Lindeijer et son équipe, qui permet de créer facilement une carte de jeu 2D, sous forme de grille

Figure 18 - Logo Tiled

2.5.3.2. Snagit

Afin de prendre les captures d’écran de la carte dessinée sous Tiled, nous avons utilisé Snagit, un outil simple et rapide, développé par TechSmith.

Figure 20 - Logo Snagit

2.5.3.3. PhotoShop

Photoshop est un logiciel de retouche et traitement d’images ainsi que de dessin assisté par ordinateur édité par Adobe. Il est essentiellement utilisé pour le traitement de photographies numériques.

Figure 19 - Logo Photoshop

2.5.3.4. PhotoFiltre

PhotoFiltre est un logiciel de retouche et de traitement d’image. Son utilisation est plus simple que Photoshop, car elle nécessite moins de connaissances, et son interface est plus ergonomique.

Figure 21 - Logo PhotoFiltre

2.5.3.5. SpriteCutter

SpriteCutter est un outil facilitant le découpage automatique des sprites, ainsi que leur assemblage. SpriteCutter dispose même d’un testeur d’animations.

2.5.4. Bilan

Au final, nous avons utilisé bon nombre de nouvelles technologies, que ce soit langages, plateformes, librairies ou logiciels. Nous pensons que ces outils sont vraiment en devenir et que certains s’imposeront comme des standards dans les années à venir concernant le développement web, comme Node.js et MongoDB.

3. Travail réalisé

Tout notre travail au long du projet a suivi différentes phases, basé sur le cycle de développement en cascade, qui est détaillé dans chapitre 5, traitant la gestion de projet. Le travail réalisé durant ce projet est donc décomposé comme tel.

3.1. Analyse - Fonctionnalités

La définition des fonctionnalités est souvent une partie délicate dans un projet logiciel, car la maîtrise d’oeuvre qui réalise le logiciel, doit réussir à bien comprendre les besoins du client, souvent flous, et aboutissent quelques fois à un désaccord une fois le logiciel terminé.

Pour notre projet, nous n’avons pas eu ce problème car, comme mentionné dans la présentation du projet, nous avons proposé notre propre sujet, c’était donc à nous de définir les fonctionnalités que nous souhaitions.
Dans un premier temps, comme notre application était un jeu, il nous a fallu définir quel type de jeu, et les règles qui allaient suivre.

3.1.1. Jeu

Pour le nombre de dimensions graphiques, nous ne pouvions pas faire mieux que la 2D, car la 3D pour une application Web n’est pas optimale : trop gourmande en bande passante. Ensuite, nous nous sommes inspirés des

différents jeux par navigateurs, que nous pouvons classer en deux types:

- Les jeux dits “flash”, comme par exemple “Angry Birds”, qui sont dynamiques, et où tout le jeu est contenu dans une partie du site web. Ceux-ci sont souvent jouables qu’à un joueur.

- Les jeux dits “par navigateur”, comme par exemple “Ogame” ou “Hordes”, qui sont souvent multijoueurs et persistants: les joueurs, qui peuvent interagir entre eux, évoluent dans un monde qui continue à fonctionner même quand ils sont déconnectés. La plupart du temps, rien n’est dynamique : les actions se font à l’aide de boutons placés sur le site. De plus, le jeu se fait sur plusieurs pages, plusieurs onglets.

Nous avons donc opté pour un style dit “par navigateur”, mais en modifiant ce type de jeu, car, dans notre projet, toute la partie “jeu” sera définie sur un seul et unique plateau regroupant tous les boutons d’actions et d’affichages (un peu comme les jeux “flash”).

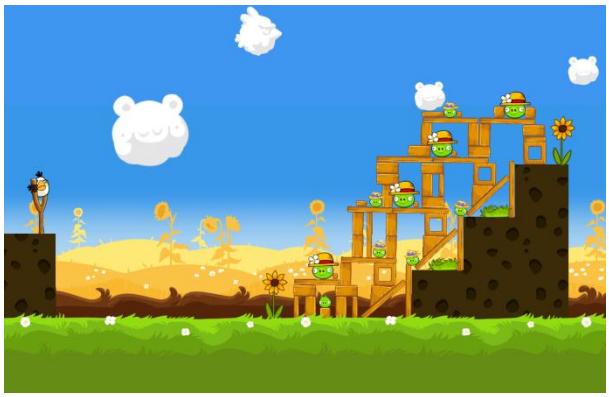


Figure 23 – Image tirée du jeu Angry Birds

Figure 22 – Image tirée du jeu Ogame

Ensuite, il nous a fallu réfléchir au style du jeu: comment jouer, qu’est-ce qu’on pourra y faire, quel en sera le but… Une partie intéressante dans laquelle nous avons modelé notre futur jeu à notre guise. Beaucoup de choses pouvaient être imaginées, mais nous devions rester réalistes sur le nombre de fonctionnalités, afin que le projet reste réalisable et terminé en cinq mois. En même temps que cette phase d’analyse, nous avons également navigué sur des sites web du même type que l’application que nous voulions concevoir afin de voir ce qu’il se faisait.

3.1.2. Site web

Pour jouer, chaque joueur à besoin d’un compte, il faut donc envisager une gestion des comptes, où chaque joueur s’inscrira, choisira un pseudo ainsi qu’un mot de passe et devra fournir une adresse email.
Mais le projet ne se résume pas en un simple jeu. En effet, il y a des fonctionnalités qui sont toujours appréciés par les joueurs, comme les classements (qui motivent à bien jouer) et les fonctionnalités pour communiquer avec les autres joueurs. Au-delà de ça, une zone administration est indispensable afin de pouvoir gérer simplement le jeu, et définir le moins de chose possibles en dur dans le code (par exemple, les dégâts infligés par les zombies).

3.1.2.1. Gestion des comptes

Pour jouer et consulter le classement, chaque joueur devra être inscrit et connecté, ce qui semble normal. Le jeu se déroulant dans une modélisation 2D de l’IstiA, nous avons choisi de restreindre uniquement au personnel ou étudiants de l’université d’Angers. C’est pour cela que nous avons mis en place un système de vérification d’adresse email. Afin de garantir que chaque joueur n’ait qu’un seul compte, une validation par mail sera demandé au joueur afin qu’il active son compte.

3.1.2.2. Tchat(s)

Afin d'assurer la communication entre les joueurs, nous avons décidé de créer un tchat pour chaque équipe, afin que les joueurs élaborent leurs stratégies. De plus, un autre tchat, entre tous les joueurs, nous paraissait intéressant.

3.1.2.3. Classement

Un classement des joueurs a été créé afin que chacun fasse de son mieux pour atteindre le haut du tableau.

3.1.2.4. Admin

D'ici, les administrateurs peuvent supprimer des comptes, lancer une nouvelle session de jeu (réinitialisation des cases, personnages et scores).

La fin de cette phase s'est soldée par la rédaction d'un mini-cahier des charges, validé par notre encadrant. Ce document définissait ce qui devait être fait, nous le retrouverons en annexe de ce rapport.

3.2. Conception

A partir du mini cahier des charges, nous en avions déduits plusieurs objets (par exemple des objets Item, Personnage, Utilisateur) et construit une architecture solide en plusieurs couches.
On peut alors découper l'application en trois parties :

- Le client, contenant le site web et le plateau de jeu. Ce ne sont que des interfaces, qui communiquent avec le serveur et affichent les données. Elle offre à l'utilisateur une interface pour qu'il puisse agir sur ces données.
- Le serveur, qui traite les données.
- La base de données, qui stocke les données.

On se retrouve avec une application de type client <-> serveur, très utilisé dans le web, où généralement le client demande une information, le serveur traite la demande et renvoie l'information. Cependant, nous allons modifier cela, car le client, une fois authentifié auprès du serveur, pourra être contacté n'importe quand par le serveur, tant que la connexion est ouverte.

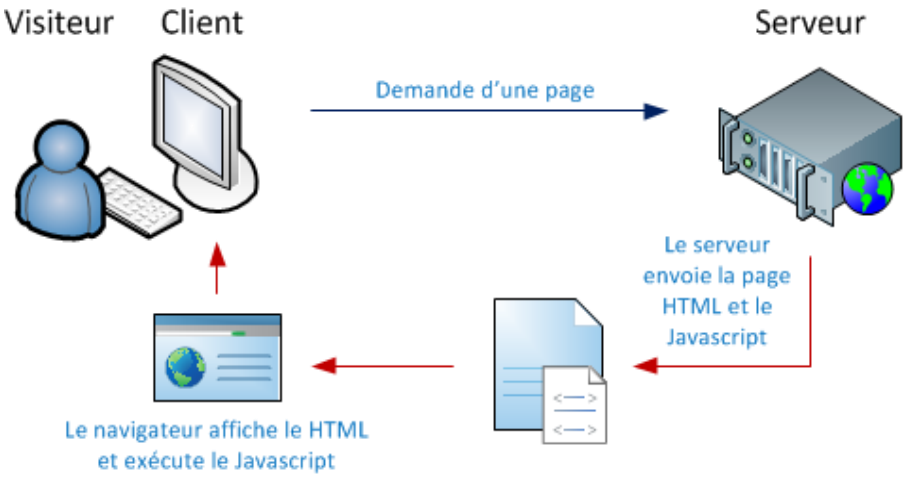


Figure 24 - Communication client-serveur JavaScript

3.2.1. Site web

Pour cette partie, nous nous sommes au début tourné vers un éditeur WYSIWYG (Adobe DreamWaver), mais, vu la complexité de prise en main et la simplicité prévue de notre site, nous avons décidé de rester à la création manuelle, en écrivant les balises HTML nous-même, à l'aide d'un éditeur de texte tel que Notepad++.

3.2.2. Plateau de jeu

Qu'est-ce que le plateau de jeu ? Comme son nom l'indique, c'est une interface permettant à l'utilisateur de jouer. Sur le plateau de jeu, l'utilisateur pourra voir toutes les informations concernant l'état de son personnage, les informations de la case dans laquelle il se trouve, ainsi que les différentes actions qu'il peut effectuer. Au commencement du projet, nous avons longuement hésité sur le choix des librairies à utiliser pour le design du canvas. Après quelques recherches, deux possibilités étaient retenues : Canvas Engine ou CreateJS.

Suite à une série de tests utilisant séparément les deux technologies, nous nous sommes rendus compte que CreateJS était plus simple d'utilisation et mieux adapté à nos besoins. Nous avons également observé que la documentation fournie par CreateJS était plus complète et plus détaillée que celle de Canvas Engine. De plus la communauté CreateJS est plus grande que celle utilisant Canvas Engine, ce qui permet de trouver plus facilement des solutions à nos problèmes, notamment grâce aux discussions sur les forums. C'est pour cela que nous avons choisi d'utiliser CreateJS.

L'organisation des différents éléments a été définie selon une maquette réalisée après mise en commun de nos idées. Dès le début, nous avions en tête de réaliser un plateau de jeu en 16:9, afin qu'il ait un aspect moderne et dynamique.

Nous avons, sur la première version, regroupé les éléments selon leurs relations avec le personnage ou la case. Sur la seconde version, nous les avons regroupé par type, c'est à dire les barres d'état du personnage d'un côté, les boutons de l'autre, les informations de la case dans autre coin, et les objets dans un autre coin.

3.2.3. Architecture coté serveur

Créer une architecture permet de séparer le code (une partie définissant les différentes entités de l'application, une partie traitant les données…) et donc rendre le développement bien plus facile et surtout maintenançable, car on sait mieux où effectuer une modification pour engendrer un résultat. De plus, cela peut servir pour séparer le travail de chacun, car, par exemple, quelqu'un travaillant sur les vues ne touchera pas le code de celui qui travaille sur les traitements, et inversement.

3.2.3.1. Définition des classes

JavaScript est un langage qui peut être utilisé comme langage objet. C'est alors ce que nous avons choisi de faire, et la première étape de la conception de l'architecture passa par la définition des différents objets de l'application. A partir des éléments de l'analyse, nous avons conçu un diagramme d'objet. On peut le retrouver dans la figurez ci-dessous

Remarque : Une case étant le modèle “informatique” d’une salle, les mots “case” et “salle” dans la suite de ce document auront alors le même sens.

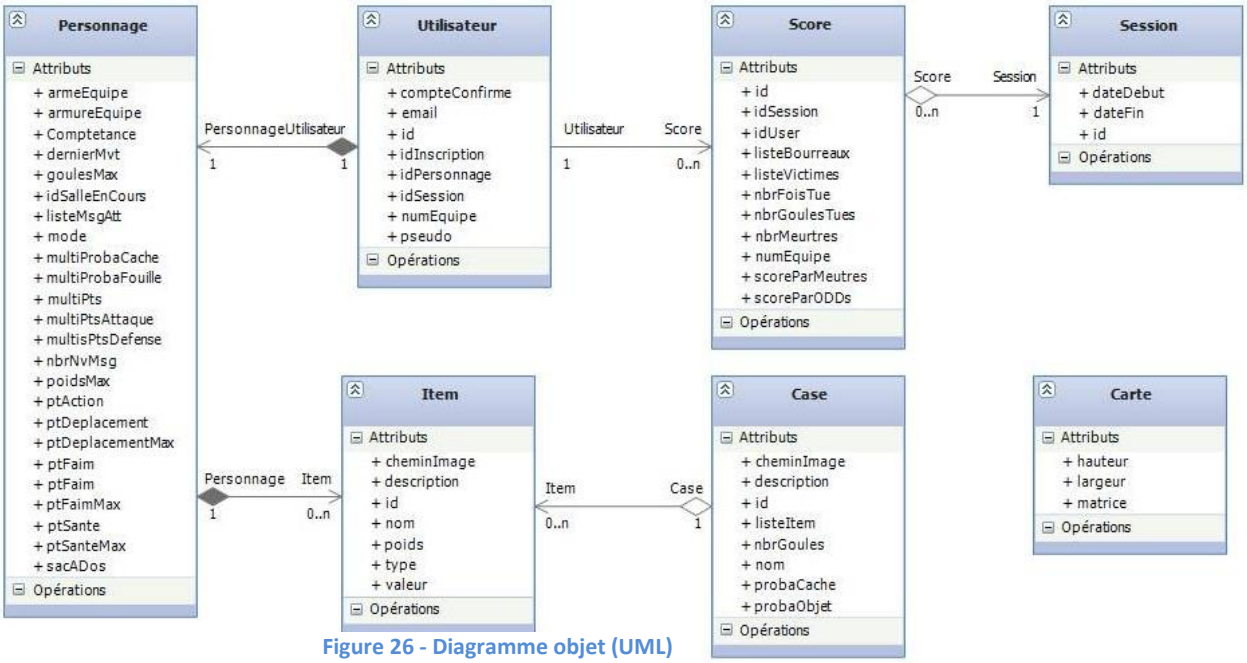


Figure 26 - Diagramme objet (UML)

Ces objets sont la base de l’application, tout tourne autour de leur gestion.

Utilisateur	Représente l’utilisateur physique : pseudo, mot de passe etc...
Personnage	Représente le personnage virtuel de l’utilisateur : points de vie, contenu du sac, compétence...
Case	Représente une salle du bâtiment : nom, description, probabilité de découvrir un objet…
Item	Représente un objet utilisable par un personnage : arme, potion...
Session	Représente une “partie” de jeu. A chaque nouvelle session, les cases, les personnages et les scores sont réinitialisés.
Score	Représente simplement les scores d’un utilisateur pour une session donnée.
Carte	Matrice représentant la position des différentes cases de la carte de jeu

Figure 27 - Tableau des classes

3.2.3.2. Définition des couches

La conception d’une application en plusieurs couches permet de bien séparer le code selon sa “fonction” : traitement, objet, interfaces graphiques, gestion de la base de données. De cet avantage, nous avons décidé de structurer notre application de cette façon, que l’on retrouve sur la figure 29. Le sens des flèches indique le sens de la communication. Par exemple, la couche coordination avait accès aux classes de la couche managers.
Pour le schéma plus détaillé, voir l’annexe 1.

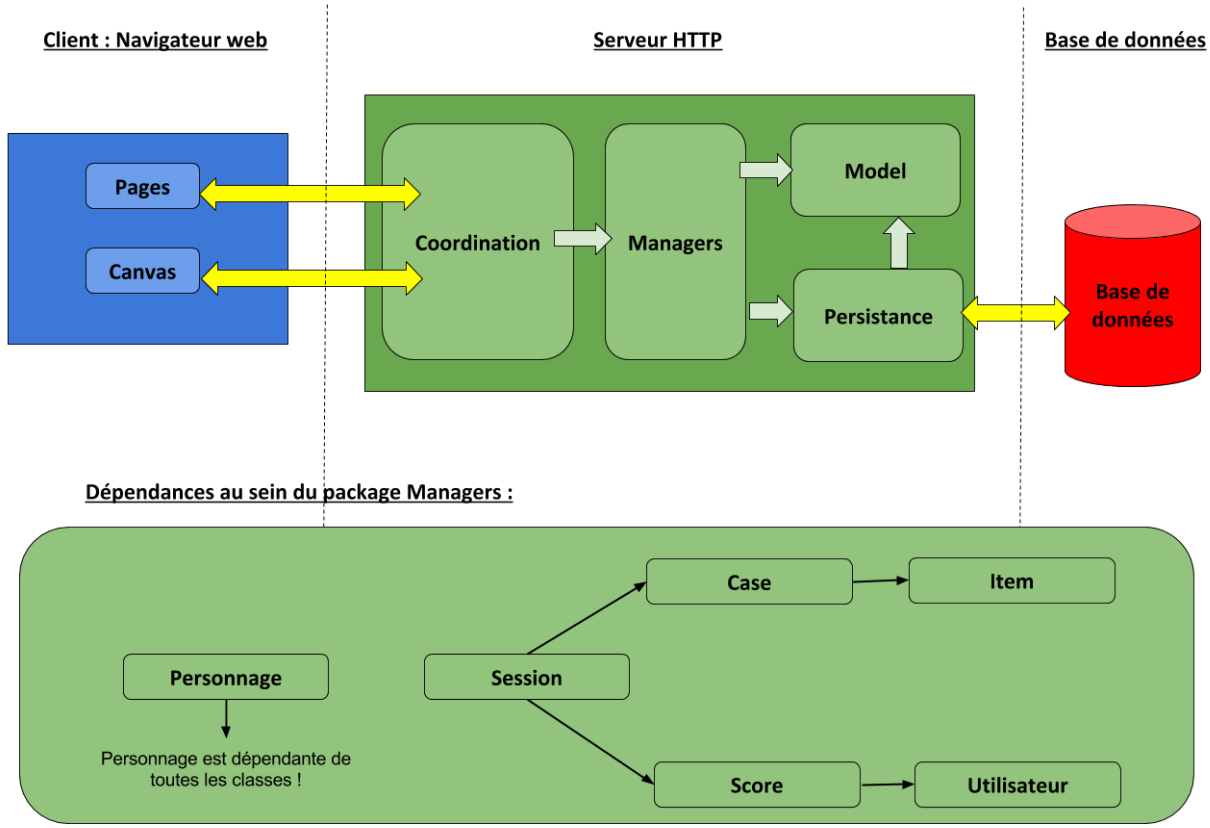


Figure 28 - Schéma de la conception en couche du serveur

Model (= objets)

Des objets "cœurs"

En effet, toute l'application repose sur ces objets. L'application n'est qu'une gigantesque interaction entre les différentes instances de ces objets.

Des contrôleurs

Ces objets *peuvent* également effectuer des contrôles. Au final, les contrôles peuvent être placés soit dans le manager, soit dans le model. Cela peut être très discutable :

- Mettre les contrôles dans les models peut être contraignant si l'objet est utilisé dans une autre application, car il faudrait changer les contrôles et en conséquence toucher à l'objet. De plus, le manager perd de son importance.
- Mettre les contrôles dans le manager peut être une très bonne idée, mais on perd les contrôleurs si l'objet est utilisé autre part dans l'application. L'objet devient moins "solide".

Persistance : La mémoire de l'application

Une interface serveur <-> base de données

Cette couche reçoit ses ordres du manager, et communique avec la base de données afin d'enregistrer, d'ajouter ou de lire des informations, qu'il renverra à ce dernier. Elle fait office d'ORM (Object Relationnal Mapping), car elle sait passer d'un objet de type JavaScript du serveur à un objet de type MongoDB de la base de données, et inversement.

Managers : La gestion des objets

Un gestionnaire

Comme son nom l'indique, cette couche gère les objets (chaque manager gère un type d'objet). Par exemple, le manager du personnage (objet de type "Personnage") de l'utilisateur contient en attribut une liste de tous les objets personnages. Il peut de ce fait directement agir dessus selon les ordres donnés par la coordination.

Un contrôleur

Être manager est une responsabilité, il doit dans ce cas effectuer quelques contrôles avant de donner un ordre à l'objet qu'il gère. Par exemple, en cas d'attaque d'un joueur sur un autre, il doit vérifier que les deux personnages ne soient pas de la même équipe.

Une base de données virtuelle

Chaque manager contiendra une liste des objets de la base de données. Au lancement du serveur, chaque manager va charger en mémoire, dans sa liste, les objets qu'il gère (par exemple, la classe Personnage_Manager va charger dans sa liste tous les personnages contenus en base de données. Chaque modification d'une donnée se fera dans cette liste, et non dans la base de données. Les informations seront sauvegardées en base de données seulement sur ordre de la couche *coordination.*

Chaque manager implémente le pattern Singleton, qui lui permet de n'avoir qu'une seule instance de l'objet. L'instance de chaque manager est connue par le serveur afin qu'il puisse communiquer avec.

Il faut noter que, au début, nous étions parti sur un modèle en 4 couches, la couche manager n'existait pas encore.

Coordination *:* Une interface client - serveur

Cette couche est composée d'un seul et unique fichier, *app.js*, qui fait office d'interface entre la vue (contenant les pages et le canvas) et le serveur. En effet, lorsque la vue a besoin de communiquer avec le serveur, les "*socket.emit()*" ou les "*get* / *push*" (voir annexe 11 sur la communication client <-> serveur) sont traités sur ce fichier. A l'inverse, lorsque le serveur doit communiquer avec la vue, pour lui répondre ou l'informer de la modification d'une donnée, c'est ce fichier qui se charge de la communication.

Il détient aucun algorithme de fonctionnement, et ne fait aucun traitement et peu de contrôles. Il laisse les managers s'en charger. Dès qu'il reçoit le code de retour de la part des managers, il renvoi ce code.

Mais, au-delà de ça, le fichier app.js est le pilier du serveur, car c'est le *point d'entrée* de l'application. C'est à dire que les premières lignes de codes à exécuter seront les premières lignes de ce fichier.

Pour résumer, il remplit 5 fonctions :

1. Configuration du serveur : port d'écoute, routes.
2. Chargement en mémoire des données : Il initialise les managers et, dans cette initialisation, les managers récupèrent à partir de la base de données les données les concernant. Par exemple, le manager des utilisateurs va charger dans une liste à lui tous les utilisateurs. Il faut bien faire attention aux dépendances lors de l'initialisation des managers; comme par exemple, pour charger les cases, il faut que les items soient chargés en mémoires, donc le manager des cases doit se lancer dès que le manager des items a fini de s'initialiser.
3. Reçoit les données du client, et lui en envoie.
4. S'occupe du rafraîchissement des interfaces utilisateurs, afin que chacun ait une interface actualisée en temps réel
5. Gère les attaques et les sauvegarde globales[4] en base de données tous les jours : la programmation se fait dans ce fichier.

[4] Evènement déclenché chaque jour par le serveur, qui demande l'enregistrement des données en mémoire vers la base de données. Voir partie 4.4.3 et 4.4.5 pour plus de détails

En l'état, cette conception était loin d'être parfaite : en jeunes développeurs que nous sommes, beaucoup d'éléments nous avaient échappés, comme la gestion des scores, des déplacements. Ils ne seront pas détaillés mais tout au long du développement, les objets ont évolués et ont vu leurs nombre d'attributs augmenter. Les sauvegardes en base de données ne se feront pas à partir de la couche model mais à partir de celle des managers. Des erreurs s'étaient glissées dans la conception, mais nous les avons comprises.

Cette période nous a également permis de découvrir nos futures technologies de travail, comme le JavaScript et Node.js. Ce fut une période assez courte mais, n'ayant jamais travaillé sur ces technologies, la meilleure façon d'apprendre, de découvrir et de concevoir restait de faire cela à chaud, tout au long du projet. Cette phase de conception nous a au moins permis de fixer un fil rouge.

De plus, pour bien travailler en équipe, il nous a fallu mettre en place un système de centralisation et de "versionning" des sources, afin que tout le monde travail sur la même version du projet. Cela est détaillé dans la partie "Gestion des sources avec GitHub", chapitre 5.3. Nous avons alors crée un dépôt sur ce site web afin que nos sources y soient centralisées.

Cette phase a également servi au choix des outils de développement et technologies, à comparer et étudier ce qu'il se fait. Bien que Node.js (donc JavaScript) nous fût imposé, nous pouvions réaliser le site web comme nous le voulions, avec les outils que nous désirions.

3.3. Développement

Cette partie fera l'objet d'un chapitre entier car, comme mentionné dans la partie 4 de ce document, le développement nous a occupés environ la moitié de notre projet. Donc, dans un souci d'équilibrage de nos parties et de clarté, cette partie sera ne sera pas détaillée ici.

3.4. Tests et améliorations

A ce niveau ci, au 5 Février, notre site web était en version dite "*Alpha*" : la globalité de l'application était développée, il ne restait que quelques détails à finir, la majorité des fonctionnalités étaient réalisées et le jeu jouable. Mais le plus important est qu'il a fallu tester intégralement l'application afin de déceler les éventuels bugs, et vérifier que toutes les spécifications définies lors de la phase d'analyse étaient respectées.

Ne connaissant malheureusement pas les tests unitaires, la résolution d'un bug entraînait parfois la création d'un autre, encore et encore… Beaucoup de temps fut utilisé durant cette phase, mais nous disposions finalement d'une application stable qui respectait (presque complètement, car certaines fonctionnalités nécessitait un peu plus de recherches, comme celle pour déclencher une attaque de nuit chaque jour) le cahier des charges. Et cette partie, originellement prévue sur une semaine, nous a causé bien plus de soucis…

3.4.1. : Tests et optimisations

Pour commencer, il faut savoir que nous n'avions pas implanté de *tests unitaires*[5], et cela s'est ressenti. Après une première batterie de tests, les bugs ont commencés à remonter, et nous les corrigions donc un par un. Le problème fut surtout que lorsque nous corrigions ces bugs, d'autres apparaissaient, car nous modifions sans cesse le code. Et sans les tests unitaires, nous ne nous sommes pas rendu compte que certaines corrections ont également apporté de nouveaux bugs. Et cela a duré assez longtemps. De plus, nous nous sommes aperçus pendant les tests qu'il y avait des problèmes auxquels nous n'avions pas pensé.

[5] Tests implémentés dans le programme qui permet de s'assurer simplement du bon fonctionnement des algorithmes de l'application

Durant la correction des bugs, nous en avons profité pour rajouter de petites améliorations. Mais qui parfois, interféraient avec le code déjà existant, ce qui nous a pris encore du temps pour résoudre ces bugs. Durant les tests, nous avons également optimisé l'application, par exemple en ne chargeant pas en mémoire les personnages des utilisateurs ne participant pas à la session en cours. Cela permettait d'alléger nos structures de données coté serveur. Coté client il y a également eu quelques optimisations, en demandant des informations au serveur plus précises (en bref, pas besoin de demander des informations sur la case en cours, seulement pour obtenir la liste des joueurs de la case). Les demandes furent donc découpées et plus ciblées.

3.4.2. Améliorations

Début Mars, nous préparions la bêta-test (abordée dans le paragraphe 3.6) et apportions toujours plus d'améliorations à notre application. En effet, même si toutes les fonctionnalités du cahier des chargés étaient respectées, nous désirions continuer à faire évoluer notre site, et rajouter des fonctionnalités hors cahier des charges :

- Ajout des scores en cours (équipe / personnage) pour la session en cours.
- Affichage des paramètres de la session pour le joueur.
- Ajout d'un tchat d'équipe rapide sur la page de jeu (branché sur celui de la page prévue à cet effet).
- Ajout de l'envoi d'un mail de confirmation à l'inscription.
- Ajout de sécurité contre les sockets non identifiées. En effet, chaque nouveau client est identifié par le serveur, et une socket est ouverte. Si l'on reçoit une donnée d'une socket non identifié, ce n'est pas du tout normal et donc le serveur devra rejeter ces données.

3.5. Déploiement

Le déploiement s'est effectué début Mars, afin de lancer la période de bêta : il fallait bien entendu que l'application soit accessible de partout dans le monde ! Pour cela, nous disposions d'un ordinateur personnel, tournant sur Ubuntu 12.04, connecté à l'Internet. Cette étape s'est donc faite en plusieurs parties, qu'on retrouvera en annexe 6 "Procédure de déploiement".

Le déploiement s'est donc effectué en une après-midi, et malgré quelques petits soucis (problème de dépendances de packages, que nous avons dû réinstaller manuellement), tout s'est globalement bien déroulé et nous avons pu apprendre comment déployer une application Node.js. L'application était alors en ligne, prête à être utilisée.

3.6. Préparation de la bêta-test

Une bêta-test, rappelons-le, est la phase où l'application est terminée, testée et validée comme stable, prête à être testée par des personnes externes au projet. Souvent, cette phase de test s'effectue "en privée", c'est à dire que le logiciel ou le lien du site de l'application (comme dans notre cas) n'est donné qu'à des personnes sélectionnées. Pour que cette phase de bêta s'effectue au mieux, nous avons :

- Crée un groupe Facebook, afin d'échanger et tenir informer les différents bêta-testeurs très rapidement.
- Crée un formulaire Google Document, afin de récolter des avis et des idées d'amélioration[6].
- Crée des comptes destinés à des personnes hors de l'université.
- Crée un module de log (qui était en fait une classe JavaScript intégré au serveur) qui enregistrait dans un fichier les informations du serveur et dans un autre les erreurs et problèmes détectés. Pour cela, les

[6] Consultable à l'adresse suivante :
https://docs.google.com/forms/d/1GyWFuvVhHDn7yUpnh8o5bjnMn_8KSQm5Fr2Od6jx3YU/viewform?usp=send_form

développeurs du coté serveur avaient juste à insérer la ligne *EventLog.log("Mon log")* pour que "Mon log" soit enregistré dans le fichier. Les erreurs étaient détectées de la même façon, par exemple lorsqu'un test sur une valeur échouait. Grâce à cela, nous pourrions plus facilement débugger les éventuelles erreurs.

Tout était donc prêt pour inviter une vingtaine de personnes à notre bêta, à essayer le fruit de notre travail. Mais, au moment de se lancer, nous nous sommes dit qu'il était trop dommage d'arrêter le développement et attendre les retours.

3.7. Version 2.0 de l'interface

Après une prise de recul sur notre travail, et en estimant le temps restant suffisant (3 semaines), nous avons pris la décision d'améliorer les interfaces, c'est-à-dire refondre complètement le site web et le plateau de jeu. Ces changements s'inscrivent dans une idée d'amélioration continue et ont rendu l'application plus ergonomique. Nous voilà donc replongés dans le développement, à 3 semaines de la fin. La construction et la mise en œuvre de cette deuxième version se retrouvera donc dans les chapitre 4.2.2 pour le site web et 4.3.2 pour le plateau de jeu.

3.8 Bilan du travail réalisé

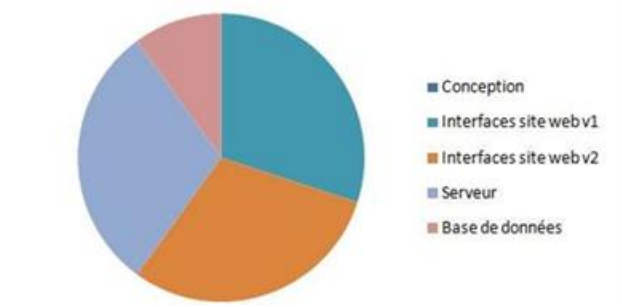


Figure 30 - Répartition du travail d'Abdelmounaim

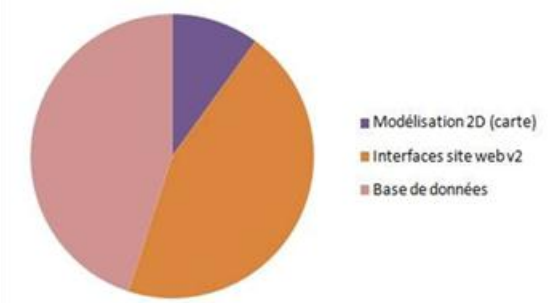


Figure 29 - Répartition du travail de Brendan

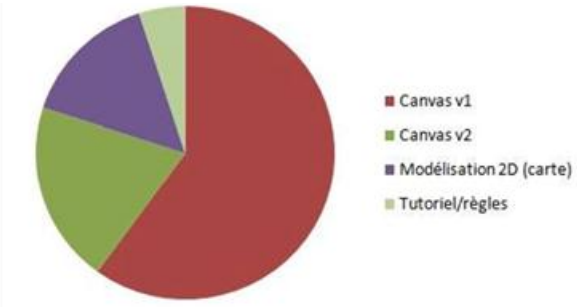


Figure 32 - Répartition du travail de Florian

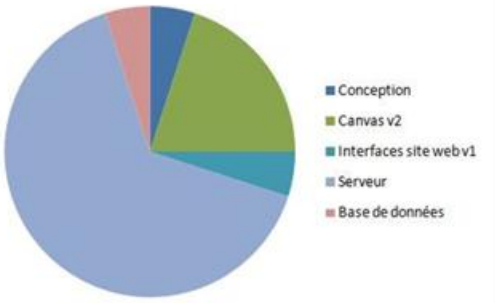


Figure 31 - Répartition du temps de travail de Johan

Notre objectif de départ est réalisé, à savoir créer un jeu, sur un site web, qui est accessible par n'importe quel utilisateur connecté à l'Internet et disposant d'un navigateur Web. Le cahier des charges initialement rédigé est complètement respecté, avec des fonctionnalités supplémentaires.

4. Développement

Avant d'aborder cette partie, il serait bon de rappeler deux points :

Premièrement, rappelons l'architecture globale donnée en 3.2.3 et expliquons la communication client-serveur. L'application est donc décomposée en trois parties bien distinctes : le client, le serveur et la base de données. Pour la communication serveur <-> base de données, nous n'avions pas à nous en occuper, le package *mongoose* le faisait pour nous. En revanche, pour la communication client <-> serveur, c'était à nous de tout mettre en place.

Et deuxièmement, n'oublions pas que le développement a repris en fin de projet avec la création d'une version 2.0, qui fut une refonte graphique du site web et du plateau de jeu. Les deux versions seront alors développées dans ce chapitre.

4.1. Communication client <-> serveur

Conformément à ce que Node.js impose, nous avions utilisé les *socket.emit()* qui permettaient d'envoyer des données du client au serveur (et inversement), et les *socket.on()*, évènements qui étaient exécutés à la réception de la socket.

Pour ce faire, nous avons décomposé les demandes clients et réceptions des données en différentes types, suffixé par le sens de la communication (par exemple, un suffixe "_SC " signifie une communication serveur -> client). Quelques exemples :

- INFO_CASE_CS, pour la demande d'informations sur la case.
- INFO_CASE_SC, pour l'envoi d'informations sur la case vers le client.
- INV_PERSONNAGE_CS, pour la modification de l'inventaire du personnage (qui contiendra donc d'autres arguments, comme le type de la modification ("équiper" ou "déséquiper"), ainsi que l'identifiant de l'item concerné.
- INV_PERSONNAGE_SC, qui renvoi la réponse au serveur (si l'action a réussi ou non).

4.2. Site web

4.2.1. Version 1.0 : Html - CSS

La première étape du développement de la version 1.0 du site, a été la recherche du thème des pages. Nous avons donc trouvé un thème dérivé de Bootstrap qui nous satisfaisait et correspondait à l'idée que nous nous faisions du site.

Celui-ci une fois déterminé, nous sommes passé au développement à proprement parlé. La première page a été l'accueil. Il nous fallait donc plusieurs modules. Un pour la connexion, que nous avons placé dans le bandeau supérieur de la page. Cette partie contenait également les onglets de redirection vers les différentes pages du site. Le second, pour l'inscription, qui fut placé tel un widget sur le côté droit de page afin d'être visible et simple d'utilisation. Ces deux modules utilisent tous les deux des formulaires (balises <form></form>), qui utilisaient une méthode POST et PUT. Sur cette même page, un troisième module était utilisé pour afficher les news liées au jeu une fois celui-ci lancé.

Lorsque que l'on se connectait, la page d'accueil était quelque peu modifiée. En effet, les deux modules, de connexion et d'inscription, n'avait plus lieu d'être et devaient donc être remplacés ou enlevés. Le premier d'entre eux a tout simplement été remplacé par un message d'accueil suivi d'un un formulaire de déconnexion avec une méthode DELETE. Le second quant à lui a tout simplement été remplacé par une image d'accueil qui elle-même a été supprimée plus tard pour des soucis de clarté.

Figure 33 - Version 1.0 du site web

Page de jeu

Une fois partis sur le même modèle, nous nous sommes attelés au développement des autres pages. Sur la page de Jeu, il s'agissait dans un premier temps d'intégrer le canvas du jeu qui un peu plus tard sera accompagné d'informations relatives sur la session en cours ainsi qu'un tchat par équipe rapide (plus petit que celui présent sur la page conçue à cet effet).

Pages de tutoriel / règles

Sur les pages de règles du jeu et de tutoriel, l'objectif était de profiter de la zone centrale et de l'agencement en colonnes de notre site pour donner à ces informations un visuel ludique et original en alternant des petits blocs d'informations à gauche et à droite. Ces pages sont les seules accessibles par l'utilisateur lorsqu'il n'est pas connecté.

Pages de tchats

Les différents tchats devaient permettre d'une part de faciliter les joueurs à élaborer leur stratégies de jeu mais aussi et surtout de se rejoindre afin de discuter/faire connaissance. Ils se devaient donc d'être simples d'utilisation et aussi clairs que possible. C'est donc pour cela que nous avons opté pour le strict minimum, c'est-à-dire: un champ de texte, un bouton d'envoi et une zone de texte en lecture seule pour l'affichage des messages du tchat. Tout ceci était accompagné d'une liste dynamique (rafraîchissement en temps réel) dans laquelle l'utilisateur pouvait savoir qui était actuellement connecté au tchat. Le fonctionnement du tchat est entièrement régi par du code JavaScript/jQuery afin d'être aussi dynamique que possible.

Page de classement

Sur la page de classement, le fonctionnement est quelque peu similaire. La page charge seulement l'en-tête du tableau des scores qui sont alors renseignés par un script en JavaScript, après une demande auprès du serveur. Nous avons choisi cette méthode afin de gérer au mieux les fonctions de tri que l'utilisateur pouvait appeler en cliquant sur les différentes catégories (Joueurs, Nombre de personnes que le joueur a gravement blessé, Nombre de fois que le joueur a été gravement blessé, Nombre de zombies tués, Scores par blessures, Scores par objets de défense ramenés).

Page d'administration

La dernière des pages et non la moins utile, était une page seulement accessible par nous, car il s'agissait de la page d'administration. Cette dernière faisait une requête sur le serveur pour avoir accès à un tableau contenant tous les utilisateurs du site. Une fois ceci reçu, nous pouvions l'afficher sur la page et en sélectionner un utilisateur afin de le supprimer de la base de données. Une deuxième partie de cette page était celle de lancement, arrêt ou changement de date de fin de session. Il s'agissait des trois boutons correspondants et de listes déroulantes pour les choix du jour, du mois et de l'année si on cherchait à lancer une nouvelle session ou changer la date de la session en cours. Un dernier bouton était présent sur la page, permettant de créer de nouveaux objets aléatoires dans les différentes salles du jeu.

Toutes les pages ont tout de même un point commun très important, elles ont une extension .ejs (utilisé par notre moteur de template EJS) qui permet d'utiliser du code entre les balises <% ... %> qui nous était extrêmement utile pour faire passer des informations du serveur aux pages sans que le client ne les demande.
Pour gérer ces pages (ainsi que le système de session, permettant à l'utilisateur de naviguer sur les différentes pages en restant connecté), nous avons choisi d'utiliser le Framework *Express,* le plus fiable et le plus utilisé dans le monde Node.js. On retrouvera sa configuration en annexe 12.

4.2.2. Version 2.0 : Bootstrap

De multiples raisons nous ont poussées à vouloir faire une deuxième version du site. Ensuite lors de nos tests personnels nous avons constaté que la première version n'était pas très ergonomique en termes de navigation, notamment sur le défilement de la page de jeu qui empêchait l'utilisateur de naviguer de façon fluide. Enfin nous voulions un site totalement personnalisable de façon à pouvoir le modifier selon nos envies.

Nous avons changé l'organisation complète du site pour palier à nos problèmes de navigation. Une barre de navigation a été rajoutée et fixée en haut des pages de façon à pouvoir se déconnecter à tout moment. Celle-ci affiche également l'heure du serveur, ainsi que l'heure de la prochaine attaque de nuit. Puis nous avons ajouté une barre de navigation sur le côté qui suit le contenu de la page courante de façon à ce que l'on puisse accéder à toutes les pages du site peu importe l'endroit où l'on se trouve. Tous les boutons ont été remplacés par des boutons plus sobres de façon à ne pas surcharger le site.

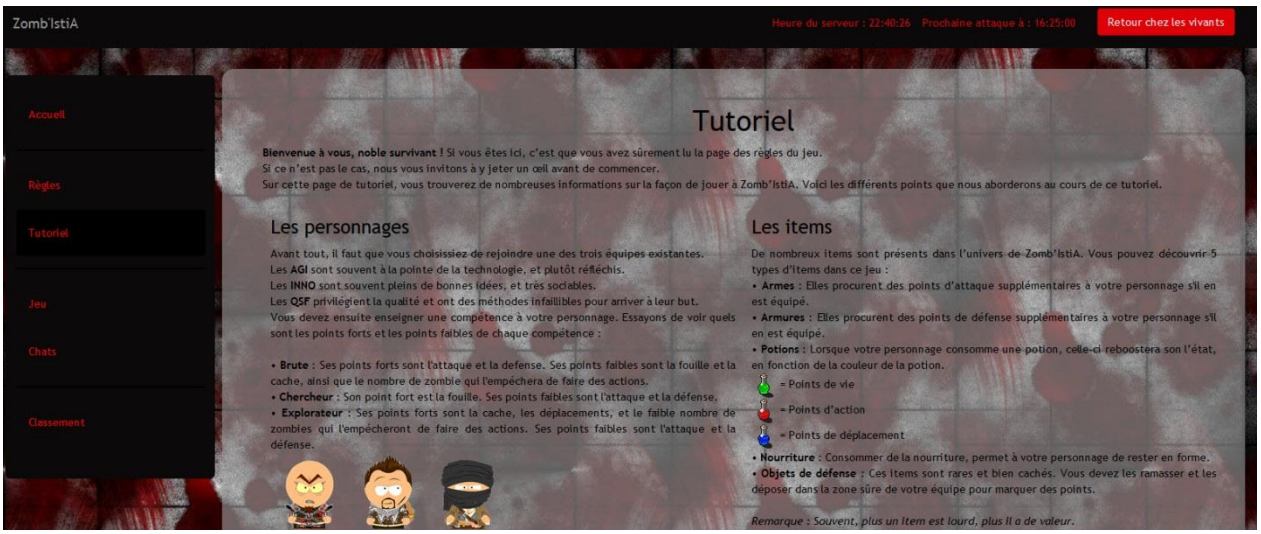


Figure 34 - Version 2.0 du site web

La page de jeu contenait beaucoup trop d'informations pour les afficher d'un bloc c'est pourquoi nous avons choisi de diviser ces informations en trois parties : Plateau de jeu contenant le canvas ainsi qu'un tchat rapide, les scores de la partie et les informations de la session. Pour naviguer entre ces trois pages nous avons ajouté une barre de navigation divisée en trois onglets.

Figure 35 - Onglets de jeu (site web 2.0)

La page de tchat a été modifiée de façon à contenir le tchat de l'équipe et le tchat général. Nous avons voulu changer la page d'accueil de façon à ce qu'elle devienne une page promotionnelle qui pousse un internaute à avoir envie de s'inscrire.

Pour cette nouvelle version nous avons utilisé les librairies Bootstrap pour toutes les barres de navigation et pour le design général du site. Pour la page d'accueil, nous avons utilisé la puissance du CSS de façon à pouvoir afficher des images en fonction du bouton cliqué. De même dans la page de jeu, pour ne pas avoir à recharger le plateau de jeu pour passer aux différents onglets. Nous avons dû personnaliser Bootstrap de façon à pouvoir utiliser les barres de navigation sur le côté qui n'étaient pas définies au préalable.

Pour le développement du module d'inscription, nous avons choisi de le faire apparaître après un clic sur un lien « Rejoignez-Nous » sous la forme d'un pop-up au milieu de l'écran. Il s'agissait d'utiliser une balise avec la classe Bootstrap-modal. Cette classe permettait de faire apparaître le formulaire d'inscription au milieu de l'écran et de donner un effet d'ombre sur le reste de la page.

Figure 36 - Formulaire version 2.0

Cependant, tout ne s'est pas fait sans mal, et les difficultés que nous avons pu rencontrer étaient surtout par rapport aux dates de fin de projet car nous avons choisis de refaire le design du site 3 semaines avant la fin du projet. Il a donc fallu apprendre à se servir de Bootstrap très rapidement.

4.3. Plateau de jeu

Le développement du plateau de jeu s'est faite en plusieurs étapes, et nous l'avons construit brique après briques. Nous avons ensuite amélioré son code ainsi que son apparence.

Remarque : toutes les notions abordées dans cette partie seront illustrées par des exemples de code en annexe 2.

4.3.1. Version 1.0

Afin d'intégrer un canvas dans un site en HTML, il suffit d'y insérer une balise <canvas> </canvas> en précisant le nom et les dimensions de notre interface.

Dans le fichier JavaScript du canvas, il faut récupérer le canvas déclaré dans le fichier HTML, puis créer un stage EaselJS. Le *stage* est une sorte de gros conteneur, qui contient tous les éléments du canvas et qui peut s'actualiser à la demande. Il est désormais possible d'autoriser certains événements sur le *stage*, comme par exemple lorsque le pointeur de la souris survole un élément, ou alors le tactile sur les dispositifs équipés.

Un conteneur (container) est comme une boite, dans laquelle il est possible de positionner plein d'objets différents. Lorsqu'un conteneur est déplacé, son contenu est également déplacé. Cependant, il est aussi possible de positionner à souhait les objets qui s'y trouvent.

Le plateau de jeu étant la vue, le canvas ne faisait que recevoir les ordres de la couche métier (du serveur). Celui-ci nous envoyait des informations via les sockets, tunnels de communication ouverts entre deux couches. Lorsqu'un ordre est reçu, un événement est déclenché. Dans cet événement, nous pouvons alors afficher les données reçues et les traiter localement, côté client. La vue ne fait pas que recevoir des ordres, elle peut aussi envoyer des demandes au serveur, comme par exemple de recevoir des informations ou lui communiquer un choix de l'utilisateur (direction du personnage, objet sur lequel faire l'action, le joueur à attaquer…).

Cependant, il faut noter que cette couche ne fait aucun traitement avant d'envoyer au serveur (pas de vérifications). Par exemple, si quelqu'un décide de ramasser un objet alors qu'il n'a plus de place pour le stocker, ce n'est pas le canvas qui va décider s'il peut ou non. Ce dernier va quand même faire la demande au serveur, qui lui répondra oui ou non. *Pourquoi ?* D'une part, pour respecter la séparation du code mise en place (client : affichage, serveur : traitements et contrôles) et d'autre part, parce que du code coté client peut être attaquable. Il vaut mieux alors, ne prendre aucun risque pour éviter les triches.

Pour la construction graphique du canvas, nous avons utilisé le module EaselJS (décrit dans la partie 2.3.2). Nous rappelons que nous ne disposions pas d'outil WYSIWIG afin de placer les éléments de manière visuelle. C'est pour cela qu'il a fallu placer les éléments manuellement, en modifiant leurs propriétés directement dans le code (position en x, en y, hauteur, largeur…). Cela n'était pas pratique, car il fallait constamment "switcher" entre la programmation dans l'IDE, et la visualisation des changements dans le navigateur, ce qui rend le placement avec précision des éléments assez long. Cependant, en liant les éléments entre eux en leur imposant des contraintes, il devient plus simple de les positionner, notamment pour les aligner et les espacer entre eux.

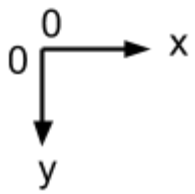


Figure 37 - Système de coordonnées de Canvas

La première chose que nous avons commencé à faire, fut d'afficher du texte (également appelé label) sur le canvas, afin d'afficher les informations envoyées par le serveur. Il est également possible de changer ce texte à tout moment en modifiant sa propriété *text*.

La seconde chose essentielle dans la composition du canvas est la gestion des images. En effet, ce sont les images qui font tout son charme, qui impose l'ambiance et le contexte du jeu. Pour cela, nous avons passé du temps à rechercher des images qui collaient à notre thème et qui s'accordaient entre elles.

Les premières images que nous avons ajoutées étaient le fond du plateau de jeu, et les images des boutons de navigation, ainsi que des boutons d'action. Nous avons utilisé Photoshop pour le design des boutons présents sur le canvas, ainsi que pour créer un effet de brouillard autour de la case en cours, afin de préciser au joueur la case dans laquelle il se trouve. En effet, pour les boutons, nous avons téléchargé un bouton déjà existant, au format PSD. Ce format, propre à Photoshop, n'est pas un format d'image à proprement parlé, il permet de conserver les calques, les couches de transparence… Ce format nous a permis de changer les couleurs et texte du bouton de manière simple et rapide.

Toutes les images que nous avons affiché sur le canvas permettent à l'utilisateur de faire une action, sélectionner un objet ou un personnage, mais également d'afficher des informations correspondantes. Pour cela, il faut brancher des événements à ces images, c'est ce que l'on appelle de la programmation événementielle. C'est-à-dire que le programme sera principalement défini par ses réactions aux différents événements qui peuvent se produire, ici, un mouvement de souris ou un clic.

Nous avons ensuite cherché à afficher les objets du jeu, ceux qui vont être utiles aux personnages et permettant de marquer des points. Pour cela, nous avons pioché un peu partout sur l'internet des images, appelées spritesheets, parfois libres de droits, parfois provenant d'autres jeux. Pour adapter ces images à notre façon de fonctionner, nous avons utilisé SpriteCutter afin de les découper.

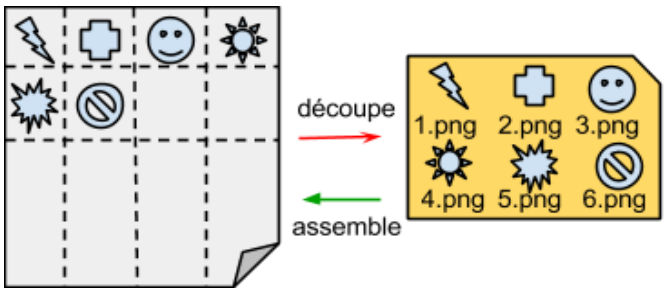


Figure 38 - Schéma de principe de découpage/assemblage de spritesheets

Une fois toutes ces images ajoutées dans leurs conteneurs respectifs, nous nous sommes aperçu que certaines d'entre elles ne s'affichaient qu'une fois sur deux, ou alors pas du tout. Nous avons rapidement trouvé la source du problème, les images ne s'affichaient pas car le navigateur mettait longtemps à les trouver et donc à les afficher. Pour résoudre ce problème, nous avons intégré le module PreloadJS à notre projet. Comme son nom l'indique, ce module nous a permis de charger les images en mémoire du navigateur.

Cela nous a donné l'idée d'ajouter une page de chargement avant l'affichage du plateau de jeu. Cette page de chargement était assez sobre, et contenait seulement deux labels, le premier pour prévenir du chargement, le second affichant le pourcentage de contenu chargé. Trouvant cela un peu vide, nous avons eu l'idée d'afficher une barre de chargement.

De cette page de chargement nous est apparu deux grandes idées. Tout d'abord nous avons décidé de découper notre code en fonctions (hors réception des sockets). Cela a simplifié la gestion des différentes pages sur le canvas, et nous a également permis de simplifier et d'organiser notre code, qui contenait alors beaucoup de tâches répétitives, souvent copiées/collées. La seconde idée fut d'afficher des jauges d'état du personnage, afin de donner les informations au joueur, autrement qu'avec du texte. Il nous suffit alors de reprendre le code de la barre de chargement, et de l'adapter pour nos besoins.

Une fois toutes les fonctionnalités testées, nous avions finit la première version de notre interface.

4.3.2. Version 2.0

Après notre prise de recul, nous avons estimé que la première version était mal organisée. En effet, l'espace occupé par la carte était beaucoup trop grand et les informations et textes occupaient le reste de l'espace. C'est pour ces raisons que nous avons décidé de nous lancer dans une seconde version de l'interface, c'est-à-dire reprendre notre code afin de l'adapter à notre seconde maquette.

Contrairement à la version 2.0 du site web, nous n'avons pas changé de technologie, nous avons simplement repensé la disposition des différents éléments ainsi que la suppression de la majorité des textes (remplacés par de petites images). Comme les éléments graphiques étaient liés entre eux par des fonctions et des variables inter dépendantes, la réorganisation du plateau de jeu s'est faite assez rapidement. Il a cependant fallut créer certains conteneurs, en supprimer, ou alors en redimensionner certains, tels que la zone réservée à l'affichage de la carte.

En effet, dans cette nouvelle interface, nous avons misé sur l'espace et l'affichage dynamique, notamment avec les tooltips[7], c'est-à-dire que lorsque l'utilisateur passe la souris sur les éléments graphiques, une info-bulle vient se superposer sur l'affichage du canvas. Ce nouveau système remplaçait donc la description en dur des éléments, et nous permettait d'aérer le plateau de jeu. Ces info-bulles n'étaient rien d'autre qu'un conteneur, rempli d'une couleur, qui venait se superposer à l'image du canvas. Cependant, afin de ne pas cacher complètement les éléments se trouvant en dessous, nous avons rendu la couleur légèrement transparente. De plus, ces info-bulles donnaient l'impression de suivre la souris lorsque l'utilisateur survolait plusieurs éléments à la suite. Cela a ajouté un aspect plus moderne à notre interface.

Figure 39 - Exemple de tooltip

En plus de ces tooltips, nous avons ajouté différentes animations, selon l'action de l'utilisateur, afin qu'il soit mieux informé du résultat de ces actions (si elles ont été réussies ou non). Par exemple, le nombre de zombie qui vient de l'attaquer durant une action va apparaître en rouge durant un court délai. Ou encore, des animations sur les jauges d'état du personnage, comme lorsque l'utilisateur se fait attaquer, et donc perd de la vie, sa barre de vie (verte) se met un instant en rouge. La réorganisation de l'affichage des items, selon leur type, dans l'inventaire du personnage et dans la case accentue l'ergonomie du jeu.

Nous avons également redessiné ces jauges, en utilisant cette fois-ci non pas EaselJS, mais l'outil graphique de base de canvas, c'est à dire le *context*. La raison est simple : EaselJS ne permettait pas de créer de rectangles à angles arrondis… Le context permet de dessiner directement sur le canvas en utilisant des fonctions prédéfinies. Celui-ci dispose des fonctions nécessaires aux tracés de nos rectangles. Il a fallu penser à redessiner ces barres à chaque fois que l'on mettait à jour le stage avec EaselJS, sinon, le fond du canvas cachait nos rectangles.

[7] Tooltip est un terme anglophone qui signifie info-bulle en français.

Figure 40 - Version 1.0 du plateau de jeu

Figure 41 - Version 2.0 du plateau de jeu

Les deux figures ci-dessus, permettent de mieux visualiser la différence entre les deux versions du plateau de jeu.

4.3.3. Design de la carte

Tiled nous a permis de créer la carte rapidement. Comme nous n'avions pas des compétences de graphistes, nous avons récupéré des spritesheets de textures ainsi que du mobilier et des objets de décoration. Parfois il était plus simple de créer nos propres spritesheets afin de regrouper nos sources d'images.

Sous Tiled, il suffit d'importer une image (spritesheet), de définir la taille des sprites que l'on souhaite utiliser (16x16, 32x32, ou 64x64 pixels). Pour notre projet, nous avons choisi le format 32x32 pixels car la plupart des images que nous avons trouvées étaient dans ce format. C'est surtout sur cette partie que nous avons le plus utilisé les logiciels de retouche d'image, afin de les redimensionner et d'ajouter de la transparence.

Avec Tiled, la transparence des images a son importance car il utilise des systèmes de calques, comme les éditeurs d'images. Ainsi, un premier calque a servi pour le sol et les murs. Un deuxième pour placer par exemple les décorations (tables, chaises…) et un troisième calque pour les détails (objets sur les tables, taches de sang…).

Afin de réaliser la carte à l'échelle de l'ISTIA, nous avons pris des photos des plans de secours du bâtiment, afin de connaître la disposition exacte des salles. Une fois la carte terminée, il nous a fallu définir les cases de notre jeu, car dans une salle, il peut y avoir plusieurs cases (par exemple le hall d'accueil est composé de plusieurs cases). Nous avons découpé les cases ainsi, de manière à ce qu'il n'y ait qu'un seul nord, sud, est et ouest possible pour chaque case. Pour mieux se rendre compte de celà, nous pouvons se référer à l'image se trouvant en annexe 4 ainsi qu'à la matrice en annexe 5.

Grâce à Snagit, nous avons pu prendre des captures d'écran de chaque case, chacune de taille identique. Le nom des images devait être en accord avec ceux définis par la persistance.

4.4. Serveur

Le serveur fut, comme le site web et le plateau de jeu, une grosse partie. Tous les traitements y étaient effectués sur les différents objets, et le développement y fut très long. Les sous parties suivantes décrivent, dans l'ordre, les grandes étapes du développement coté serveur.

Rappelons que les couches vues sont en général dépendantes des couches métiers (ici notre serveur). Cela fait que le développement d'une fonctionnalité coté serveur entraîne l'implémentation de la fonction coté vue. Les développeurs côté vue suivaient donc, en général, les développeurs côté serveur.

On retrouvera en annexe 14 un exemple de traitement à travers la communication entre différentes couches.

4.4.1. Apprentissage et bases du jeu

Dates : début décembre - mi-janvier

Nous partions alors de rien. Pour cette première partie, notre but était d'installer les bases du jeu, c'est à dire une carte (une matrice, où chaque case représentait un objet Case), où les personnages pourraient se déplacer, ramasser/déposer des objets et en consommer ou s'en équiper / déséquiper. De plus, il nous fallait implémenter les trois modes possibles : *fouille* (probabilité de trouver un objet chaque heure), *cache* (les ennemis ne nous voient plus dans la salle) et *défense (attaque et défense boostés),* modes qui restaient actifs tant que le personnage ne bougeait ou n'attaquait pas.

A l'époque, la couche "Managers" n'existait pas, et tous les traitements étaient effectués dans app.js, qui, est, rappelons-le, notre point d'entrée de l'application, ainsi que l'interface client <-> serveur. De plus, nous ne gérions qu'un objet personnage et qu'un objet utilisateur. Le multi-utilisateur n'était pas encore programmé.

Serveur

Pour commencer, il a fallu mettre en place un serveur minimaliste. Pour ce faire, nous avons suivis des tutoriels (site officiel de Node.js, OpenClasserom) afin de produire notre code minimaliste, communiquant avec le client.

Tous les traitements étaient effectués sur ce fichier, et donc toute la logique applicative, c'est-à-dire les algorithmes de traitement. C'est ce dans ce fichier qu'on retrouvait les contrôles sur les données reçues, ainsi que les conditions, par exemple si un joueur pouvait ramasser un objet ou non.

Et enfin, c'est durant cette période que nous avons mis en place tout un système d'inscription de joueurs, de choix d'équipe et de compétence, de connexion et déconnexion, ainsi que les mécanismes de session afin qu'un utilisateur connecté puisse naviguer de page en page. On retrouvera un exemple de session en annexe 13.

Couche model

La communication étant assurée, la première chose à mettre en place était les objets, c'est-à-dire la couche "model", représentant les objets qui seront traités. Pour cela, nous avons étudié la programmation objet avec JavaScript et c'est là que nous nous sommes rendu compte que ce langage est vraiment très... Hybride. En effet, JavaScript n'est pas un langage objet à la base, mais il peut être utilisé comme tel.

Nous avons donc définis nos classes comme construites lors de la phase de conception. Au début de la phase de développement, nous n'avions défini que quatre objets : Utilisateur, Personnage, Item, Case, ainsi que une classe Carte.js, classe implémentant le pattern *Singleton.*

Couche persistance

Il faut savoir que lorsque le serveur s'arrête, toutes les données qu'il contenait en mémoire sont détruites. Nous avions alors besoin d'une base de données pour stocker durablement ces données.

Pour mettre en place la partie persistance nous avons choisis d'utiliser MongoDB, une base de données NoSQL. Dans un premier temps il fallait intégrer la base de donné à notre projet, pour ça, nous avons utilisé le package Mongoose qui permet la communication entre le serveur MongoDB avec le serveur Node.js. Pour fonctionner, Mongoose a besoin de schémas c'est pourquoi nous avons créé un modèle pour notre base de données qui permet de créer tous les schémas (Utilisateur, Personnages, Item, Case, Session) au moment de l'initialisation de la base. Ensuite nous avions besoin de contrôler l'enregistrement des données. Donc pour chacun des schémas précédents nous avons créé des classes définissants les actions que nous avions à traiter, par exemple l'enregistrement d'un nouveau personnage ou la mise à jour d'un utilisateur.

Le premier problème auquel nous avons dû faire face dans cette partie fût le fonctionnement asynchrone de Node.js et de Mongoose, car étant habitué à la programmation synchrone nous avons eu du mal à nous familiariser avec l'utilisation des callbacks. Pour illustrer ce problème prenons le cas où un utilisateur veut s'inscrire sur le site, il doit passer par la partie client puis les informations arrivent au serveur et sont traitées. Au moment d'enregistrer l'utilisateur nous faisions des tests qui dépendaient de l'enregistrement de l'utilisateur dans la base dans la fonction d'enregistrement au lieu de le faire dans une fonction callback.

Le second problème fût l'initialisation des parties item et case car il ne fallait pas remettre à zéro la matrice case si le serveur crash. Pour pallier à ce défaut nous avons choisi d'utiliser un fichier texte pour initialiser les items et les cases car c'était ce qu'il y avait de plus pratique pour éviter de retoucher au code si l'on voulait modifier le nom d'un item ou la description d'une case.

Une fois ces problèmes résolus, nous avons décidé de créer une classe pour gérer la persistance de chaque objet. Ainsi, nous avons créé les classes Utilisateur_BD, Personnage_BD, Case_BD et Item_BD pour gérer la persistance de leurs objets respectifs.

Difficultés rencontrées

Les difficultés de cette étape étaient simplement liées à l'apprentissage, car nous n'étions pas familiers avec les technologies. La façon de programmer "asynchrone" nous a posé problème, car nous n'avions jamais appris à développer de cette manière. (Ce qui est d'ailleurs étrange, c'est que l'asynchrone soit déstabilisant, car nous, êtres vivants, fonctionnons en asynchrone : nous n'attendions pas la fin d'une tâche pour en commencer une autre, nous savons les gérer en parallèle).

4.4.2. Chargement et sauvegarde des données

Devant le nombre d'objets à gérer (Cases, Personnages, Items), il nous a fallu développer des fonctions afin de manipuler ces objets directement en mémoire, et non plus demander à chaque fois à la base de données. Cela s'est donc fait à partir de app.js, qui demandait les listes des objets aux classes de la persistance et les sauvegardaient dans ces propres listes.

Ensuite, nous nous sommes interrogés sur la fréquence des sauvegardes : fallait-il sauvegarder un objet à chaque fois qu'il est modifié ? Ayant des doutes sur la rapidité de MongoDB à traiter toutes ces données efficacement (imaginons 100 joueurs en train de joueur en même temps…), nous avons décidé de créer ce que l'on a appelé la *sauvegarde globale,* qui est un événement déclenché chaque jour (juste après l'attaque de nuit), et qui a pour but d'enregistrer d'un coup toutes les données dans la base de données.

En plus des données, nous avons créé un fichier nommée *GamesRules.js* afin de stocker toutes les données concernant les règles de jeu : les dégâts infligés par les goules, l'heure de l'attaque de nuit, probabilités maximales de trouver un objet etc…

Une fois cette partie complétée, nous étions prêt pour aborder une étape importante : le multijoueur !

4.4.3. Place au temps réel et au multijoueur

Dates : mi-janvier - début février

Nous pouvions donc désormais contrôler un personnage, le déplacer dans la carte et interagir avec les objets. Il nous fallut alors ajouter les zombies dans les cases, et mettre en place le multi-utilisateur, afin que plusieurs joueurs puissent jouer en même temps sur le serveur.

Serveur

Pour gérer ce côté multi-utilisateur, il a donc fallu créer des listes contenant les objets : une liste pour charger en mémoire et gérer tous les personnages, et de même pour les utilisateurs, les cases et les items. A partir de ce moment-là, tous les utilisateurs pouvaient être connectés et bouger leurs propres personnages dans les cases.

Ensuite, il a fallu faire en sorte que chaque utilisateur soit averti en temps réel des changements dans la salle : par exemple, quand un allié arrivait dans la salle, il fallait qu'à l'écran, on voit effectivement qu'un nouvel allié est dans la salle. Pour cela, nous avons créé des fonctions permettant d'envoyer les données sur toutes les sockets des utilisateurs. Bien entendu, les données étaient envoyées aux utilisateurs dont leur personnage était dans la même salle.

Puis, nous avons rajouté la gestion des zombies dans chaque case. Rappelons le cahier des charges : à chaque action d'un personnage, il a une probabilité de se faire attaquer par un certain nombre de zombies, et une probabilité plus faible qu'en plus, son action rate. Cela ne fut pas le plus dur, car nous avons mis en place une fonction, toute simple, qui calcule le nombre de zombies attaquants, les dégâts infligés, etc… Après, ce n'était que de l'algorithmie, en veillant à respecter le cahier des charges.

Pour terminer, nous avons dû développer les attaques entre les joueurs, donc ici encore quelques traitements (vérifier que le joueur attaqué n'a pas quitté la salle entre temps, qu'il est bien d'une équipe différente de l'autre). Allant de pair avec ces fonctionnalités, nous avons dû également gérer lorsque la barre de vie du personnage était à zéro.

Couche model

Ajout des fonctions dans les objets permettant de réaliser ces nouvelles fonctionnalités.

Difficultés rencontrées

Nous nous sommes posé quelques questions concernant le rafraîchissement automatique : *quand le faire ? Comment ?* La solution d'envoyer les nouvelles données sur toutes les sockets des utilisateurs nous a semblé être la meilleur, car Node.js à l'air de très bien supporter cela, et d'effectuer cela très rapidement.

Nous avons également rencontré un autre problème assez gênant : le fichier app.js devenait trop imposant, et pouvait être caractérisé par l'anti-pattern de « God-Object» : ce simple fichier contenant tout le code de configuration du serveur, servait d'interface entre le client et le serveur, et réalisait tous les traitements sur les objets. Le fichier devenait ingérable, surtout quand nous étions plusieurs à travailler dessus, car si nous modifions la même fonction, git indiquait un conflit, qu'un développeur devait résoudre.

Et, pour boucler le tout, un autre problème est apparu : les models avaient besoin d'appeler leurs classes de la couche persistance afin de pouvoir s'enregistrer dans la base de données : par exemple, l'objet Case appelait la méthode Save() de la classe Case_BD pour sauvegarder ses données. Cependant, les classes de la persistance avait également besoin des models pour recréer les objets une fois les données reçues de la base de données : la classe Case_BD appelait le constructeur de Case. Il y avait donc une interdépendance entre deux couches, chose qui nous fallait résoudre.

4.4.4. L'arrivée des managers

Dates : quelques jours fin Janvier

Il fallait donc résoudre ce problème de «God-Object» et d'interdépendance. Après maintes réflexions, nous avons décidé de créer une nouvelle couche, nommée « managers », afin de gérer les objets. Un unique manager pour gérer toutes les instances d'un type d'objet. Par exemple, nous avons créé Case_Manager, contenant la liste de toutes les instances de cases et pouvant agir sur eux.

Couche managers

Les managers détiennent de plus les algorithmes de traitement préalablement dans app.js. Cela supprime l'anti-pattern « God-Object » de ce dernier. De plus, nous avons décidé que ce ne seront plus les models qui appellerons la persistance pour enregistrer les données mais les managers eux même. Désormais, ce sont eux qui appellent la persistance et leur envoient les objets qu'ils gèrent afin de mettre à jour la base de données. La couche model n'est désormais plus dépendante de la couche persistance, l'interdépendance est supprimée.

App.js

Ce fichier perd donc les algorithmes de traitement, et se soulage de beaucoup de lignes de code. Désormais, ce fichier sert juste de coordinateur : il reçoit les demandes clients, les transmet au manager le plus approprié, récupère la réponse du manager puis répond au client. Il ne contrôle rien. Par contre, il s'occupe toujours du rafraîchissement en temps réel des interfaces client : en cas d'action réussie par exemple dans une case, il va rafraîchir les vues des utilisateurs concernés.

De plus, il a fallu configurer le chargement de données en mémoire grâce à ce fichier : nous avons alors fait au sorte que au lancement du serveur, ce denier appel chaque manager, et lui demander de charger les données en mémoire. Il a fallu faire attention au sens des dépendances entre les managers, car certains nécessitait de consulter les données d'un autre.

Couche model

Cette couche, comme dit plus haut, perd seulement son accès vers la couche persistance. Aucune autre modification n'est apportée, seulement des petites améliorations algorithmiques.

Difficultés rencontrées

Mettre en place cette nouvelle architecture nous a pris du temps, et il a fallu retester toute l'application, car la transcription des algorithmes ne s'est pas fait sans mal. De plus, la couche disposant de quatre managers, il a fallu créer un arbre de dépendances afin de ne pas créer d'interdépendance, et éviter la programmation dite «spaghetti» (un manager ne devait pas en appeler un autre qui aller en appeler un autre …).

4.4.5. Messages et attaques de nuit

Dates : fin Janvier – début février

La plus grosse partie du jeu était faite: les joueurs pouvaient commencer à jouer, interagir avec les objets, se déplacer de case en case, et s'attaquer entre eux. Dès que la vie d'un joueur tombait à zéro, une partie de son équipement tombait à terre (aléatoirement) et il était renvoyé dans sa zone sûre, le tout conformément au cahier des charges.

Cependant, il restait du travail pour le respecter. Et la prochaine étape fut l'introduction des messages, afin de garder un historique d'actions passées visualisable par le joueur. Et cela s'est fait assez simplement : l'ajout d'un attribut « listeMessageEnAttente » dans l'objet Personnage, et l'implémentation de fonctions pour ajouter un message. Il a ensuite fallut appeler cette fonction à différents endroits dans le code, et cette partie était terminée.

Pour les attaques de nuit, c'était un peu plus compliqué. Pour rappel, ces attaques sont des événements programmées qui augmentent le nombre de zombies dans chaque case, et ces derniers attaquent automatiquement les joueurs dans ces salles s'ils ne sont pas cachés. Il a fallu ensuite prévenir tous les utilisateurs connectés de l'attaque, en leur affichant une image sur toute la superficie du plateau de jeu.

Difficultés rencontrées

C'est bien la programmation de cet événement qui nous a posé problème. Au début, nous avions utilisé la méthode *setInterval* de JavaScript afin de répéter l'action toutes les tant d'heures (qu'on pouvait facilement configurer), mais cela ne faisait pas ce qu'on voulait : déclencher l'action à heure fixe. Ce n'est que bien plus tard que nous avons découvert le package *Node-schedule* pour déclencher cet événement à heures fixes.

4.4.6. : Sessions, scores et classement

Dates : début février

A ce niveau-là, tous les mécanismes du jeu étaient développés, il restait à mettre en place les sessions de jeu et les scores des joueurs. Il a fallu créer alors trois nouvelles classes : Score, contenant les score d'un joueur pour une session, Score_Manager, afin de gérer tous les scores différents, et Session_Manager. Pas d'objet session car il n'y avait que une seule session à la fois. Nous nous ne désirions pas charger les données sur les autres sessions, car elles n'étaient jamais utilisées. Elles étaient stockées dans la base de données, mais nous n'avions pas besoin de récupérer les données pour le moment.

Avec les sessions, il fallait détecter, lors de la demande d'un utilisateur d'accéder à la page de jeu, si le personnage de l'utilisateur appartenait à cette session. Si oui, le plateau de jeu lui était affiché, ainsi que les scores en cours. Si non, on devait l'informer de la fin de la session à laquelle il appartenait, un bref rappel des scores d'équipe et des siens de cette session, puis lui afficher un formulaire afin de choisir son équipe et sa compétence. A la validation, son personnage était mis à jour et il pouvait enfin jouer dans la session courante.

Une fois mis en place, chaque personnage avait son objet score à lui, un par session. Cet objet stockait plusieurs statistiques, comme le nombre de zombies tués, le nombre de fois blessés et autre, informations qui seront utilisés pour afficher le classement des joueurs sur le site web. Pour ce dernier, nous avons dû modifier la coordination afin de recevoir le tri désiré par l'utilisateur (tri par nombre de points grâce aux combats par exemple). Un argument est donc ajouté dans l'URL afin de savoir quel type de tri, et il suffisait ensuite de demander la nouvelle liste au manager des scores, puis de la renvoyer au client.

Difficultés rencontrées

Les scores nous ont posé beaucoup de problèmes, toujours algorithmiques : gestion d'une liste à clé à deux dimensions (car la liste des scores avait comme ligne l'id de l'utilisateur et comme colonne l'id du score). De plus, il a fallu faire attention au chargement des données, car tous les personnages n'ont pas de scores dans chaque session.

4.4.7. : L'administration et la confirmation de compte

Dates : mi-février

La dernière étape une zone administration. Celle-ci avait plusieurs rôles : la gestion des comptes et des sessions. Pour la première, on a développé un système simple pour supprimer les comptes. Cela avait pour effet de supprimer l'utilisateur et son personnage de la base de données, ainsi que de déconnecter l'utilisateur s'il était connecté (en le prévenant de la suppression de son compte). Radical en soi. Puis, pour la gestion des sessions, cette zone devait permettre la création d'une session, en spécifiant une date de fin, qui mettra automatiquement fin à la session en cours, réinitialisera les cases à partir d'un petit fichier texte placé dans la persistance

De plus, nous avons implémenté un système de confirmation de compte, afin que les utilisateurs ne créent pas plusieurs comptes. Pour cela, chaque inscription enregistre l'utilisateur avec un champ "compteConfirme" égal à false et un champ 'idInscription', chaine de caractères générée grâce à cinq hachages en SHA1 de son pseudo. Ensuite, un mail lui était automatiquement envoyé en lui demander de cliquer sur un lien (de type zombisita-beta.dyndns.org/confirmationCompte/*idInscriptionDeLUtilisateur)*. Dès qu'il clique dessus, le champ "compteConfirme" passe à *true* et il peut être se connecter au site.

Serveur

Dans la coordination, nous avons donc dû placer nos fonctions afin de détecter chaque appui sur un bouton de la zone administration. Ensuite, c'est comme le reste : appel des managers (et donc avec la création de fonctions prévues à cet effet) (ici les managers de Case, pour réinitialiser les cases, et Personnage pour les réinitialiser) et renvoi de la réponse au client.

Difficultés rencontrés

La difficulté ici fut de trouver comment envoyer automatiquement des mails. Après avoir fait fausse route avec un service automatique d'envoi de mail trouvé sur l'Internet (qui demandait un email avec un nom de domaine privé), la découverte du package *Nodemailer* nous a permis d'envoyer facilement et automatiquement des mails à partir d'un compte Yahoo créé pour le projet.

4.5. Bilan de l'application

Au final, voici les possibilités offertes par l'application, séparés en deux parties : le site web, ce que l'utilisateur peut y faire et le jeu, où l'utilisateur contrôle un personnage :

Site web	Jeu
Gestion de compte • S'inscrire (avec mot de passe haché en SHA1 dans la base de données). • Valider son compte avec la	Actions "gratuites" : • Ramasser / déposer des objets. • S'équiper / se déséquiper d'arme/armure. • Consommer des potions / nourriture. • Voir la liste des alliés / ennemi dans la case.

réception d'un mail.
- Se connecter / se déconnecter.
- Choisir une équipe / une compétence pour son personnage à chaque nouvelle session.

Communication entre joueurs
- Discuter sur un tchat avec tous les joueurs.
- Discuter sur un tchat avec son équipe.

Scores et classement
- Consulter le classement des joueurs.
- Consulter les scores des équipes en cours.

Administration
- Créer une nouvelle session (remise à zéro des score / cases / personnages).
- Stopper une session.
- Ajouter aléatoirement des objets à toutes les cases.
- Supprimer des comptes (et donc leurs personnages).

Autre
- Page d'accueil promotionnelle pour les utilisateurs non connectés.
- Page d'accueil pour les utilisateurs connectés.
- Page de règles du jeu.
- Page de tutoriel pour apprendre comment jouer.

Actions coûtant des points de mouvement :
- Se déplacer d'une case à l'autre.

Actions coûtant des points d'actions :
- Attaquer des zombies présents dans la case.
- Attaquer un ennemi dans la case.
- Effectuer une fouille rapide (probabilité de découvrir un objet (dépend de chaque case)).

Actions passives
- Passer en mode fouille (probabilité de découvrir un objet ou un ennemi caché dans la case toutes les heures).
- Se cacher (invisible aux yeux des ennemis).
- Passer en mode défense (augmentation points d'attaque / défense).

Impact des zombies:
- Si le nombre de zombies dans une case est supérieur au maximum autorisé du joueur (dépend de la compétence choisie), ce dernier ne peut aller que vers une seule case : celle d'où il vient.
- Chaque action effectuée entraîne l'attaque automatique par un nombre aléatoire de zombie. De plus, l'action peut échouer dû à cette attaque.
- Durant les calculs, chaque allié présent dans la case diminue de un de nombre de zombies pris en compte. L'allié fait donc office de "protection".

Principes de jeu:
- Chaque jour le personnage perd un point de faim.
- Si la barre de vie tombe à 0, le personnage est considéré comme gravement blessé. Il est donc ramené dans sa zone sûre, et perd une partie de son équipement.
- Chaque nuit (à 3 heures du matin), c'est *l'attaque de nuit* : chaque case voit son nombre de zombies augmenter. Ces derniers attaquent les joueurs qui ne sont pas cachés, et tous les joueurs regagnent le plein de points d'action et de déplacement.
- Si la barre de faim descend en dessous des 10 points (sur les 20 prévus), les caractéristiques maximales (santé / points d'actions / points de déplacement) se verront diminuer.
- Chaque équipe a une zone sure : aucun zombie n'apparaît dedans et aucun ennemi ne peut y entrer.
- Chaque joueur peut porter un nombre limité d'objets, qui dépend du poids du sac de chacun.
- Le joueur dispose d'un espace "messages", où il sera informé des actions qui se sont déroulées dans la même case que lui.
- Un seul, ou au maximum deux zombies, sont tuables en une seule attaque (pas de barre de vie) et infligent des dégâts aléatoires.

4.6. Conclusion

Malgré le fait que la phase de conception nous a aidés à démarrer, l'apprentissage des technologies "sur le tas" n'est jamais simple, car nous prenions du temps à chercher des solutions à nos problèmes, des solutions qui ne sont parfois pas les plus optimales. Alors nous reprenons notre travail, nous recommençons, nous faisons cela mieux. C'est comme apprendre à dessiner : on crayonne, on gomme, on recommence. Jusqu'à temps que l'on soit satisfait du résultat. Cela n'a pas toujours été facile, et plus le nombre de lignes augmentait, plus les bugs étaient nombreux, malgré qu'ils furent corrigés au fur et à mesure.

De plus, il a fallu faire en sorte que le travail de tous puisse communiquer, et cela passa par une gestion centralisée du projet, par l'intermédiaire du chef de projet. Les délais posés, au vu du nombre de fonctionnalités envisagées, étaient souvent justes, mais cela restait un plaisir de travailler tous ensemble, et de voir le projet se construire, petit à petit.

Nous avons au final construit une application qui respecte le cahier des charges initial, avec des fonctionnalités en plus, et cela est très satisfaisant.

5. Gestion de projet

Pour assurer sa réussite, chaque projet a besoin d'être planifié, organisé et réfléchi. Dans le développement logiciel plus que tout, car il faut savoir que seul 50 % de ces projets, dans le monde professionnel, aboutissent à des surcoûts, des dépassements de délais et quelques fois à des abandons. De plus, chaque projet doit avoir un leader (aussi appelé "chef de projet") afin de guider le groupe dans la même direction, et prendre des décisions. Pour ce rôle, c'est Johan qui a choisi d'endosser cette responsabilité.

5.1. Cycle de vie

Pour développer un logiciel, il existe plusieurs cycles de vie, du plus robuste (cycle en cascade, cycle en spirale) au plus souple, en utilisant les méthodes agiles (par exemple, Extreme Programming). Notre équipe étant majoritairement composée d'étudiants n'ayant qu'une faible expérience dans le développement logiciel, un cycle de développement en cascade nous paraissait plus sûr, et nous a permis de prendre le temps de bien réfléchir avant de nous lancer.

Comme présenté sur la figure 43 ci-contre, ce cycle de vie nous imposait donc un découpage en plusieurs phases bien distinctes, que nous avons légèrement modifié avec l'ajout des phases "optimisation" et "version 2.0", qui n'existent habituellement pas dans ce cycle.

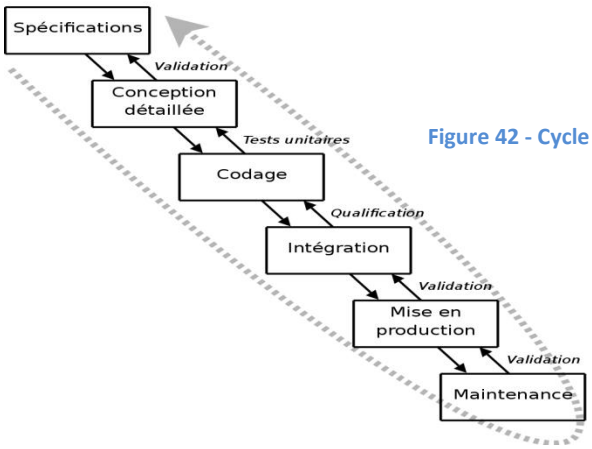


Figure 42 - Cycle de développement en cascade

Voici donc comment s'est déroulé le projet:

- **Analyse**

  Cette phase est l'analyse des besoins du client, et la spécification des fonctionnalités de l'application. Mots clés de cette période : **imagination** et **pragmatisme**.

- **Conception**

  Phase primordiale, elle permet de réfléchir et de concevoir notre future application avant de se jeter à corps perdu dans la programmation. Cette phase, souvent négligé dans les petits projets, reste une phase importante car elle permet de bien définir les différents *objets* de l'application, l'*architecture,* ainsi que les *maquettes* des interfaces graphiques.

  C'est un peu comme l'architecte qui dessine les plans de la maison en veillant bien à respecter le cahier des charges, afin que l'ouvrier n'ait pas à réfléchir comment faire mais simplement faire. L'idée de la phase de conception est là : réfléchir et définir avant de construire, afin de savoir par où commencer, mais également pour éviter les risque de devoir tout refaire car on n'avait pas pensé à un détail (imaginez-vous vouloir mettre une salle de bain au deuxième étage de votre maison alors que la plomberie ne se limite qu'au rez-de-chaussée… "J'aurais dû y penser avant", diriez-vous).

  Mots clés de cette période : **compréhension** et **réflexion**.

- **Développement**

  La plus longue période, mais également la plus enrichissante. Globalement, cette phase réunie proportionnellement trois activités : "apprentissage", "recherche de solutions" et "écriture du code". Cette dernière devait être effectuée avec rigueur, en prenant garde à respecter les règles définies en conception et en faisant l'effort de conserver un code propre. *"Laissez toujours le code plus propre que vous ne l'avez trouvé".*[8]

  Mots clés de cette période : **rigueur** et **réflexion**.

- **Test**

  Les tests de fin de développent ont deux buts : dans un premier temps, de vérifier que toutes les fonctionnalités du cahier des charges ont été implémentés, et dans un seconde temps, de vérifier que le programme ne détient plus aucun bug, et que tout fonctionne correctement.

- **Optimisation**

  Réduction du temps de rafraîchissement, de temps de traitement des données, voilà ce qui est important. Notre but était de fournir une application fluide.

- **Amélioration**

  Utilisation du temps restant pour développer de nouvelles fonctionnalités.

- **Déploiement**

  La plus courte des périodes, car elle consiste seulement en l'installation de l'application et de la base de données sur une machine Linux, faisant office de serveur.

- **Version 2.0**

  Refonte des interfaces (site web et plateau de jeu) pour une meilleure ergonomie.

[8] Citation inspirée d'une ancienne règle des Scouts.

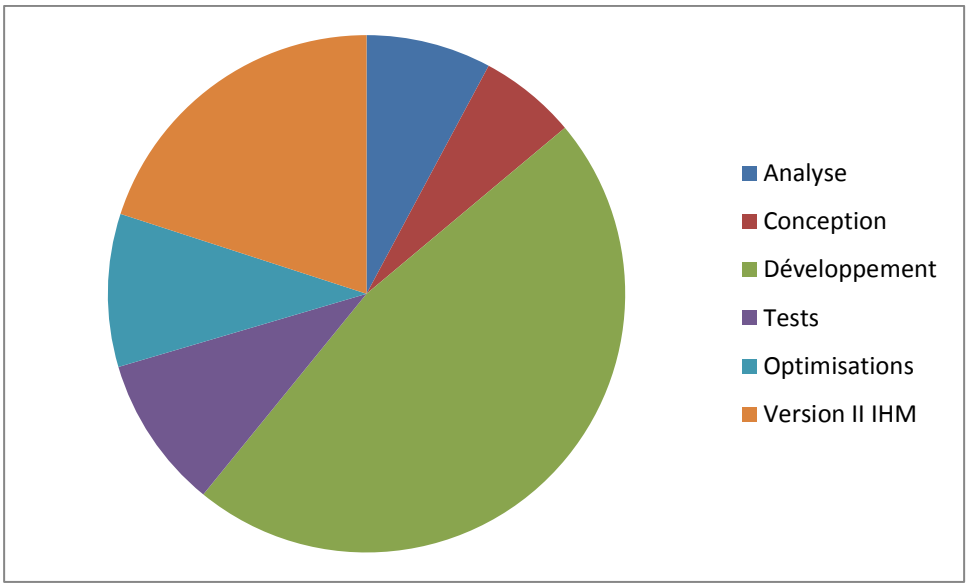
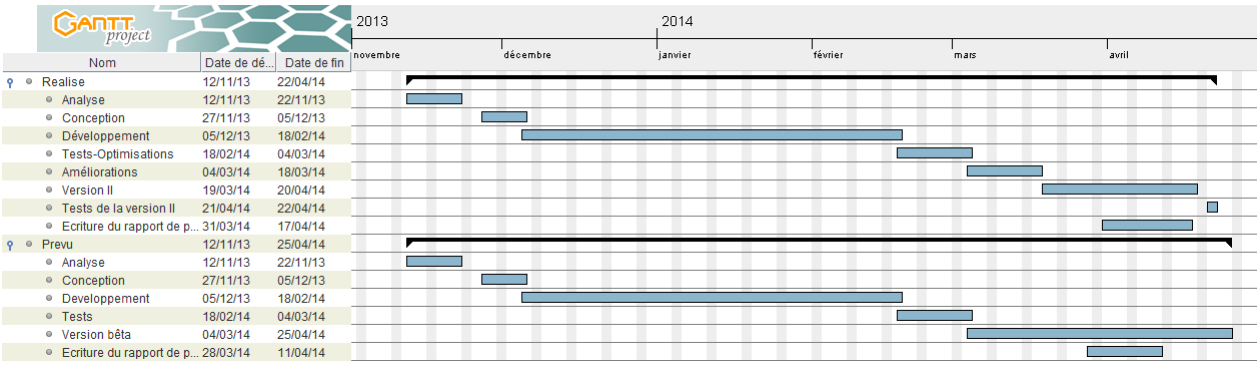


Figure 43 - Répartition du temps de travail

5.2. Macro-planning

Le macro-planning défini en début de projet est définit en figure 45. On peut voir que le Gantt réel était fidèle à ce que nous avions prévu. Et dans la première partie de cette figure, et en dessous se trouve le Gantt prévisionnel.



Nom	Date de dé...	Date de fin
Realise	12/11/13	22/04/14
Analyse	12/11/13	22/11/13
Conception	27/11/13	05/12/13
Développement	05/12/13	18/02/14
Tests-Optimisations	18/02/14	04/03/14
Améliorations	04/03/14	18/03/14
Version II	19/03/14	20/04/14
Tests de la version II	21/04/14	22/04/14
Ecriture du rapport de p...	31/03/14	17/04/14
Prevu	12/11/13	25/04/14
Analyse	12/11/13	22/11/13
Conception	27/11/13	05/12/13
Developpement	05/12/13	18/02/14
Tests	18/02/14	04/03/14
Version bêta	04/03/14	25/04/14
Ecriture du rapport de p...	28/03/14	11/04/14

Figure 44 - Diagrammes de Gantt réel et prévisionnel

Il faut noter que jusqu'à la fin de la partie “Tests & Optimisations”, nous étions complètement dans les temps du planning. Arrivé à ce point, nous avons alors déployé le projet, et tout était prêt pour lancer la bêta test. Cependant, nous avons décidé de continuer à travailler sur l’application, la faire évoluer, et refaire toute l’interface afin de la rendre plus ergonomique et esthétique.

L’idée de la bêta sur un mois ne nous enchantait finalement pas du tout car cela signifiait une période où l’on devait seulement collecter des retours et corriger de petits bugs. Nous ne voulions pas de cela, nous voulions continuer à améliorer et optimiser notre travail. Nous voulions un rendu agréable à voir, rempli de fonctionnalités intéressantes et où il ne manque aucune finition.

5.3. Gestion des sources avec GitHub

Durant le développement en équipe, nous étions obligés d'avoir un lieu où centraliser toutes nos sources. Nous avons donc choisi le site web GitHub.com pour cela. C'est un service web d'hébergement et de gestion de développement de logiciels, utilisant le programme Git, qui permet aux développeurs travaillant sur le projet de télécharger la dernière version du code et d'envoyer leurs modifications du code (plus de détails sur ce programme sont données dans la partie 3.4.1 de ce document). Le site fournit également des fonctionnalités de type réseaux sociaux : flux, suivi de personnes ou de projets, graphes de réseau pour les dépôts, etc. Notre dépôt est consultable à cette adresse : **https://github.com/MainMain/Ei4_Proj3ct**

Figure 45 - Logo GitHub

Durant le développement, lorsqu'un fichier est créé ou mis à jour, par un développeur il est donc «commité» puis envoyé au serveur par l'intermédiaire de la commande *git push*. Puis, une fois connecté au site du dépôt, les membres peuvent consulter les logs afin de voir ce que chacun a fait.

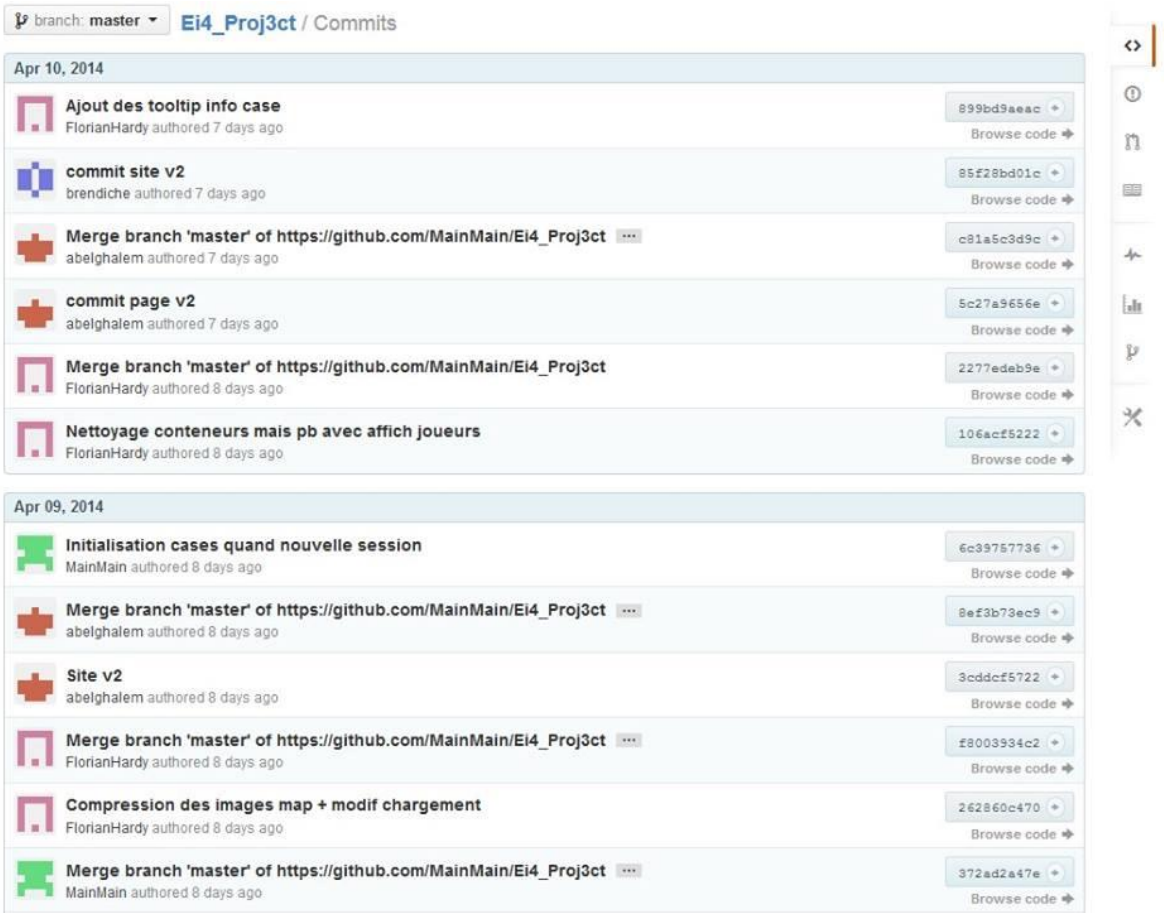


Figure 46 - Logs de commits sur GitHub

On observe alors le numéro du commit (ici non affiché par souci de place) et l'auteur du commit, la date et un commentaire sur ce qui a été fait. Cela permet à n'importe qui de savoir ce qui a été fait et de pouvoir continuer à coder sur la base du travail effectué. Le chef de projet consultait très souvent cet écran afin de constater l'évolution du projet, ajuster le planning en conséquence et éventuellement donner des consignes supplémentaires.

Quand un développeur souhaite se remettre à travailler, il n'avait qu'à entrer la commande *git pull* sur son poste afin de télécharger les modifications effectuées par les autres membres. Ensuite, il pouvait développer, en effectuant les tests en local. En effet, il est préférable de bien tester son code avant de l'envoyer sur le serveur, afin que les autres membres du projet ne se retrouvent pas tous avec la même erreur.

Et c'est grâce à cette gestion des sources que nous avions pu bien découper le travail entre nous quatre, chacun occupant une partie de l'architecture du programme.

5.4. Séparation du travail

Afin de ne pas avoir le souci de travailler sur la même portion de code en même temps, chaque partie de l'application fut délégué à une personne. De plus, partager en plusieurs parties permet de "spécialiser" les membres de l'équipe, afin d'accélérer le développement.

Nous avons donc opté pour ce découpage du travail, selon les envies et goûts des personnes :

- Site web (HTML/CSS) : Abdelmounaim
- Plateau de jeu : Florian
- Serveur : Johan
- Persistance : Brendan
- Modélisation 2D : Brendan - Florian

Bien entendu, tout n'était pas rigide, et un peu de souplesse permettait d'accélérer le développement. Cela signifie que n'importe qui pouvait toujours travailler sur la partie d'un autre pour résoudre une erreur, ou encore implémenter de lui-même une partie de programme afin de tester son code. Le seul risque à faire cela est la création d'un "conflit", mais qui était souvent relativement simple et rapide à résoudre.

Mais cette séparation ne fut vraie qu'au début du projet. On a pu voir au bout de 4-5 semaines, Abdelmounaim venir travailler en renfort sur le serveur (qui demandait plus de travail que le site web). Quand deux personnes travaillent sur la même partie, il faut rester vigilant à ne pas travailler sur le même fichier source pour ne pas créer trop de conflits (perte de temps), si les deux codent la même fonction.

Arrivés à la version 2.0, les choses ont changé, car seules les interfaces étaient à retoucher. Nous avons alors formé deux groupes : Abdelmounaim - Brendan pour l'interface web et Florian - Johan pour le plateau de jeu. Pour ce qui est du premier groupe, une fois le template[9] du site posé, chacun pouvait travailler ses pages de son côté. En revanche, pour le deuxième groupe, le plateau de jeu n'est programmé uniquement sur un seul fichier JavaScript. Nous avons alors décidé qu'un seul à la fois travaillerait dessus; ainsi, une bonne communication fut nécessaire afin de savoir quand l'un "prenait la main" et "la redonnait".

5.5. Gestion des documents

Le projet générait beaucoup de documents : des suivis de travail chaque semaine, la liste des fonctionnalités, des notes sur la conception de l'architecture, les images du site… Pour cela, nous avons donc crée un Google Drive afin d'entreposer tous ces documents, consultable pour tous.

5.6. Gestion des tâches

Durant les huit semaines premières de développement, c'était le rôle du chef du projet de donner les tâches de chacun pour la semaine suivante. De ce fait, il pouvait assurer une certaine cohérence dans la construction de l'application et dans l'interaction du travail des différents membres de l'équipe. Par exemple, pour la fonction

[9] Comparable au terme "mise en page"

d'inscription, les tâches étaient données de tel sorte que la base de données soit d'abord mise en place, puis le traitement côté serveur et enfin la construction de la vue. Il s'assurait également que la fonctionnalité était correctement réalisée. Chaque semaine se concluait par la rédaction d'un rapport, regroupant les tâches effectuées par chacun et les tâches à réaliser pour la semaine suivante.

Puis, pour les semaines suivantes (à partir de février), la gestion des tâches fut plus souple. Nous commencions à avoir nos habitudes au sein du projet, les tâches étaient toujours distribuées par le chef de projet et données au fur et à mesure, mais pour une durée qui pouvait aller d'une après-midi à plusieurs jours. Chaque tâche effectuée était validée et coché sur la liste des tâches, et une autre était donnée et notée, en assurant toujours la cohérence entre le travail de chacun, tout en prenant soin de respecter le planning.

Durant la version 2.0 du projet (à partir du 19 Mars), les choses ont un peu changées : avant, c'était Johan qui, plus ou moins chaque semaine, donnait individuellement chaque tâche. Durant la version 2.0, donc durant 3 semaines, Brendan était responsable de la partie site web et était complètement autonome, et avait à sa charge de gérer son propre planning, ainsi que de distribuer les tâches pour Abdelmounaim travaillant également sur le site web. Il en était de même pour Florian et Johan.

5.7. Bilan de cette gestion

Le point le plus important est que les délais ont été respectés. A partir de là, cette gestion peut être considéré comme une réussite. Au-delà de ça, la charge de travail n'a jamais trop variée, les semaines étaient assez équilibrées et nous n'avons jamais dû quadrupler notre temps de travail en approche des *deadlines*[10]. La seule, là où nous étions très juste niveau temps était quand on a décidé de passer à la version II, non prévue initialement : on se rend compte que toute chose non prévu dès le début casse tout le rythme préalablement établit. Ce dernier restait assez soutenu tout au long du projet et nous étions dans la nécessité de fournir un bon travail personnel en dehors des heures de projets pour réaliser les tâches assignées. Mais c'était par choix, et nous étions pleinement conscients de la charge de travail qu'imposait un tel projet.

Les todo list (liste des tâches à faire) de chacun étaient toutes définies par le chef de projet, notés dans un seul et unique carnet et soigneusement vérifiées et cochées, ce qui permettait de ne rien oublier au fur et à mesure de l'avancement du projet.

Nous sommes assez satisfaits du choix d'un cycle en cascade, car tout a pu se faire progressivement, de façon construite et réfléchie, et il était plus simple de prévoir la suite du projet au fur et à mesure du temps. En définissant un planning bien à l'avance, et en prévoyant bien les tâches de chacun, nous étions sûrs de respecter les délais si nous respections bien les tâches définies chaque semaine. Et c'est ce qu'il s'est passé.

[10] Terme couramment utilisé signifiant une date limite.

5.8. Quelques statistiques

Nom de la couche	Nombre de lignes de code (LC) et de commentaires (coms)
Plateau de jeu	~ 4300 LCs ~1000 de coms
Managers	~1700 LCs ~350 coms
Configuration serveur et coordination	~1100 LCs ~ 400 ~coms
Models	~200 LCs ~ 57 coms
Persistance	~ 1000 LCs ~150 coms
TOTAL	~ 8300 LCs ~ 2000 coms
Nombre de commits	~ 600

Figure 47 - Tableau des statistiques de code

Distributions de l'heure des commits (où l'on aperçoit que l'on a travaillé chaque jour sur le projet, de environ 10 heures du matin jusqu'à des fois 3 heures de la nuit) :

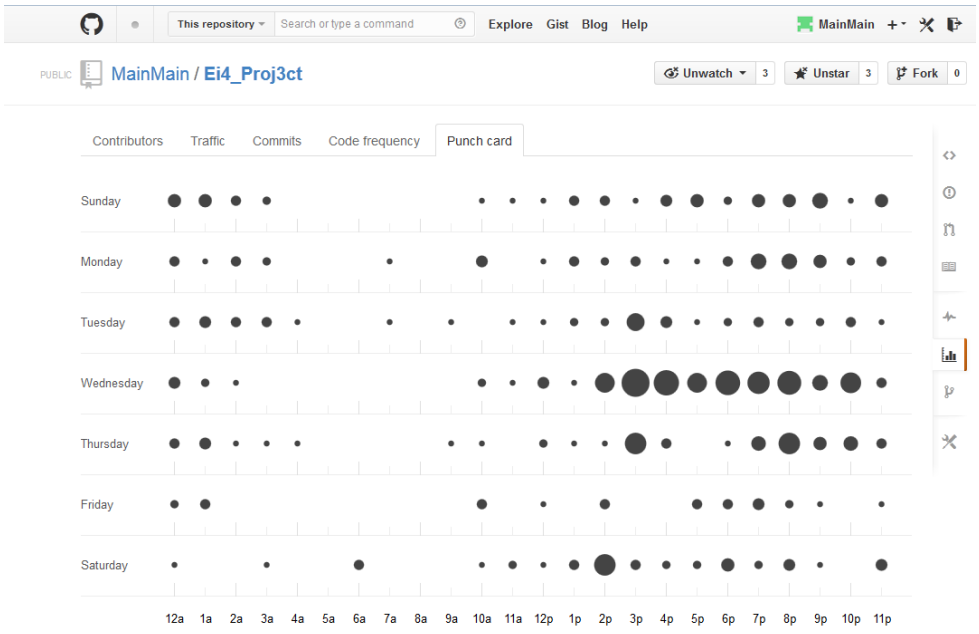


Figure 48 : Distribution de l'heure des commits

Remarque : le plus petit point signifie 1 seul commit, le plus gros correspond à 22 commits.

6. Bilan global

6.1. Idées d'amélioration

Avec un projet comme celui-ci, les possibilités d'amélioration sont nombreuses, et nous pouvons laisser libre cours à notre imagination ainsi qu'à notre créativité. Si un jour nous décidions de continuer ce projet personnellement, nous pourrions envisager de lui apporter plusieurs améliorations.

- Nous pourrions étendre l'exploration de l'IstiA avec cinq étages supplémentaires, dont un sous-sol dont l'accès serait caché et qui renfermerait d'anciens secrets. Avec l'ajout de ces étages, nous pourrions envisager le fonctionnement de l'ascenseur, avec un choix d'étage, mais qui coûterait le même nombre de points de déplacement que le nombre d'escalier à monter/descendre.
- Il pourrait être également intéressant d'instaurer un système de clés afin de déverrouiller certaines salles secrètes. Nous pourrions multiplier le nombre de bonus à attribuer à un personnage, grâce à des équipements supplémentaire (casque, gants, bottes…).
- Pour plus de divertissement, il pourrait être plaisant de pouvoir capturer des zombies et s'en servir comme arme ou bouclier, ou alors, pour ne pas se faire attaquer par les zombies.
- Pour les étudiants de l'IstiA, c'est bien connu, en Amphi E, il est difficile de capter le réseau avec son téléphone, c'est pour cela que l'on pourrait penser à désactiver les tchats pour les joueurs se trouvant dans cette salle.

D'autres améliorations, plus techniques, peuvent également être envisagées :

- Mettre en place un système de "news", publiables par les administrateurs.
- Mettre en place des forums, afin que les joueurs puissent publier sur le site, et créer plusieurs fils de discussion.

6.2. Aspects pédagogiques

Nous avons beaucoup appris, tout d'abord un plan technique, car les enseignements proposés par l'IstiA n'ont pas portés sur tous les domaines utilisés. Nous avons découvert le fonctionnement événementiel et asynchrone de Node.js. Nous avons également appris à programmer en JavaScript et à utiliser un framework pour le canvas. Certes, plusieurs connaissances que l'on nous a enseigné nous furent utiles, comme la programmation objet, quelques principe de conception logiciel, l'algorithmique ou encore le développement de scripts pour Linux, mais pour ce qui est du JavaScript, le NoSQL ou encore la programmation asynchrone, nous avons dû apprendre seul.

Nous avons également appris sur un plan managérial (développement en équipe, séparation du code), même il est dommage de ne pas avoir eu de préparation au travail d'équipe avec Git, ou encore Svn, car ce sont des outils indispensable au bon travail en équipe.

Puis, pour finir, nous avons également appris sur le plan humain, où nous constatons qu'il faut faire preuve d'humilité, d'écoute, d'entraide et de patience pour bien travailler en équipe. Le métier de développeur ne se résume pas simplement à "taper du code". Il faut être patient, rigoureux, et un minimum sociable pour pouvoir travailler en équipe efficacement, et avec plaisir.

6.3. Difficultés rencontrées

6.3.1. Partie plateau de jeu

Nous avons rencontré des soucis avec le navigateur Google Chrome, qui n'affichait pas correctement le canvas. Après quelques recherches, nous avons découvert qu'il fallait rafraîchir en permanence le canvas, avec une fonction de « tick » (à 24 images par seconde par exemple, soit un rafraîchissement toutes les 20 millisecondes).

6.3.2. Port d'écoute du serveur

Originellement, le port d'écoute du serveur était le 8080. Cependant, on a constaté que les sockets ne s'envoyaient pas sur l'ordinateur de Brendan. Impossible de communiquer avec le serveur, même en local. Impossible de développer au final. Ce problème est réapparu sur d'autres ordinateur et ce n'est qu'au bout de longues recherchent que l'on a découvert que cela pouvait provenir d'une règle de sécurité, que l'ordinateur pouvait interdit la création d'une websocket sur ce port. En changeant le port d'écoute du serveur pour le 25536 (arbitrairement choisi entre 1024 et 65535), ce problème fut éliminé.

6.3.3. Téléchargement des images de l'application

Comme nous le savons, il faut que le temps de chargement d'un jeu soit minimal afin de ne pas ennuyer le joueur avant même que celui-ci n'ait commencé à jouer. Concernant notre application, le chargement du plateau de jeu était vraiment, vraiment trop long (dû aux nombreuses images). Le tableau ci-dessous permet de mieux s'en rendre compte.

	Version 1.0	Version 2.0
Connexion 100 Ko/s	7 minutes au premier chargement 42 secondes en cache	2 minutes au premier chargement 12 secondes en cache
Connexion 1 Mo/s (estimée)	~ 1 minutes au premier chargement	12 secondes au premier chargement 3 secondes en cache

Figure 49 - Tableau comparatif des temps de chargement du plateau de jeu

L'optimisation du temps de chargement du plateau de jeu s'est faite en compressant au maximum les images, tout en veillant à conserver leur lisibilité. Cependant, la compression (30 % en moyenne, format jpeg) n'est toujours pas optimale et l'application reste relativement lente à charger, dû principalement au fait que la LiveBox qui relie le serveur à l'Internet à un débit d'upload (envoi des données sur l'Internet) de 100 Ko/s. La vitesse d'affichage du site serait meilleure si le site serait hébergé chez un hébergeur proposant un débit d'upload supérieur à 1 Mo/s.

6.3.4. Jouabilité sur tablettes et mobiles

Durant le projet, nous avions toujours gardé en tête l'idée que notre jeu puisse être jouable sur tablette, voir téléphone portable. Pour cela, nous avons créé de gros boutons sur le canvas, et tenté de faire un site-web redimensionnable à la taille de l'écran, en utilisant donc le "responsive-design". Nous avons intégré sur le canvas tout ce qu'il fallait pour les évènements liés au tactile mais le canvas ne fonctionnait pas toujours : cela dépendant des mobiles / tablettes et du navigateur utilisé. Très aléatoire, et nous n'avons jamais eu le temps de faire de longues recherches pour corriger ces défauts. De ce fait, le jeu n'est pas jouable sur tablette. Dommage donc. De plus, malgré la présence de gros boutons, l'interface de jeu est bien trop petite sur les mobiles : le texte est bien trop petit, et il faudrait repenser une autre version plus ergonomique et adapté aux mobiles avec des écrans de petit format.

6.3.5. Partie serveur:

A lire le chapitre 4 sur le développement, on pourrait penser que ce dernier s'est fait sans mal, sans peine, d'une traite. Mais non. La majorité des problèmes que nous rencontrions étaient algorithmiques, et ne disposant pas d'outils de débogage, la résolution des bugs pouvait être très longue. De plus, il fallait relancer le serveur à chaque modification du code de ce côté, se reconnecter au site…

Par ailleurs, avec le travail à plusieurs, il arrivait parfois, que l'un modifiait une fonction que l'autre utilisait et, sans le savoir, faisait indirectement crasher son code. Ce genre de soucis est arrivé plusieurs fois, ce qui était donc un frein au développement. Pour remédier à ce problème, plus de communication fut nécessaire entre les membres de l'équipe, surtout ceux travaillant sur la même partie, et encore plus quand ils travaillaient sur la même classe (que ce soit un manager ou un objet). De plus, au fur et à mesure de l'ajout de nouvelles fonctionnalités, le code évoluait et de nouveaux bugs firent leur apparition. Et, quelques fois, avec ces nouvelles fonctionnalités, il fallait réécrire des blocs entiers de code, ce qui demandait du temps et beaucoup de réflexion.

6.4. Avis personnels

Abdelmounaim :

Le projet Zomb'IstiA a été pour moi un des projets les plus complet de ma scolarité. J'ai pu, à travers ces quelques mois de travail en groupe, apprendre à utiliser de nouvelles technologies comme le jQuery ou le Node.js, j'ai également eu la chance de découvrir et comprendre le fonctionnement d'appel de fonction asynchrone.

La participation à un projet d'une telle taille utilisant un modèle en couche a été une grande source d'apprentissage sur la façon de réaliser une application de ce genre de A à Z. Sur le côté humain ce projet m'a permis de prendre part à un groupe pendant une longue période et donc d'avoir à tout moment toutes sortes de responsabilités qu'elle soit en vers mes camarades/mon équipe ou moi-même.

Même si quelques problèmes, qu'ils soient techniques ou de délais, ont été rencontrés durant la durée du projet, je tire de cette expérience une très grande satisfaction d'avoir atteint notre but, ainsi qu'un très agréable souvenir.

Brendan :

J'ai rejoint le projet–après la phase d'analyse–car je n'avais pas encore choisi de projet. Ce qui m'a motivé à participer à Zomb'IstiA c'est la quantité de choses qu'il fallait découvrir par soi-même et la façon de travailler qui étaient différente des autres groupes. Au vu du cahier des charges dressé par l'équipe je ne pensais pas que l'on serait arrivé au bout mais je pense que l'on voulait tous relever le défi.

Pour moi travailler en groupe a été un moteur car je ne voulais pas décevoir et ralentir mes collègues. De plus tout au long du projet nous nous sommes soutenus, comme quand j'avais des difficultés sur la base de données ou que je n'avais pas le temps de faire quelque chose. Abdelmounaim, Johan et Florian était toujours prêts à me donner un coup de main, ainsi que leur avis. Je ne retire que du positif de cette expérience car j'affectionne particulièrement ce mode de fonctionnement et l'équipe avec laquelle je me trouvais. En espérant renouveler ce type de projet.

Florian :

J'ai trouvé l'idée du projet très intéressante, car depuis tout petit j'avais l'envie de créer un jeu-vidéo. J'ai fait le choix de m'occuper de la partie graphique car c'est la première chose que voit le joueur. Ainsi, je me sentais plus proche des potentiels utilisateurs et au fil du projet, j'ai pris le temps de leur présenter mon travail, afin de prendre en compte leurs avis, pour que notre jeu plaise au plus grand nombre.

Côté gestion de projet, je suis très satisfait de notre groupe. Nous avons su gérer notre temps comme il le fallait, en nous mettant la pression avec des tâches à réaliser dans des délais précis. Même si ce projet a demandé beaucoup de temps et d'investissement sur notre temps libre, je suis fier du résultat que nous avons fourni. Cela n'aurait pas été aussi simple sans un bon chef de projet et une bonne entente au sein du groupe de travail.

Personnellement, ce projet m'a permis de découvrir et d'apprendre de nouvelles technologies, mais surtout la gestion d'un projet de développement. En cours de génie logiciel, nous avons vu cela de façon théorique, et ce projet était l'occasion de mettre ces connaissances en pratique afin de mieux comprendre le déroulement d'un tel projet. Bref, je n'en tire que du positif.

Johan :

Pour ma part, ce fut une expérience assez enrichissante, que ce soit en termes de développement que de gestion de projet. Le travail en équipe a vraiment du bon, car il nous apprend l'humilité, l'écoute, et la compréhension. On apprend de l'autre, qui saura toujours quelque chose que vous ne savez pas. Niveau technique, j'ai pu perfectionner mes compétences en développement, en algorithmique mais également en conception, à mieux penser les choses à l'avance, à mieux formaliser une architecture et des objets sur les spécifications du cahier des charges.

Pour ce qui est de la casquette de chef de projet, c'est un rôle qui n'est pas toujours simple, car il implique des responsabilités et des décisions. Il faut penser les tâches de chacun à prévoir sur le long terme afin d'être sûr de respecter les délais, sans surcharger les membres de son équipe. Il faut prévoir, prévenir, et suivre le travail de chacun. Maîtriser le temps, et imposer son agenda. Mais en tant que "leader", je pense qu'il ne faut jamais montrer d'inquiétude sur le projet, ne jamais montrer sa surprise aux problèmes. Car ils nous font confiance. Nous avons été confronté à différents problèmes, qu'ils soient techniques ou en terme de délais, et qui ont affecté un peu le moral de l'équipe, et je pense qu'il est de ce rôle d'inspirer la confiance, et de trouver les mots justes pour motiver et rassurer. Ce projet n'est qu'un avant-goût de ce qu'est le vrai management d'entreprise mais c'est une profession qui nécessite à mon goût énormément de compétences humaines et sociales. A nous de trouver au mieux les astuces pour gérer, pour que cela plaise à tout le monde, s'adapter à chaque personnalité.

Ce projet fut un défi pour toute l'équipe, et chacun était parti dans cette optique de "challenge". Nous avons su rester soudés, passionnés, et même si parfois c'était un peu dur (complexité, délais), j'en ressors avec un très bon souvenir, et cela confirme mon projet professionnel.

Conclusion

Pour conclure ce projet, on pourrait commencer par dire que nous avons beaucoup appris, techniquement comme humainement. Dans ce premier cas, nous avons découvert la technologie Node.js qui commence à être de plus en plus populaire dans le monde du développement Web, et qui sera de plus en plus utilisé, et apporte donc une vrai valeur ajouté à notre capital de connaissances. Il en est de même pour Bootstrap, ou encore le NoSQL, comme la mise en place et l'utilisation de MongoDB. Et dans le deuxième cas, le travail en équipe fait beaucoup appel à nos qualités sociales : l'humilité, la patience, l'entraide. C'est important dans un travail en équipe, et c'est durant ce genre de projet que l'on s'en rend compte. La cohésion sociale du groupe est le pilier du projet, qui ne peut aboutir sans cela.

Notre projet est à la hauteur du cahier des charges prévu, mais pas complètement à la hauteur de nos ambitions. Les différents problèmes rencontrés nous ont beaucoup ralentis, et même avec beaucoup de travail, il faut toujours du temps pour apprendre, comprendre et bien utiliser de nouvelles technologies. Nous avons investi beaucoup de temps et d'énergie dans ce projet et, globalement, nous aurions voulu mieux. Plus esthétique. Plus ergonomique. Plus de fonctionnalités, et plus d'interactivités.

Si c'était à refaire, nous garderions l'architecture mise en place, car nous la trouvons robuste et bien organisée. Cependant, nous passerions plus de temps durant la conception, car la modification de celle-ci en cours de route prend beaucoup de temps, et nous savons que nous en avons perdu à causes des refontes d'architecture ou des objets. Même si au final notre phase de conception fut respectée dans les grandes lignes (programmation objet, modèle en couche), le développement aurait été beaucoup plus rapide si cette phase avait été "parfaite". Si le projet devait être repris, les seuls conseils qui pourraient être données seraient de conserver l'architecture mise en place.

"Don't hurry your code. Make sure it works well and is well designed. Don't worry about timing."
Linus Torvald

Glossaire

Client-serveur : architecture très utilisée sur le web. Désigne un mode de communication à travers un réseau entre plusieurs programmes ou logiciels : le client, envoie des requêtes ; le ou les serveur(s) attend(ent) les requêtes des clients et y répond(ent).

Design pattern : c'est un patron utilisé en conception. Il permet de résoudre des problèmes en imposant une structure. Par exemple, le pattern Observer permet de mettre à jour une vue à chaque changement des données sans que la couche traitement n'ait référencé la couche vue.

Développeur: une personne qui développe des programmes. Peut-être un professionnel (un ingénieur, un informaticien ou un analyste programmeur) ou bien un amateur.

Flash : une suite de logiciels permettant la manipulation de graphiques vectoriels, d'images et de scripts ActionScript, qui sont utilisés pour les applications web, les jeux et les vidéos.

Framework : ensemble de composants qui servent à créer les fondations, l'architecture et les grandes lignes d'un logiciel. L'objectif premier d'un framework est d'améliorer la productivité des développeurs qui l'utilisent.

IDE : pour « Integrated Development Environment », logiciel disposant de multiples outils embarqués afin d'aider et d'assister le développeur dans le développement d'un logiciel.

Langage de programmation : un langage qui permet aux développeurs d'écrire des instructions qui seront analysées et traitées par l'ordinateur.

Langage interprété : langage non compilé, c'est-à-dire qu'aucun fichier exécutable (.exe) n'est créé, le code source reste tel quel. Si l'on veut exécuter ce code, on doit le fournir à un interpréteur qui se chargera de le lire et de réaliser les actions demandées.

Maitrise d'œuvre : personnage physique ou morale ayant la compétence et les moyens de réaliser un projet.

Moteur graphique : composant logiciel permettant de créer des images matricielles. Il dispose d'une API permettant au développeur de facilement créer le graphisme de son application, par exemple un jeu.

Moteur V8 de Google Chrome : fait ce qu'on appelle de la compilation JIT (Just In Time). Il transforme le code JavaScript très rapidement en code machine et l'optimise même, grâce à des procédés complexes.

MySQL : c'est un système de gestion de bases de données. Il s'oppose à MongoDb, car il utilise le langage SQL

Navigateur web : logiciel servant à afficher des pages au format html se trouvant sur le web. Plus techniquement, c'est un client http.

ORM : pour « Object Relational Mapping » permet de faire la correspondant entre un objet utilisé par le langage de programmation du logiciel et le même objet stocké dans la base de données.

Programmation événementielle : logique de programmation raisonnant sur des évènements, c'est-à-dire que quand il se passe telle chose, alors on fait cela. Elle s'oppose à la programmation séquentielle.

Pixel : petit carré possédant des propriétés de coloration. Une image est composée de plein de pixels, et c'est l'association de plusieurs pixels qui va former l'image.

SHA1 : algorithme de hashage, transformant n'importe quelle chaine de caractère hexadécimale en une suite de 160 bits. Le risque de collision (deux mots différents qui donnent le même résultat) est très faible.

Service (en informatique) : processus tournant en arrière-plan, comme par exemple un anti-virus.

Singleton : design pattern souvent utilisé lorsque l'on a besoin d'exactement un objet pour coordonner des opérations dans un système. Il s'assure qu'un objet n'aura qu'une seule instance de lui-même.

Spritesheet : une image composée de plusieurs petites images, appelées sprites. Les spritesheets sont utilisées pour les animations dans les jeux en deux dimensions (2D).

SSH : protocole de communication sécurisé. Le protocole de connexion impose un échange de clés de chiffrement en début de connexion.

Template : analogue à « mise en page »

Toolkit : extension gratuite pour Flash Professional CS6 qui permet de publier du contenu Flash en HTML5, en s'appuyant sur CreateJS.

Websocket : tunnel de communication bidirectionnel entre les navigateurs et les serveurs web.

Widget : composant d'interface graphique.

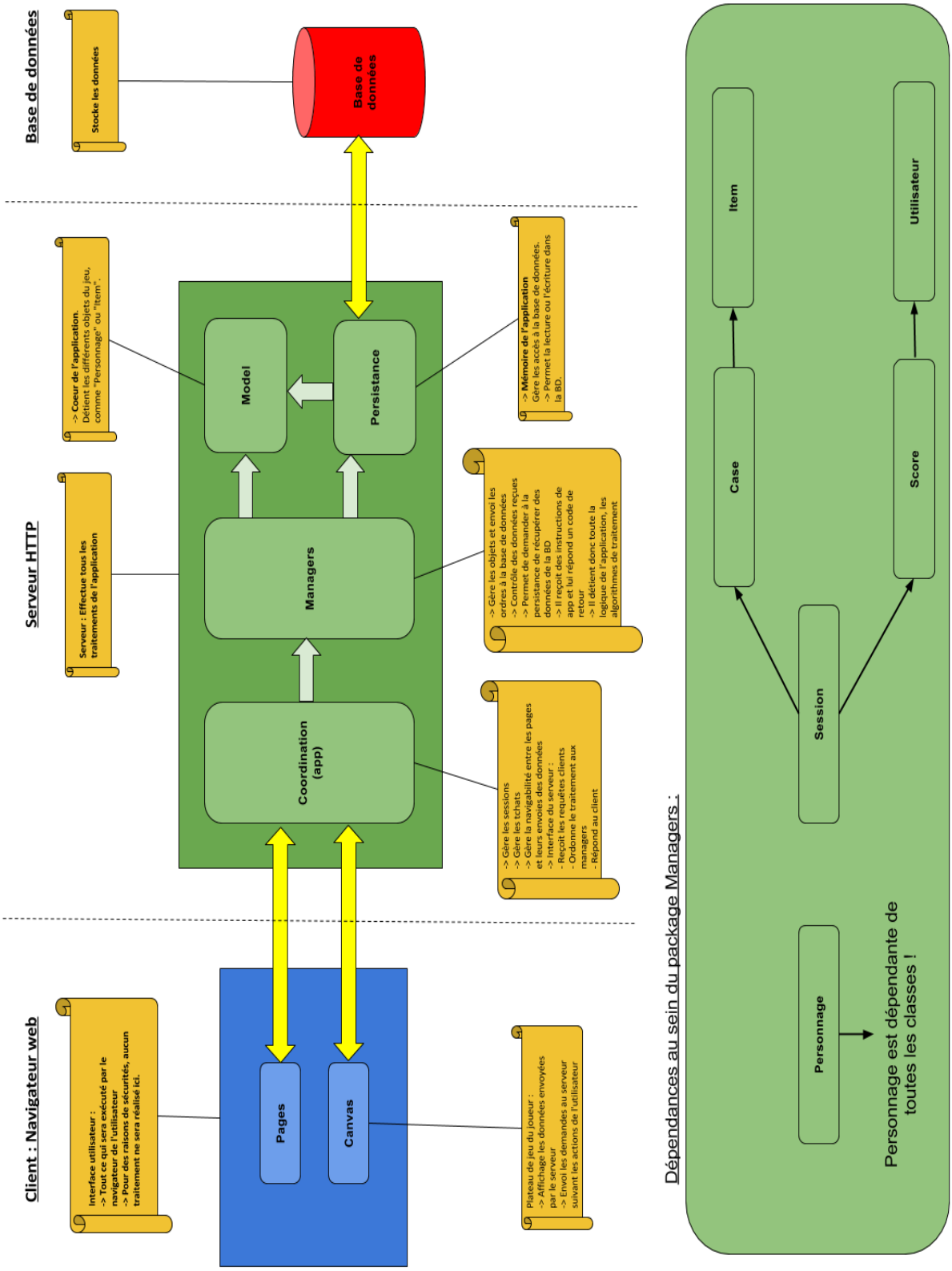
WYSIWYG : What You See Is What You Get, qui signifie "ce que vous voyez est ce que vous aurez". En effet, on crée à l'aide d'un outil une interface graphique, et on obtient le code de ce que l'on a créé.

Zoë : application Adobe AIR qui convertit les animations SWF sur des spritesheets (feuilles de sprites).

Table des illustrations

Table des Annexes

Annexe 1 – Schéma détaillé de la conception en couche du serveur



Annexe 2 - Code concernant le plateau de jeu

Code HTML et sa balise <canvas>

```
<canvas id="mon_canvas" width="1100" height="620">
	Si vous voyez ce message apparaitre, c est que votre navigateur ne supporte pas
Canvas
</canvas>
```

Code JavaScript permettant de créer le canvas

```
// Dès que <canvas> est appelée dans le code HTML (onload), on dit quelle sera la première
fonction
onload = initialize;

// Première Fonction appelée
function initialize()
{
    // Création du canvas
    canvas = document.getElementById("myCanvas");
    //Création du stage (la scène)
    stage = new createjs.Stage(canvas);
    // autoriser le mouse over / out events
    stage.enableMouseOver(20);
    // autorise le tactile si l'appareil le supporte
    if(createjs.Touch.isSupported())
    {
        createjs.Touch.enable(stage);
    }
```

Exemple de création d'un conteneur

```
// Création d'un conteneur
var mon_conteneur = new createjs.Container();
// Ajout du conteneur au stage
 stage.addChild(mon_conteneur);
```

Exemple de création d'un texte

```
// Création du texte
var mon_label = new createjs.Text("Texte du label", police, couleur);
// Modification du texte
mon_label.text = "Nouveau texte";
```

Exemple de création d'une image

```
// Création de l'image, en précisant la source de l'image à ajouter
var mon_image = stage.addChild(new createjs.Bitmap("mon_image.png"));
```

Exemple d'ajout d'éléments dans un conteneur

```
mon_conteneur.addChild(mon_label, mon_image);
```

Annexe 3 - Exemple de création d'une barre de chargement sur le canvas

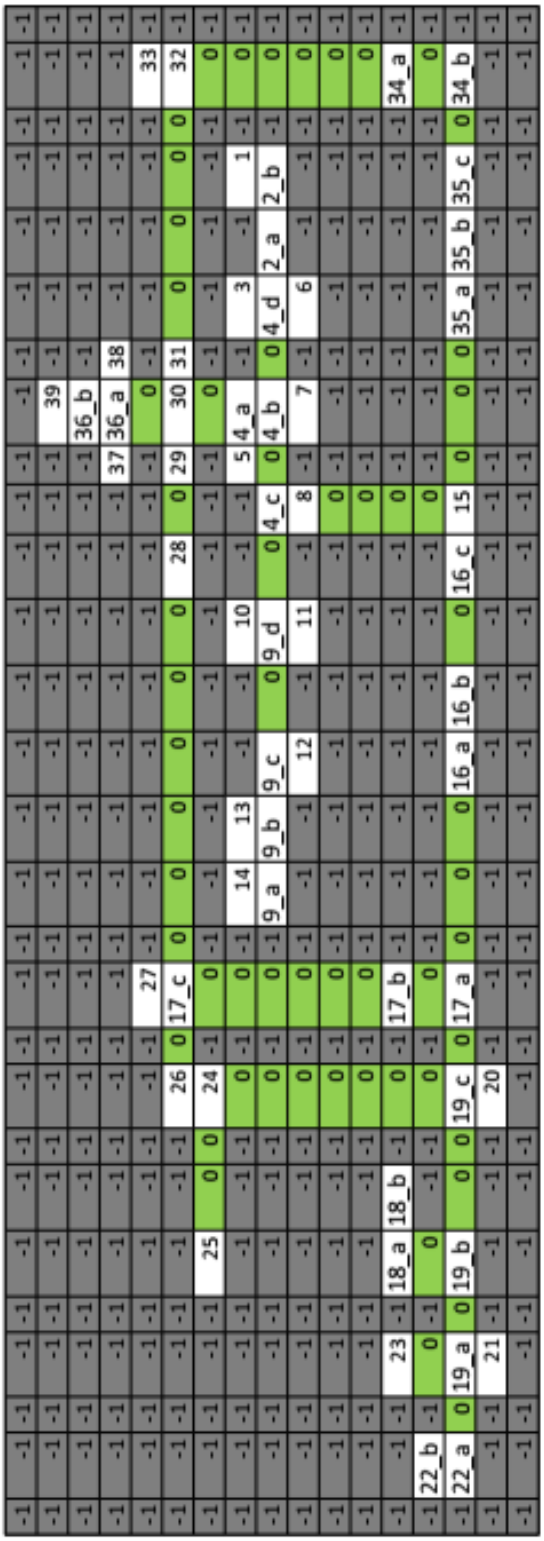
```
//Dans la fonction d'initialisation :

    // Variable permettant de définir tous les éléments à charger
    var manifest = [{src:"image.png", id:"idImage"}, … ]

    // Création d'un label affichant un message pendant le chargement
    loadProgressLabel = stage.addChild(new createjs.Text("","70px Fstein",colorChargement));

    // Création d'un second label affichant le pourcentage du chargement
    labelPourcentLoad= stage.addChild(new createjs.Text("","70px Fstein",colorChargement));

    // Création d'un conteneur pour placer la barre de chargement
    loadingBarContainer = new createjs.Container();

    // Définition des caractéristiques de la barre
    loadingBarHeight = 80;
    loadingBarWidth = 700;
    LoadingBarColor = createjs.Graphics.getRGB(95,0,0);

   // Création de la barre de chargement en elle même
    loadingBar = new createjs.Shape();

    // Son remplissage 'ici à 0'
    loadingBar.graphics.beginFill(LoadingBarColor).drawRect(0, 0, 1,
loadingBarHeight).endFill();

   // Création du contour de la barre de chargement
    frame = new createjs.Shape();
    frameBarColor = createjs.Graphics.getRGB(0,0,0);
    padding = 3;

    // Dessin du contour de la barre
    frame.graphics.setStrokeStyle(1).beginStroke(colorChargement).drawRect(-padding/2, -
padding/2,     loadingBarWidth+padding, loadingBarHeight+padding);

    // Ajout de la barre de chargement et de son contour dans le conteneur
    loadingBarContainer.addChild(loadingBar, frame);

    // Ajout du conteneur de la barre de chargement au stage du canvas
    stage.addChild(loadingBarContainer);

    // On charge la pile avec la variable contenant tous les éléments
    preload.loadManifest(manifest);

// Fonction - Pendant le chargement du canvas
function handleProgress()
{
    // Remplissage de la barre de chargement
    loadingBar.scaleX = preload.progress * loadingBarWidth;

    // Calcul du pourcentage de progression
    var progresPercentage = Math.round(preload.progress*100);

    // Affichage du pourcentage
    loadProgressLabel.text =("Loading Apocalypse");
```

```
    labelPourcentLoad.text= progresPercentage + " %";
}
// Fonction - Une fois le chargement du canvas terminé
function handleComplete()
{
    // Suppression du label de chargement
    stage.removeChild(labelPourcentLoad);

    //Affichage d'un nouveau texte
    loadProgressLabel.text = "Click to Survive";

    // Ajout des événements pour démarrer le jeu
    canvas.addEventListener("click", handleClick);
    canvas.addEventListener("touchstart", handleClick);
}

// Fonction - Click de l'utilisateur après le chargement
function handleClick()
{
    // Suppression des outils utilisés pour le chargement
    stage.removeAllChildren();

    // Suppression des événements sur le canvas
    canvas.removeEventListener("click", handleClick);
    canvas.removeEventListener("touchstart", handleClick);

    //Lancement de la et affichage du plateau de jeu
    start();
}
```

Annexe 4 - Découpage des cases de la carte

Annexe 5 - Matrice résultant du découpage des cases

Les nombres sont les id des cases, les -1 sont les endroits où le personnage ne peut pas aller, les 0 permettent de structurer la matrice, ce ne sont pas des cases, ils sont ignorés par l'algorithme.



Annexe 6 - Procédure de déploiement

En italique rouge les commandes à taper sur un terminal Unix, en root :

- Installer Git, afin que le serveur puisse se connecter au dépôt des sources et de télécharger les dernières mises à jour, de façon simple (en une seule commande). Une fois Git installé, il faut l'initialiser et cloner le dépôt sur notre serveur. Cette étape de clonage n'est à faire qu'une seule fois.
  - *apt-get install git*
  - *git init*
  - *git clone https://github.com/MainMain/Ei4_Proj3ct*
- Installer les packages utilisés par l'application. Si le fichier package.json est à jour, une seule commande suffit. (S'il manque des packages, les installer à la main avec la même commande)
  - *npm install -g*
- Installer NodeJs, afin de pouvoir lancer l'application.
  - *apt-get install nodejs*
- Installer mongoDb, afin de pouvoir faire tourner notre base de données.
  - *apt-get install mongodb-org*
- Sachant que nous développions en local, dans les sources clients nous spécifions l'adresse sur serveur en localhost. Cependant, il faut changer cela, car le serveur n'est désormais plus sur les postes clients mais distant ! Un petit script, en utilisant la commande *sed* permet d'ouvrir chaque fichier et de remplacer les localhost en l'adresse du serveur. (Voir annexe 5)

- Il y a cependant encore un détail à régler : que l'application tourne en tant que processus d'arrière-plan, comme un service. Nous avons pour cela installé le package *forever*, qui permet de lancer notre serveur en tant que service, l'application tourne en continu, même lorsque personne n'est connecté en ssh au serveur.
  - *npm install -g forever* Lancer la base de données puis le serveur (la base de données doit
  - service mongod start
  - node app.js

La partie suivante est optionnelle :

- Réserver un nom de domaine. Effectivement, quand nous nous connectons à un site web, nous utilisons une adresse URL compréhensible par un être humain, nous n'utilisons pas l'adresse IP. Pour cela, nous avons utilisé un service gratuit[11] du site dyndns.org afin de réserver un nom de domaine, qui redirigera les clients au serveur.
- Ouvrir un accès ssh, afin de pouvoir administrer notre serveur à distance (par exemple à l'aide de l'outil Putty)
  - *service ssh start*

[11] Ce service gratuit sera définitivement fermé début Mai 2014

Annexe 7 - Script de lancement du serveur

(Rappelons que forever est un package pour Node.js qui, une fois installé sur le serveur, permet de lancer un script Node.js comme processus d'arrière-plan (= service))

```
#!/bin/bash

# Deplacement dans le répertoire du projet
cd ~/Ei4_Proj3ct-master/Ei4_Proj3ct/

echo "Mise à jour des sources..."
# Suppresion des modifications
git checkout .

# Téléchargement des nouveautés a partir du dépôt GitHub
git pull

# Remplacement dans les fichiers sources des "localhost" (adresse de développement)
# .. en zombistia-beta.dyndns.org (adresse de production = URL du serveur)
sed -i "s/localhost/zombistia-beta.dyndns.org/g" ./view/accueil.ejs
sed -i "s/localhost/zombistia-beta.dyndns.org/g" ./view/accueil-connecte.ejs
sed -i "s/localhost/zombistia-beta.dyndns.org/g" ./view/admin.ejs
sed -i "s/localhost/zombistia-beta.dyndns.org/g" ./view/chat.ejs
sed -i "s/localhost/zombistia-beta.dyndns.org/g" ./view/classement.ejs
sed -i "s/localhost/zombistia-beta.dyndns.org/g" ./view/confirmerCompte.ejs
sed -i "s/localhost/zombistia-beta.dyndns.org/g" ./view/game.ejs
sed -i "s/localhost/zombistia-beta.dyndns.org/g" ./view/noAdmin.ejs
sed -i "s/localhost/zombistia-beta.dyndns.org/g" ./view/regles.ejs
sed -i "s/localhost/zombistia-beta.dyndns.org/g" ./view/tutoriel.ejs
sed -i "s/localhost/zombistia-beta.dyndns.org/g" ./app.js

# Arret des processus forever
echo "Arret des processus forever..."
forever stopall

# Lancement su serveur avec forever
# app.js le point d'entre de l'application
echo "Lancement du serveur nodeJs..."
forever start app.js
```

Annexe 8 - Mise en place d'un serveur Node.js simple

Code JavaScript permettant de créer un serveur Node.js minimaliste

```
/*
 * fichier app.js
 */
// Inclus le package http, primoridal
var http = require('http');

// on crée un serveur. Sa config sera accessible par la variable server
var server = http.createServer(function(req, res) {
  // Ecrit le code 200 sur l'entete http de retour (signifique que totu est ok)
  res.writeHead(200);

  // Envoi un message qui sera lisible par le navigateur web client
  res.end('Hello world !');
});
// Le serveur écoute sur le port 8000
server.listen(8080);
```

Le serveur sera lancé de cette façon-là : node app js.

Annexe 9 - Communication client <-> serveur avec socket.io

Code JavaScript coté client pour envoyer et recevoir des données au serveur

```
// Ajout de l'evement "click sur boutonde fouille"
Btn_Mode_Fouille.addEventListener('click', function(event)
{
    // Demande au serveur un changement de mode
    socket.emit('PERSONNAGE_MODE_CS', 2);
});


// Reception d'une réponse du serveur pour le changement de mode
socket.on('PERSONNAGE_MODE_SC', function (mode, reponse, degatsInfliges, nbrGoulesA)
{
    switch(reponse)
    {
    case 1:
        afficherMessageRetour("Vous venez de changer de mode ", 1);
        break;
    case -4 :
        afficherMessageRetour("Changement de mode mode raté ! Vous êtes dans ce mode !",
2);
        break;
    case -10:
        afficherMessageRetour("Changement de mode raté : vous n'avez pas assez de points
d'action !", 3);
        break;
    }
});
```

Coté serveur, on peut reprendre notre code de l'annexe 5, et le compléter avec le code qui suit :

```
// Toutes les sockets reçues arriveront ici
io.sockets.on('connection', function (socket)
{
    // Reception d'une demande de changement de mode
     socket.on('PERSONNAGE_MODE_CS', function (idUser, mode)
    {
       // Envoi l'argument mode envoyé par le client à la couche métier
       var reponse = oPersonnage_Manager.ChangementMode(mode);

        // Renvoi au client la réponse de la couche métier
        socket.emit('PERSONNAGE_MODE_SC', mode, reponse.reponseChangement,
reponse.degatsSubis, reponse.nbrGoules);
    });
}
```

Annexe 10 - Exemple de code de configuration des routes

```
/*
 * fichier /routes/index.js
 */

// exports.jeu : le client a entré l'url site.com/jeu
exports.jeu = function(req, res){
      // On lui revoi la page nommée 'game'
      res.render('game', { title: 'Express' }); // utilisation du framework expres
obligatoire
};


exports.classement = function(req, res){
      res.render('classement', { title: 'Express' });
};
```

Annexe 11 - Communication page web <-> serveur

```
// utilisation du framework express
var app        = express();


app.get('/admin', restrictAdmin, function fonctionAdmin(req, res)
{
    var s = req.session;
    var options = { "username" : s.username, "idUser" : s.idUser, "dateDebut" : null,
"dateFin" : null, "idSession" : null};

    ajouterInfosHeures(options);
    res.render('admin', options);
}
);

app.post('/admin', restrictAdmin, function fonctionAdmin(req, res)
{
    var s = req.session;
    var idUser = req.param("idUser");
    var options = { "username" : s.username, "idUser" : s.idUser, "dateDebut" : null,
"dateFin" : null, "idSession" : null};

    res.render('admin', options);
});

app.put('/admin', restrictAdmin, function fonctionAdmin(req, res)
{
    var s       = req.session;
    var action    = req.param("action");
    var year    = parseInt(req.param("year"));
    var month    = parseInt(req.param("month"));
    var day     = parseInt(req.param("day"));
    var options = { "username" : s.username, "idUser" : s.idUser, "dateDebut" : null,
"dateFin" : null, "idSession" : null};

    var date = new Date(year, month, day, 0, 0, 0, 0);

    switch(action)
    {
        case "demarrer":
               oSession_Manager.demarrer(date);
               break;
        case "update":
               oSession_Manager.definirDateFin(date);
               break;
        case "stop":
               oSession_Manager.stopper();
```

```
                break;
        case "remplir":
                oCase_Manager.RemplirCases();
                break;
    }
    res.render('admin', options);
});
```

Annexe 12 - Configuration d'un serveur utilisant le framework Express

```
// port d'écoute du serveur
app.set('port', process.env.PORT || 25536);
// emplacement des pages (pour le fichier contenant les routes)
app.set('views', __dirname + '/view');
app.set('view engine', 'ejs');

// configuration des sessions
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.cookieParser());
app.use(express.session({secret: "motSecret"}));
app.use(express.methodOverride());
app.use(app.router);

// emplacement de la racine du projet
app.use(express.static(path.join(__dirname, '/')));
// réponse par défaut quand l'utilisateur cherche une URL qui n'existe pas
app.use(function(req, res, next)
{
    res.status(404);

    res.send('Page not found');
});/
```

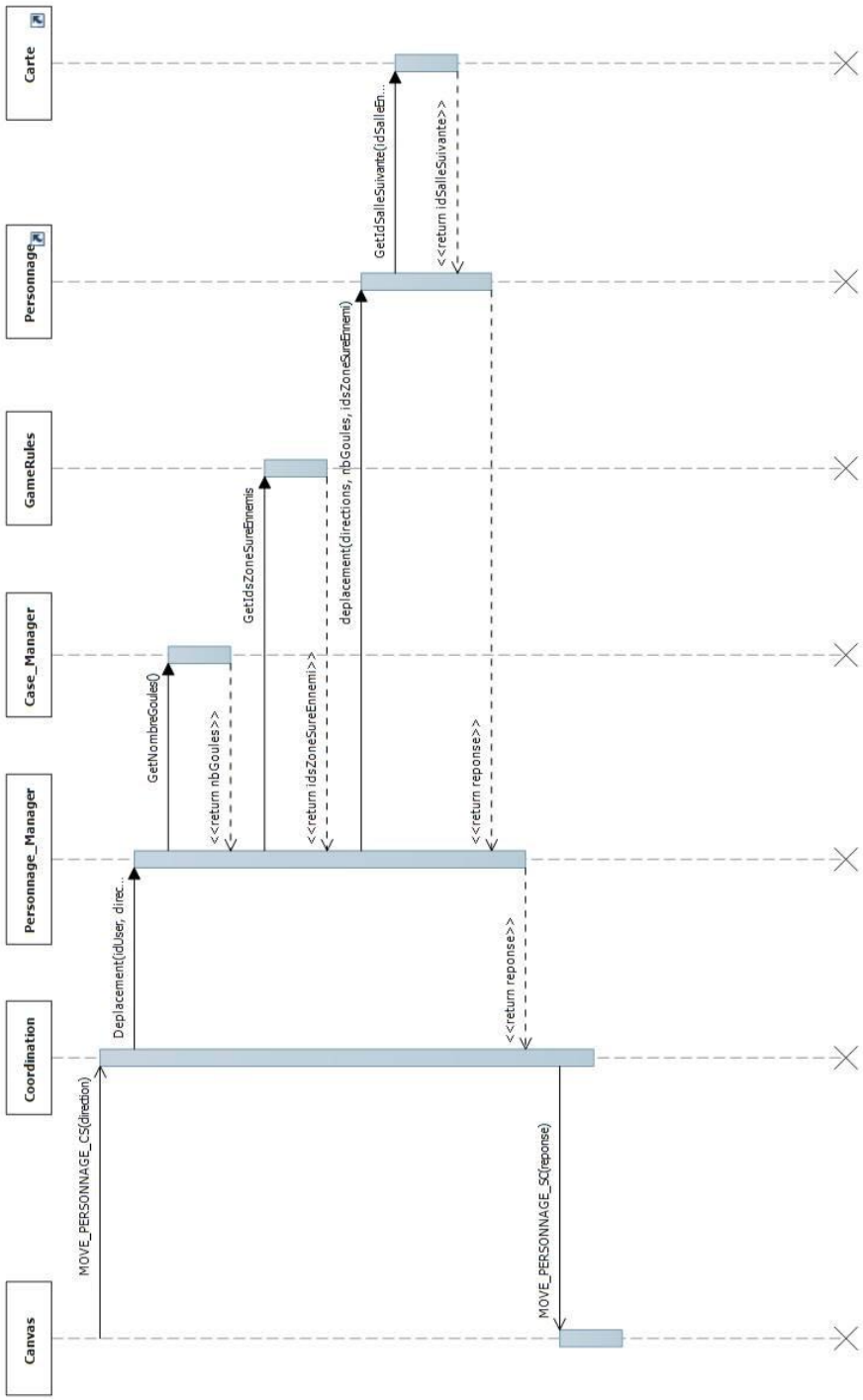
Annexe 13 – Configuration des sessions

```
// Configuration du serveur afin de pouvoir utiliser les sessions
app.use(express.session({secret: "secretKey"}));
// Connexion d'un client à la page /index. Les paramètres sont req pour Request et res pour
Result
app.get('/index', function FonctionIndex(req, res) {
// Si l'utilisateur a une session en cours if(req.session.username != null)
{
// On renvoie sur la page d'acceuil lorsque l'on est connecté
Options.message = « Bonjour » + req.session.username ;
res.render('accueil-connecte', Options);
}
// Sinon
else
{
// On renvoie sur la page de connexion
Options.message = « Veuillez-vous connecter ! » ;
res.render('connexion', options); }
});
```

Annexe 14 - Diagramme de séquence pour la demande de déplacement

Exemple donnée pour la demande de déplacement d'un personnage. « Move » exprime la direction demandée.

Note : Lorsque la couche « Managers » n'existait pas, la communication se faisait directement de la couche « coordination » à la classe « GameRules » ou la couche « Model ».



Annexe 15 – Exemple de définition d'un objet en JavaScript

Exemple pris pour la modélisation d'un Utilisateur

```
//inclusion des règles
var GameRules   = require('../GameRules');
var EventLog = require('../EventLog');
var Utilisateur = (function() {
'use strict';

// --- ATTRIBUTS DE CLASSE ---
Utilisateur.id;
Utilisateur.pseudo;
Utilisateur.email;
Utilisateur.numEquipe;
Utilisateur.idPersonnage;
Utilisateur.idSession;
Utilisateur.compteConfirme;
Utilisateur.idInscription;

// --- METHODES DE CLASSE ---
Utilisateur.build = function() {return new Utilisateur();};
// Constructeur utilisé par la base de données
function Utilisateur(id, pseudo, email, numEquipe, idPersonnage,
       idSession, compteConfirme, idInscription) {
// --- Attributs d'instance
this.id = id;
this.pseudo             = pseudo;
this.email              = email;
this.numEquipe          = numEquipe;
this.idPersonnage       = idPersonnage;
this.idSession          = idSession;
this.compteConfirme     = compteConfirme;
this.idInscription      = idInscription;
}
// --- METHODES D'INSTANCE
Utilisateur.prototype =
{
getIdSession : function() { return this.idSession; },
getNumEquipe : function() { return this.numEquipe; },
getPseudo : function() { return this.pseudo; },
getIdPersonnage : function() { return this.idPersonnage; },
getScore : function() { return this.scoreByMeutreCumule; },
getUser : function() { return this; },
getCompteConfirme : function() { return this.compteConfirme; },
setScore : function(newScore) { this.scoreByMeutreCumule = newScore; },
setNumEquipe : function(newNumEquipe) { this.numEquipe = newNumEquipe; },
setIdSession : function(newIdSession) {        this.idSession = newIdSession; },
confirmerCompte : function() { this.compteConfirme = true; },
};
// On pense à retourner le constructeur (afin de pouvoir construire des
// instances, sinon tout
// le code de notre classe serait inutile car non visible depuis
// l'extérieur)
return Utilisateur;
}());
module.exports = Utilisateur;
```

Annexe 16 - Cahier des charges du projet

« Avez-vous déjà imaginé un univers apocalyptique, où seul la survie compte ? »

Notre projet consiste à concevoir un jeu par navigateur multi-joueurs par équipe, où le but est d'accumuler le plus de points dans une période donnée (1 mois par exemple). Les joueurs, plongés dans un monde persistant (qui continu à vivre quand l'utilisateur est déconnecté) pourront interagir avec ce monde et les autres joueurs afin de réaliser des actions lui permettant d'atteindre son but : accumuler les points pour son équipe. Le tout, en évitant de se faire attraper par les zombies.

But du jeu

Chaque équipe aura sa zone sûre, et le but de jeu sera d'accumuler le plus de points, de deux manières possibles : en ramenant certains types d'objet (objets de défense) à sa base ou en tuant d'autres joueurs. Les objets de défenses (ODD) auront des valeurs, et bien entendu, les plus précieux seront les plus loin et bien cachés. Ils pourront être transportables par les joueurs et seront disposés dans différentes salles.

Fonctionnalités

FG 01 : Gérer les utilisateurs (espace administrateur)

Pouvoir supprimer des comptes si les joueurs ne respectent pas les règles du jeu.

FG 02 : S'inscrire au site

Chaque utilisateur pourra s'inscrire au moyen d'un formulaire. Ces informations seront envoyées au serveur et enregistrées dans la base de données.

FG 03 : Se connecter / Se déconnecter du site

Chaque utilisateur possédant un compte dans la base de données pourra se connecter à l'aide d'un formulaire.

FG 04 : Pouvoir contrôler un seul personnage dans le jeu

Un utilisateur possède un personnage propre à lui. Il ne pourra y avoir qu'un seul compte par adresse email.

FG 05 : Respecter l'intégralité du livret de règles de jeu

Chaque règle du cahier ci-dessous devra être respectée.

FG 06 : Le joueur aura la possibilité de rejoindre une des trois équipes du jeu

Les différents joueurs lors de leur inscription pourront choisir une équipe.

FG 07 : Chaque partie doit reboucler tous les mois

Les scores par équipe retombent à zéro. Un compteur affiche combien de temps il reste avant la fin de la partie.

FG 08 : Un classement des joueurs sera affiché sur le site

Un classement des joueurs selon les points qu'ils ont, s'actualisera à chaque fois qu'ils marquent des points.

FG 09 : Un tchat sera ouvert à tous sur la page d'accueil du site (pour les membres connectés)

Chaque utilisateur pourra communiquer avec les autres, en temps réel, afin de rendre le jeu plus dynamique.

FG 10 : Un tchat sera également disponible par équipe, à tout moment du jeu.

Chaque équipe pourra ainsi planifier une stratégie d'équipe, sans que l'autre équipe soit au courant.

Règles du jeu

RJ 01 : Le personnage dispose d'un quota de points d'actions par jour (non cumulable)

RJ 02 : Certaines actions coûtent des points d'actions au joueur.
RJ 03 : Le personnage se déplace de salles en salles sans utiliser de points d'actions, mais des points de déplacement.
RJ 03bis : Chaque salle est différente. Une description sera affichée afin que le joueur s'imagine la salle (description visuelle, probabilité d'être découvert quand il est caché, probabilité de découvrir un objet)
RJ 04 : Le personnage dispose d'une barre de vie. Si elle tombe à zéro... **cf. Mort du joueur**
RJ 05 : Le personnage peut avoir plusieurs objets sur lui (mais en nombre limité par leurs poids)
RJ 06 : Le personnage peut attaquer d'autres joueurs des équipes adverses (**cf. Attaque entre joueurs**)
RJ 07 : Le personnage peut se mettre en "fouille". Dès lors, il à une probabilité de trouver un objet toutes les heures. La probabilité varie en fonction de la salle. Le joueur peut se déconnecter tout en restant en "fouille".
RJ 08 : Chaque équipe dispose d'une salle "de repos", dans laquelle les membres de l'équipe sont intouchables.
RJ 08bis : Les personnages ne peuvent aller dans les salles de repos adverses.
RJ 09 : Chaque jour, des zombies apparaissent aléatoirement dans les salles (sauf salles de repos)
RJ 09bis : Plus le temps passe, plus les zombies seront nombreux.
RJ 10 : Un joueur peut se cacher dans une salle. La probabilité qu'il soit découvert lors d'une fouille des caractéristiques de la salle.
RJ 11 : Les statistiques numériques des ennemis ne seront pas affichées, ainsi que leurs pseudonymes.
RJ 11bis : Seuls les alliés pourront voir leurs informations précises et détaillées.
RJ 12 : Le joueur peut s'équiper d'une arme d'attaque pour augmenter ses points de dégâts.
RJ 12bis : Le joueur peut s'équiper d'une armure pour augmenter ses points de défense.
RJ 13 : Si un joueur se cache dans une salle occupé par des zombies, il a une probabilité de se faire attaquer avant de se cacher et une autre probabilité (plus faible) que sa cache rate.
RJ 14 : Les déplacements dans une salle peuvent être limités à cause du nombre important de zombies. **cf Zombies**

Blessure grave du joueur :

- Il est directement renvoyé à sa base.
- Perte partielle de son inventaire : se retrouvant par terre et disponible dans la salle.
- Si tué par un membre de l'équipe adverse : - 5 points pour son équipe, + 5 points pour l'autre équipe.
- Si tué par un ou plusieurs zombies: -3 points pour son équipe.

Autres :

Zombie: -> Personnages non joueur aléatoire peuplant les salles. Ils ont une barre de vie, des points de défense et des points d'attaque. Ils n'attaquent le joueur QUE si ce dernier fait une action dans la zone (fouiller, attaquer, se cacher (cette dernière est déterminée par un pourcentage)).
-> Lorsque que le nombre de zombies est deux fois supérieur au nombre de joueurs alliés dans la salle, le seul mouvement possible est d'aller de là où l'on vient (exemple : si le joueur a fait un déplacement vers le haut pour rejoindre cette salle, il ne pourra faire un déplacement que vers le bas).

Bibliographie

http://fr.wikipedia.org

http://fr.openclassrooms.com

http://fr.openclassrooms.com

http://nodejs.org

https://addons.mozilla.org

http://commentcamarche.net

http://developpez.com

http://www.createjs.com

http://noframeworks.blogspot.fr/2008/03/une-pratique-rpandue-dans-larchitecture.html

http://docs.mongodb.org/manual/tutorial/getting-started/

http://www.commentcamarche.net/contents/473-cycle-de-vie-d-un-logiciel

http://fr.openclassrooms.com/informatique/cours/gerez-vos-codes-source-avec-git

http://naholyr.fr/2013/12/mise-en-production-serveur-node-js-phusion-passenger/

Source des illustrations

Logo HTML5
http://www.w3.org/html/logo/

Logo CSS3
http://www.indigotheory.com/img/logo_css3.pn

Logo Javascript
http://imgarcade.com/1/javascript-logo/

Schéma Javascript
http://uploads.siteduzero.com/files/292001_293000/292939.png

Logo Node.js
http://www.redbubble.com/shop/free+stickers

Logo MongoDB
http://leonvanderveen.nl/blog/search-engine-for-mongodb

Logo V8
http://fr.openclassrooms.com/informatique/cours/des-applications-ultra-rapides-avec-node-js/pourquoi-node-js-est-il-rapide

Schémas Non-bloquant
http://fr.openclassrooms.com/informatique/cours/des-applications-ultra-rapides-avec-node-js/pourquoi-node-js-est-il-rapide

Logo Git
http://devopsangle.com/2013/05/31/git-1-8-3-releases-with-better-triangular-work-flow-support/

Logo Firebug
http://krushnai.com/technologies.php

Logo CreateJS
http://labs.teresalight.com/presentation-createjs/

Logo Tiled
http://www.mapeditor.org/

Logo Snagit
http://computertotaal.nl/apps-software/transparante-en-schone-screenshots-in-snagit-10-21692

Logo Photoshop
http://leyendasdeprimavera.blogspot.fr/2013/11/photoshop-logo.html

Logo PhotoFiltre
http://machacreation.centerblog.net/37-icone-photofiltre?ii=1

Logo Bootstrap
http://audricmas.olympe.in/

Logo jQuery
http://zenika.github.io/Presentations/20130926_ptidej_angularjs/#/7

Logo Aptana
http://juanchavezwebsite.com/Resume.html

Logo SublimeText
http://fr.wikipedia.org/wiki/Fichier:Sublime_Text_%28logo%29.png

Logo Putty
http://commons.wikimedia.org/wiki/File:PuTTY_icon_128px.png

Logo Sprite Cutter
http://spritecutter.sourceforge.net/

Angry Birds
http://www.infos-mobiles.com/angry-birds/angry-birds-revient-sur-le-marche-et-fait-oublier-flappy-bird/51265

Ogame
http://freegames.askthefurryone.com/home/?page_id=683

Résumé

Zomb'IstiA est un jeu multijoueur en ligne, intégré dans un site web, en temps réel et dans un monde persistant. Sur le site web, les utilisateurs peuvent s'inscrire, se connecter, et jouer. Dans ce jeu, ces derniers peuvent explorer l'IstiA modélisé en 2D, interagir avec beaucoup d'items, fouiller et se cacher. Il y a trois équipes, et le but de chaque équipe est d'accumuler le plus grand nombre de points, en trouvant des objets précieux ou en combattant d'autres joueurs.

Le but de ce projet tuteuré d'EI4 consiste à créer ce site, avec le jeu, en 5 mois. Pour réaliser ce projet, beaucoup de compétences ont été utilisées : développement web, programmation objet, algorithmie et administration linux. Ce fut un challenge, car beaucoup de travail fut nécessaire pour développer cette application web. Les technologies utilisées étaient inconnus par les membres du groupe et il fut nécessaire d'apprendre et comprendre assez vite.

Pour réussir, un développement en cascade fut nécessaire, en découpant chaque tâche : analyse, conception, développement, et tests. De plus, les tâches de développement devaient être intelligemment distribuées entre les développeurs. Chacun avait sa spécialisation : interfaces utilisateurs, traitements ou base de données.

A la fin, Zomb'IstiA a été déployé sur un serveur personnel, connecté à l'Internet et peut être visité par n'importe qui avec son ordinateur et un navigateur web. Les interfaces graphiques ont été retravaillées afin d'être plus attractives et ergonomiques.

Mots Clés

Développement logiciel, application web, temps-réel, jeu multijoueur, programmation événementielle et non bloquante, JavaScript

---

Summary

Zomb'IstiA is an online multiplayer game, integrated into a website, in real time and in a persistent world. On the website, users can sign up, sign in, and play. In this game, users can explore a 2D model of the IstiA, interact with a lot of items, search and hide. There are three teams, and the aim of each one of them is to accumulate the largest number of points, in finding precious objects or fighting other players.

The aim of this EI4 mentored project consists in creating this website with the game, in 5 months. To carry out this project, lots of skills were used: web development, object programming, algorithmic and Linux Administration. It was a challenge, because a lot of work was required to develop this web application. Technologies used were unknown by the members of the group and it was necessary to learn and understand quickly.

To succeed, a waterfall development was necessary, by cutting each task: analyze, conception, development and application tests. Further, development tasks had to be intelligently distributed between the developers. Everyone had its specialization: user interfaces, process, or database.

At the end, Zomb'IstiA was deployed on a personal server, connected on the Internet and can be visited by anybody with his computer and a web browser. Graphical interfaces have being reworked to be more attractive and ergonomic.

Keywords

Software development, Web application, real-time, multiplayer game, event and non-blocking programming, JavaScript.