Le Mans Université

Institut Informatique Claude Chappe

Rapport de Projet

World Of Dungeons

Par Jasmin Galbrun, Mathieu Vaudeleau, Maxence Despres et Tibane Galbrun

19 avril 2019

`https://github.com/Checkam/WorldOfDungeons`

Sommaire

1 Introduction **3**

2 Analyse et Conception **4**
2.1 Fonctionnalités 4
2.2 Choix des solutions 5

3 Organisation du travail **6**

4 Codage, méthodes et outils **7**
4.1 Mise en place des outils 7
4.1.1 Makefile 7
4.1.2 Gestion des erreurs 7
4.1.3 Chemin d'accès 7
4.1.4 Création des textures (SDL) 8
4.1.5 Listes pseudo-génériques 8
4.2 Enregistrement des données 9
4.2.1 Fichier JSON 9
4.2.2 Fichier binaire 9
4.3 Gestion des évènements 10
4.4 Menu 10
4.5 Bloc et items 11
4.5.1 Bloc 11
4.5.2 Item 11
4.5.3 Inventaire 11
4.6 Entité 12
4.6.1 Définition 12
4.6.2 Positionnement 13
4.6.3 Sprite 14
4.7 Map 14
4.7.1 Définition 14
4.7.2 Génération d'une map 15
4.7.3 Sauvegarde et chargement d'une map 15
4.7.4 Affichage d'une map 15
4.8 Donjon 15

5 Résultats **17**

6 Conclusion **18**

7 Annexes **19**

1 Introduction

Pour notre projet, nous voulions faire un jeu de type "Bac à sable", c'est-à-dire qu'il n'y a pas d'objectifs précis imposés au joueur. Il peut ainsi faire ce qu'il veut dans un monde mis à sa disposition. Pour ce faire nous nous sommes inspirés de deux jeux : Minecraft[2] et Terraria[8] tous deux de type "Bac à sable" (cf Figures 1 et 2).
Nous avons donc imaginé **World Of Dungeons**, un jeu ayant pour principal objectif la survie et l'exploration dans un monde presque infini, où le joueur pourra interagir avec son environnement et combattre divers monstres.

FIGURE 1 – Minecraft

FIGURE 2 – Terraria

2 Analyse et Conception

Après avoir défini l'objectif général du projet, nous allons détailler les différentes fonctionnalités permettant de répondre à la problématique de départ ainsi que les solutions techniques apportées.

2.1 Fonctionnalités

Notre objectif est que chaque joueur puisse avoir une partie différente (cf. Figure 3). Mais si deux joueurs, sur deux machines différentes, veulent jouer sur le même monde, c'est possible (ils auront la même génération, avec chacun leur propre carte mais ils ne peuvent pas se rencontrer, ce n'est pas du multijoueur). (cf. Figure 4)

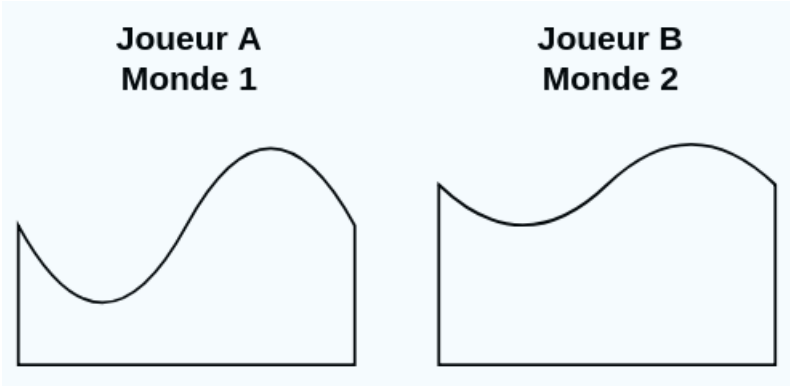

FIGURE 3 – Deux mondes différents pour deux joueurs différents

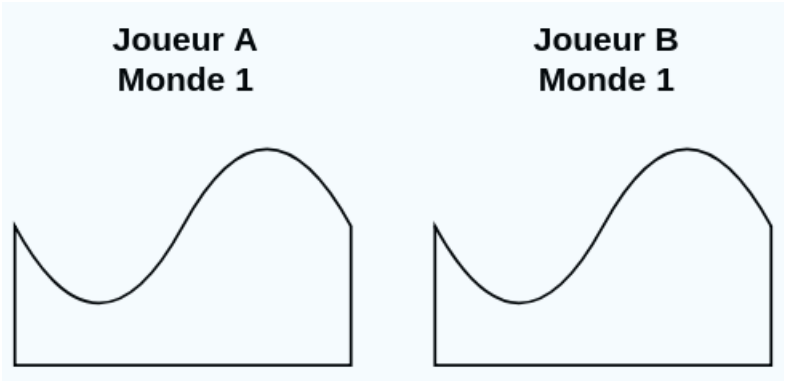

FIGURE 4 – Deux mondes identiques pour deux joueurs différents

Le joueur après avoir générer une carte doit également pouvoir explorer, construire, casser, combattre et compléter des donjons. Mais aussi enregistrer, supprimer et créer sa partie.

2.2 Choix des solutions

Pour répondre au mieux à nos objectifs, nous avons cherché, testé et choisis différentes solutions.
Pour la génération des mondes, nous avons opter pour de la génération procédurale. Mais un problème s'est posé, quel algorithme utiliser ? Après divers tests, nous avons retenu l'algorithme du "bruit de Perlin"[9] dont nous parlerons par la suite.
Pour les donjons, nous avons décidé que leurs accès se feraient par des portails. Cela permet de moins charger le monde de base et de faciliter la création de ces derniers. L'algorithme de génération est détaillé plus bas dans la section 4.8.
Pour l'affichage graphique, nous avons choisis la librairie Simple Directmedia Layer 2 (SDL2)[7].
Afin d'enregistrer les différentes données du jeu, nous avons utilisé les fichiers JSON et binaires qui sont expliqués dans la section 4.2.
Pour mener à bien notre projet nous avons utilisé l'arborescence suivante.

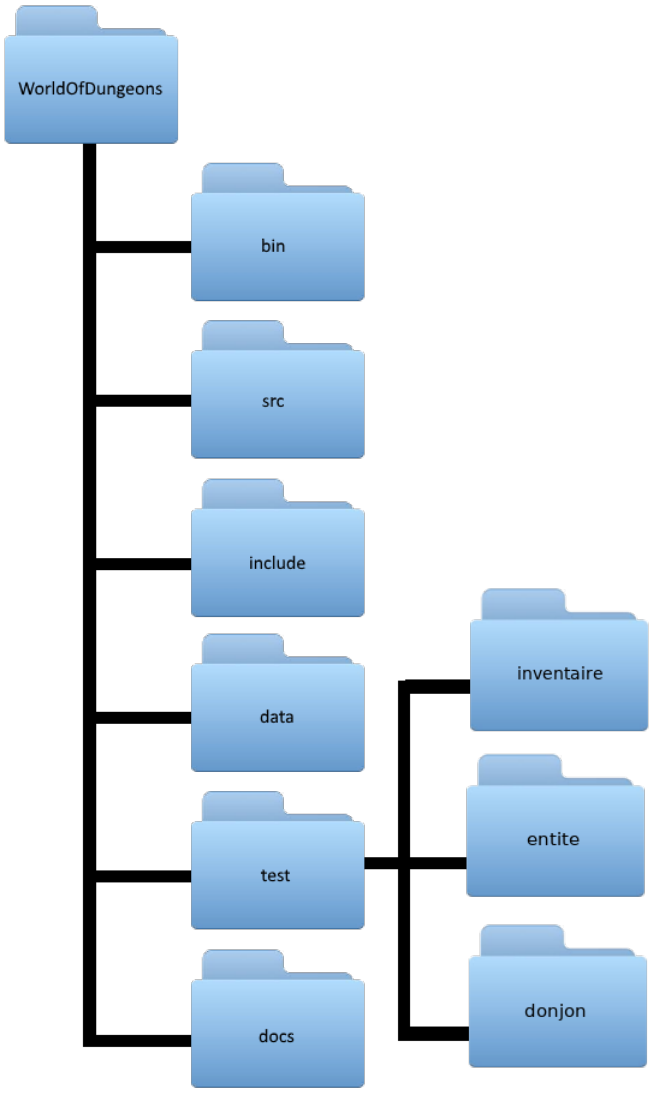

FIGURE 5 – Arborescence du projet

3 Organisation du travail

Pour organiser notre travail au mieux, nous avons définis toutes les tâches qui devaient être réalisées [3]. Chaque tâche s'est vue attribuée une priorité. Puis elles ont été distribuées par préférence aux différents membres du groupe :

- Jasmin :

→ le menu

→ les donjons

→ l'Intelligence Artificielle (IA)

- Tibane :

→ les listes pseudo-génériques

→ l'enregistrement de données

→ les entités

- Maxence :

→ les blocs

→ la map (Génération,Affichage,Stockage)

- Mathieu :

→ la gestion des évènements (entrées clavier souris)

→ les items

→ les inventaires

Après avoir attribué les différentes tâches aux membres du groupe, nous avons définis les outils de travail en commun dont nous allions avoir besoin.

En effet, nous avons utilisé Git et GitHub pour stocker et partager notre code au sein du groupe mais également pour le gestionnaire de version. De plus, pour tout ce qui est autre chose que du code, nous avons utilisé Google Drive.
Afin de communiquer, nous nous sommes servis de Discord et de Messenger.

4 Codage, méthodes et outils

Lors de cette partie, nous verrons les différents outils, algorithmes et méthodes utilisés pour mettre en place les fonctionnalités et solutions vues précédemment.

4.1 Mise en place des outils

Pour simplifier la création de chaque module, nous avons mis en place différents outils.

4.1.1 Makefile

Pour la compilation de notre projet un problème est vite apparu, la création de test nous obligeais de créer une copie du github et de modifier le main. Pour palier a ce problème et garder tout les tests créer depuis le début du projet, 2 makefile on été créer :
un pour les tests.
un makefile principal.
Grâce au makefile principal nous pouvons exécuter la compilation de tout les tests grâce a la commande make test qui cherche tout les makefile du dossier test (Figure 5 - Arborescence du projet) et les executes.

4.1.2 Gestion des erreurs

Afin que tout le groupe puisse comprendre au mieux les codes de retour de chaque fonction, nous avons créé un module pour gérer les erreurs. Celui-ci comprend deux fonctions de débogage ainsi que les différents codes erreurs.
La première fonction sert à enregistrer les erreurs dans un dossier log, ce qui permet de garder une trace des erreurs.
La dernière fonction permet quand à elle d'afficher les erreurs dans la sortie erreur du terminal.

4.1.3 Chemin d'accès

Durant la réalisation des différentes tâches de notre projet, un problème est apparu, les chemins d'accès aux différents fichiers.
En effet, suivant l'endroit où nous exécutions le code, les fichiers ne chargeaient pas forcément. Le problème était les chemins d'accès qui n'étaient pas absolus.
Pour cela, nous avons créé un module permettant de créer des chemins absolus vers les fichiers voulus pour que notre code marche peut importe l'endroit où nous l'exécutons.
Pour se faire, nous avons utilisé les arguments du "main", qui nous permettent de récupérer deux valeurs importantes et qui sont, le chemin du programme et le chemin absolu de l'endroit où nous sommes sur la machine lors de l'exécution.
Ces deux valeurs nous permettent de créer le chemin absolu de la racine de notre projet. Cela nous permet, par la suite, via une fonction, de créer d'autres chemins absolus vers les fichiers voulus.

4.1.4 Création des textures (SDL)

Le processus de création d'une texture en SDL est, selon la donnée, relativement long et répétitif (cf. Figure 6). Nous avons donc mis en place deux fonctions pour simplifier cela. Une fonction de création de texture à partir d'une image et une autre à partir d'un texte.

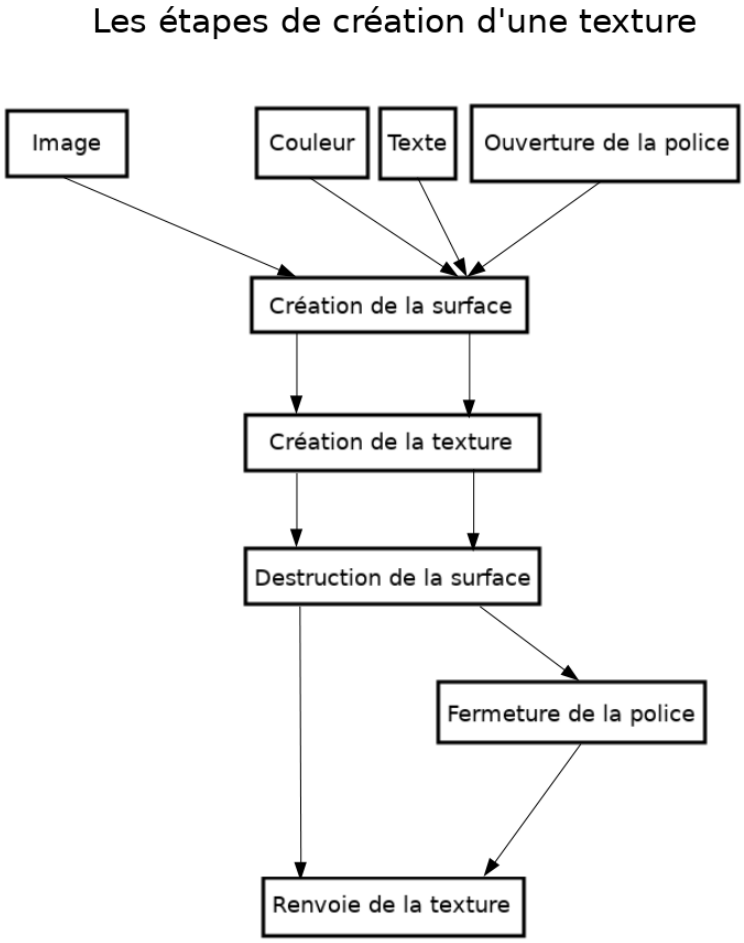

FIGURE 6 – Les étapes de création d'une texture

4.1.5 Listes pseudo-génériques

Pendant le pré-projet, quand nous testions la génération procédurale, nous avons eu besoin de liste. Nous est donc venu l'idée de créer des listes pseudo-génériques pour pouvoir stocker ce que l'on veut en fonction des besoins du projet.
Le "pseudo-générique" vient du fait que l'on ne stock pas l'objet avec des pointeurs sur des fonctions d'affichage, de suppression, ..., mais juste un pointeur sur l'objet. Il est donc difficile de faire des listes hétérogènes, mais son utilisation est plus simple.

4.2 Enregistrement des données

Notre projet étant un jeu de survie, il nous fallait pouvoir enregistrer ces mondes, ainsi que tous ce qui leurs est lié (statistiques du joueur, caractéristiques du monde, ...).

4.2.1 Fichier JSON

Le premier format de fichier dont nous allons parler est le JSON que nous avons choisis pour la sauvegarde des données qui ne prennent pas beaucoup d'espace disque et qui ne nécessitent pas un temps d'accès rapide.
Pour le codage, nous avons choisis de créer notre propre module d'enregistrement au format JSON. Cela nous permet de comprendre la difficulté qu'est l'implémentation de ce format ainsi que de pouvoir, au besoin, rajouter des types de valeurs possible à enregistrer.
Pour effectuer l'implémentation, nous sommes allés regarder les normes actuelles sur le format JSON qui sont la RFC-4627[5] et la ECMA-404[1], toutes deux en concurrences. Mais pour nous simplifier la tâches, nous avons seulement codé un très petit bout de ces normes, car nous n'avions pas besoin de tout, et il fallait que le module reste simple à utiliser et à coder.
Nous avons donc utilisé la syntaxe suivante pour l'enregistrement (cf Figure 7), qui est qu'un objet est définit par un crochet ouvrant et fermant, qui contient des "clé :valeur" séparées par des virgules. On peut évidemment mettre autant d'objet que l'on veut dans le fichier.

```
{clé:valeur,clé:valeur}
```

FIGURE 7 – Un objet JSON contenant deux clés, valeurs

Pour les valeurs, il y a deux types d'enregistrements, le premier est la chaîne de caractères qui doit être encadrée de guillemets (cf Figure 8), et le second est le nombre, qu'il soit à virgule ou non, qui n'est pas entouré de guillemets (cf Figure 9).

```
{"chaîne":"chaîne de caractères"}
```

FIGURE 8 – Enregistrement de chaîne de caractère au format JSON

```
{"nombre 1":75.00045}
{"nombre 2":3476776}
```

FIGURE 9 – Enregistrement de nombre au format JSON

4.2.2 Fichier binaire

Le deuxième type de fichier que nous avons pris est le binaire, qui a été utilisé pour le stockage du monde, car les données étaient nombreuses et nous avions besoin d'un temps d'accès en lecture et en écriture très rapide. Les fichiers binaires répondent à ces besoins car ils sont non formaté et facile à mettre en place, et du fait qu'ils sont non formatés, ils prennent très peu de place en mémoire.

4.3 Gestion des évènements

Dans ce module, le but est de simplifier le plus possible l'utilisation des touches dans le programme, et que le joueur via un menu option puisse changer la configuration des touches comme il le désire (de se diriger avec les flèches plutôt qu'avec zqsd par exemple).
Le principale problème rencontré est du fait que la SDL ne récupère uniquement les fronts descendent et ascendant des touches, c'est à dire lorsque une touche est appuyé, la SDL envoie un signal que l'on peut facilement récupérer, mais si l'utilisateur continue d'appuyer sur cette touche sans la relâché, la SDL n'envoie pas de signal, elle renvoie un signal uniquement lorsque la touche est relâché, il est donc impossible de savoir si une touche est appuyé on non. Pour remédier a ce problème et le fait que l'utilisateur puissent changer sa configuration de touche, on créer deux tableau : un tableau de booléen contenant pour chaque touche 0 si la touche est relâché ou 1 si elle est appuyé, un autre tableau contenant le code des touches SDL [6] choisis par l'utilisateur. Pour que le tableau de boolen soit mis à jour, on compare les signaux SDL avec le second tableau.
Il y a cependant un problème avec cette méthode : si le joueur, pendant qu'une touche est appuyé perd le focus de la fenêtre, et pendant que le focus est perdu il relâche la touche alors le relâchement de la touche ne sera pas capté par la SDL donc la touche restera appuyé pour le jeu même si en réalité la touche n'est plus appuyé. L'une des solution serait de mettre en pause le jeu si le focus est perdu, mais nous n'avons pas trouvé de telle fonction.

4.4 Menu

Pour pouvoir naviguer à travers le jeu, nous avons mis en place un module permettant de créer des menus facilement. Un menu est défini selon certains critères simples (cf. Figure 11) qui sont une liste de boutons ainsi qu'un titre. La structure du menu se veut simpliste pour que le jeu soit réalisé dans le temps imparti. En effet la fonction de gestion du menu renvoit le bouton sur lequel on a cliqué permettant ainsi de gérer chaque action que l'utilisateur effectue.

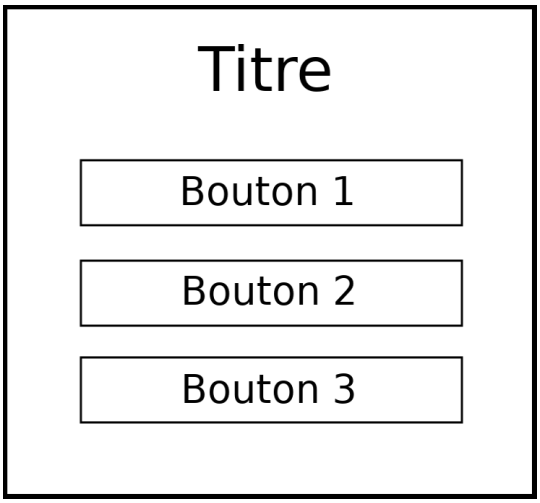

Figure 10 – Schéma simplifié d'un menu

4.5 Bloc et items

Ces deux modules sont intimement liées puisque un bloc est reliés a un ou plusieurs items (une feuille d'arbre peut donner une pomme et une pousse d'arbre), et un item est reliés à un ou zéro bloc.
Nous avons été deux a travaillé sur ces modules (Maxence module bloc et Mathieu module item et inventaire) Il a donc fallu très souvent échanger afin de bien comprendre ce que l'autre faisait et de ne pas s'égarer.

4.5.1 Bloc

Le module bloc a été l'un des premier module créer. Il est a la base même du jeu (on se déplace sur des blocs, on casse et pose des blocs). Chaque bloc a des caractéristique comme son matériau, ça texture, ça difficulté a être cassé ou si il est incassable. Les matériaux son une énumération de tout les types de blocs (terre,roche,bois, etc).

4.5.2 Item

Le but du module item est de pouvoir après avoir cassé un bloc, récupérer ces items dans l'inventaire (et également qu'il tombe par terre si l'inventaire est plein) pour ensuite pouvoir fabriquer d'autres items (pièces d'armure, épées ...) en passant par des étapes de fabrication (le minerais de fer doit être fondu dans un four avant d'être utilisé). le joueur peut également construire grâce a des items ce qu'il veut (une maison, une mine ...) ayant pour seule limite son imagination.

4.5.3 Inventaire

Le module inventaire est étroitement liée au module item car la majeur partie de ce module est de pouvoir stocké des items dans des structures de données.
la plupart des entités peuvent avoir un inventaire (les ennemis peuvent avoir des armes et armures, et lorsqu'ils meurent ils peuvent faire tomber des objets que le joueur peut récupérer, le joueur a également un inventaire), ou même certains bloc ont des inventaires (comme les coffres).
Le module inventaire ce concentre sur deux points :

- structurer les données
- affichage des inventaires

Structuration des données :
Le but ici est de ne pas stocké autant de fois un item qu'il est présent dans des inventaires. Par exemple si on a deux coffres qui ont tous deux a l'intérieur un item de terre chacun, il ne faut pas que cette item soit sauvegardé deux fois, pour deux items ce n'est pas grave mais si c'est mille fois que cette item est "dupliqué", cela peut provoqué un mauvais fonctionnement du jeu.
Pour éviter cela, dans item.h il y a un tableau déclaré de manière global afin que tous les modules puissent y avoir accès, contenant tous les items du jeu avec toutes leurs spécifications. Tous les inventaires stockeront désormais uniquement un pointeur pointant sur l'item désirer dans ce tableau. si l'item peut être posé, alors il contient une variable

contenant le numéro du bloc au quelle est rattaché l'item, si l'item ne peut pas être posé (pièce d'armure, pomme ...) alors cette valeur vaut zéro (dans block.h cette valeur appartient au "bloc" d'air).

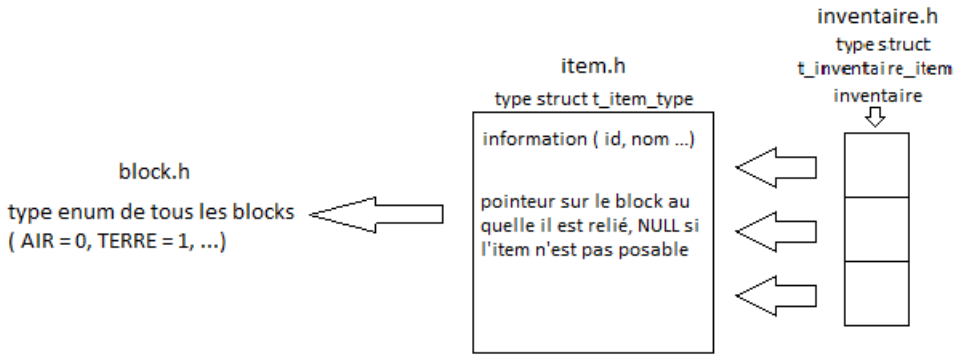

FIGURE 11 – organisation bloc/item/inventaire

4.6 Entité

Maintenant, nous allons voir un module important du projet, il s'agit du module entité. En effet, pour que notre jeu soit jouable, il y a besoin que le joueur puisse contrôler une entité qui est capable d'interagir avec son environnement ainsi qu'avec d'autres entités.

4.6.1 Définition

Tout d'abord, définissons ce qu'est une entité. Une entité est une structure composée d'un certain nombre de caractéristiques telles que sa position, sa taille, etc (cf Figure 12). Tout cela nous permettra, par la suite, de simplifier le codage.

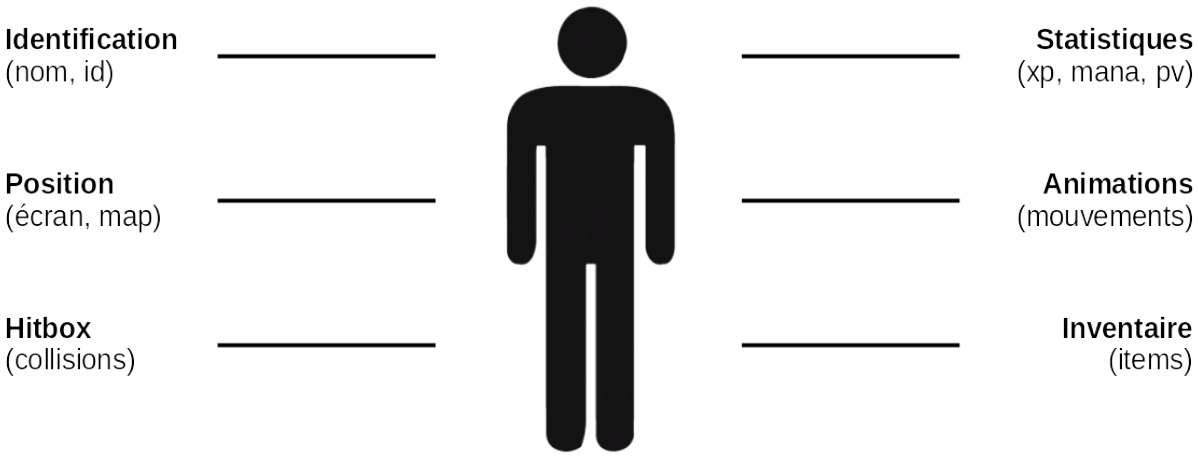

FIGURE 12 – Définition d'une entité

Après avoir définit ce qu'est une entité, nous allons nous attarder sur les deux grosses difficultés rencontrées lors du codage de celle-ci qui sont, son positionnement et ses animations.

4.6.2 Positionnement

En effet, la première grosse difficulté rencontrée est le positionnement d'une entité, quelle soit sur la fenêtre ou sur la map, car toutes deux possèdent un système de coordonnée différent ainsi que des coordonnées différentes (cf Figure 13).

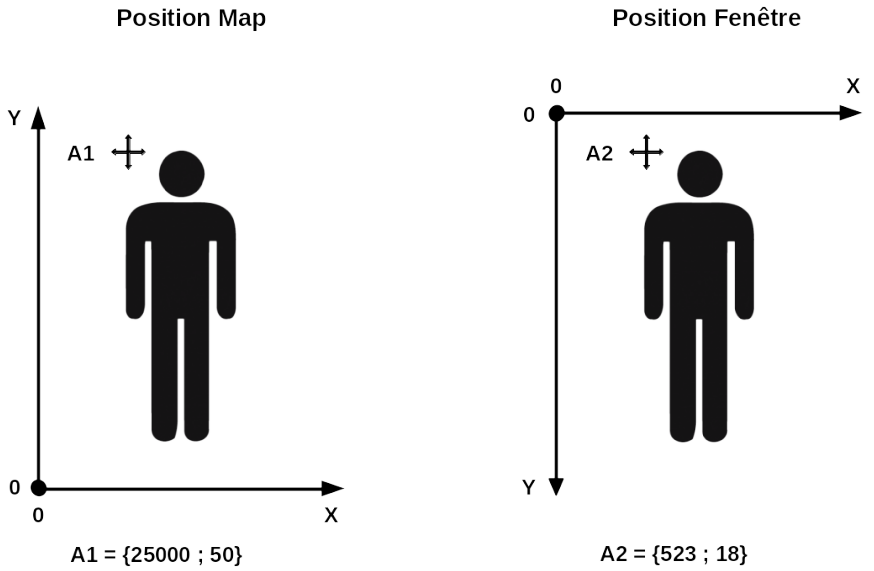

FIGURE 13 – Position d'une entité sur la map et sur la fenêtre

Par exemple, si le joueur avance sur la droite, le "x" de la map va augmenter alors que celui de la fenêtre va rester fixe (cf Figure 14).

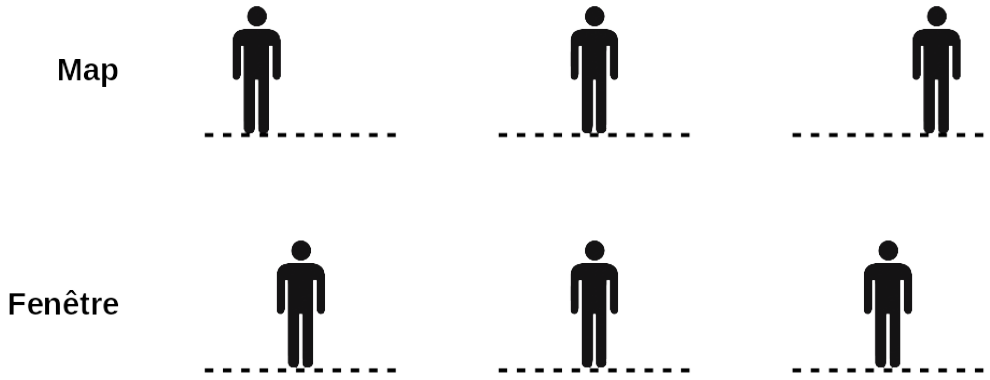

FIGURE 14 – Mouvement en "x" d'une entité

Autre exemple, si le joueur saute, le "y" de la map va augmenter de même que celui de la fenêtre mais va être bloquer durant la descente après être revenu à son "y" initial pendant que le "y" de la map continue à diminuer (cf Figure 15).

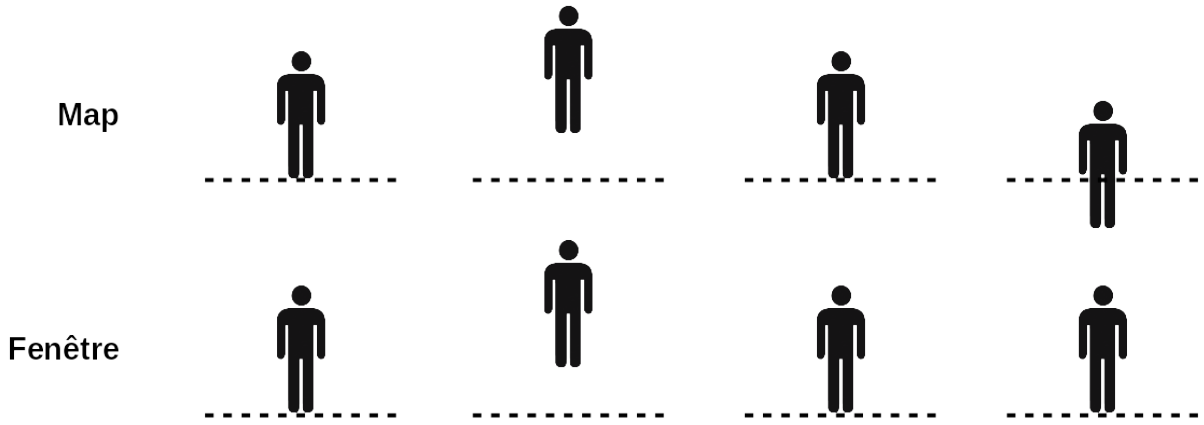

FIGURE 15 – Mouvement en "y" d'une entité lors d'un saut

Pour répondre aux différents problèmes vus ci-dessus, nous avons créer deux rectangles, l'un contient les coordonnées du joueur sur la map, tandis que l'autre contient les coordonnées du joueur sur la fenêtre (écran).

4.6.3 Sprite

Désormais, nous allons parler de la deuxième grosse difficulté rencontrée. Pour que notre jeu ne soit pas plat et peu vivant, nous avons décidé de rajouter des animations aux entités lorsqu'elles font un mouvement. Pour cela, nous avons utilisé des "sprites" (cf Annexe 1).
Un sprite contient sur chaque ligne une action qui est décomposé de manière grossière en plusieurs images. L'animation d'un mouvement consiste à afficher, les unes à la suite des autres, ces différentes images. La difficulté a été de faire un système qui gère cet affichage d'image.
Pour cela, nous avons créé une structure qui contient pour chaque action, son numéro de ligne dans le sprite, le temps qu'il faut attendre entre l'affichage de chaque image ainsi que le nom de l'action. Nous avons également ajouté, dans les paramètres de l'entité, tous ce qui nous permet de savoir où nous en sommes dans l'animation de l'action en cours.

4.7 Map

Maintenant que nous avons vu le joueur et les blocs nous allons nous intéressé a la structure d'une map.

4.7.1 Définition

Premièrement , définissons la structure d'une map. Chaque map a un nom choisi par l'utilisateur lors de la création qui permettra de les différencier. Une map a aussi un seed, permettant sa génération. Enfin, il nous fallait un moyen de représenter facilement une map. Le moyen le plus simple trouver est une liste de pointeur sur des tableaux de bloc (un tableaux correspond a une colonne de la map). On a aussi intégré une entité (le joueur) dans la structure d'une map.

4.7.2 Génération d'une map

Nous nous sommes intéressé à la génération procédurale qui permettait de créer une infinité de map et qu'elles soient les plus grandes possible. On souhaitait aussi, pour des joueurs différents, sur des machines différentes et pour un seed donné, que notre programme génère la même map.

Nous avons donc choisi d'utiliser l'algorithme de génération procédurale appelé le bruit de Perlin qui nous permettait de créer ce que nous souhaitions. Cette algorithme a été utilisé pour chacunes des 4 étapes de la génération d'une map qui sont :

- Génération des hauteurs
- Génération des minerais
- Génération des grottes
- Génération des structures

A noté que la génération de la map est centré sur le joueur, on ne génère pas toute la map, seulement ce que le joueurs découvre.

4.7.3 Sauvegarde et chargement d'une map

Chaque map est sauvegardée dans un dossier, du nom de celle-ci, qui contient un json contenant les metadonnées de la map (seed, nom, date), on a dans un sous-dossier, toutes les informations sur le joueur de cette map et finalement la map sauvegarder dans un fichier binaire.

4.7.4 Affichage d'une map

L'affichage de la map a été une tache complexe pour centré correctement l'affichage avec le joueurs pour la partie SDL. Pour l'affichage terminal nous avons utilisé les couleurs que celui-ci nous offre pour représenté plus simplement les différents type de blocs.

4.8 Donjon

L'une des fonctionnalités du jeu est d'avoir des donjons, nous avons donc mis en place un système pour faire cela.
En effet nous avons opter pour une génération procédurale du donjon. Dans ce cas-ci la génération dépend du seed et de la position du joueur dans le monde. La seule chose qui n'est pas procédurale est l'apparition des monstres.
L'algorithme de génération se décompose en 2 grandes parties qui sont la génération de la structure du donjon et la génération de la structure de chaque salle. Pour générer la structure du donjon, on part du principe que chaque salle est collée l'une à l'autre et qu'une salle possède deux coordonnées, une composante en x et en y. L'algorithme utilisé est le suivant :

1. Créer une liste
2. Si la liste est vide :
   - créer une salle au centre du donjon
   - ajouter cette salle à la liste
3. Sinon :
   - sélectionner une salle, dans la liste, qui a 3 voisins ou moins
   - sélectionner une cellule adjacente, à cette salle, qui n'est pas une salle
   - créer une salle au coordonnée de cette cellule
   - ajouter la nouvelle salle à la liste
4. Répéter l'étape 3 tant que l'on n'a pas atteint le nombre de salle désirer

Cet algorithme peut être appliquer à plusieurs dimensions en changeant quelques paramètres. Pour rendre cet algorithme procédural, nous avons ajouté "Perlin Noise" aux deux phases de sélection. Ainsi on remplace la sélection aléatoire des salles par du pseudo-aléatoire.
De plus, une salle possède une structure(cf. Figure 16). Celle-ci est générée en fonction des voisins de cette salle ce qui permet de lier les différentes salles entre elles. Le plafond de la salle est également généré avec "Perlin Noise". Cela ajoute du relief tout en restant procédural. Dans un donjon il faut un début et une fin. Dans notre cas, nous avons choisi de dire que la première salle créée est la salle de départ et que la salle de fin est celle la plus éloignée de la salle de départ. Les autres salles sont considérées comme intermédiaire. Enfin une salle possède également une difficulté. La salle de départ n'a pas de difficulté et la salle de fin a comme difficulté "final". Les autres salles ont une difficulté aléatoire entre facile, moyen et difficile. Le type et la difficulté sont stockés dans les données de chaque salle.
Pour conclure sur ce module, le donjon se joue en salle par salle. C'est à dire que le joueur ne voit qu'une seule salle à chaque fois. En effet l'affichage n'est donc pas centré sur le joueur contrairement au monde. Pour faire cela nous avons créé une entité de référence au centre de la salle qui est invisible et avec laquelle on ne peut pas interagir.

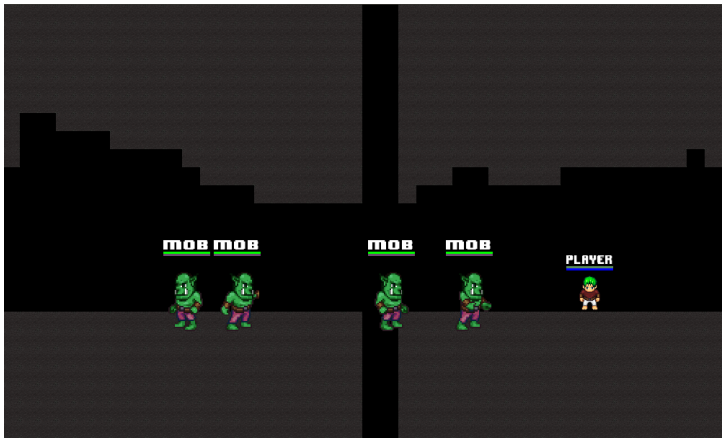

FIGURE 16 – Salle de donjon (SDL)

5 Résultats

Au départ nous avions prévu un affichage terminal, cependant plusieurs choses était très compliqué a réaliser dans un terminal comme l'affichage des blocs, casser un bloc (en SDL on le fait avec la souris, or en terminal il n'y a pas possibilité d'avoir une souris), l'affichage d'un inventaire et bien d'autre chose. Nous avons donc décider de nous concentrer exclusivement sur l'affichage graphique.
Ce qui résulte de ce travail est un jeu, World Of Dungeons dans lequel un joueur peut créer autant de map qu'il le souhaite. Il pourra évoluer dans chacune de ces map, casser, poser des blocs sur son chemin et vaincre des donjons. Les blocs qu'il casse seront envoyer dans son inventaire.
Certaine fonctionnalité tel que l'intelligence artificiel des mobs ou la génération des structures peuvent être améliorer d'autre reste encore a implémenté tel qu'un système de combat et ajout d'un choix de difficulté lors de la création de la map. Il pourra également a l'avenir créer des armes et armures, jouer en réseau.

FIGURE 17 – Etat actuel de notre jeu

6 Conclusion

Dans l'ensemble nous sommes très satisfait de ce que nous avons réalisé jusqu'à maintenant et ce projet à été très enrichissant pour nous tous même si nous nous attendions à faire plus (Nous avions vu large. Nous avons finalement manqué de temps pour réaliser le système de fabrication d'items, le mode client/serveur et finalisé certains points ...).
De plus, cela nous a permis de mettre en pratique nos connaissance en langage C et nous a également permis d'apprendre a utilisé une API tel que SDL.
Ce projet nous a également permis de découvrir des outils tel que GDB, Git ou Makefile, et de découvrir les problèmes liées au travail à plusieurs (bien se synchroniser afin de ne pas attendre la fin du module de quelqu'un pour pouvoir continuer le sien, les conflits lors de merge sur Github, ...). Il a fallu également travailler notre communication pour s'assurer que à tout instant, tout le monde ait compris la même chose. Certains problèmes de communication nous ont retarder dans le développement le projet.
Nous nous sommes tous mis d'accord pour continuer le jeu jusqu'à obtenir quelque chose qui nous convient à tous, ce qui n'est pas le cas actuellement.

7 Annexes

7.1 Exemple de sprite

7.2 Exemple de débogage

```
==3905== HEAP SUMMARY:
==3905==     in use at exit: 24 bytes in 1 blocks
==3905==   total heap usage: 5 allocs, 4 frees, 4,192 bytes allocated
==3905==
==3905== Searching for pointers to 1 not-freed blocks
==3905== Checked 254,456 bytes
==3905==
==3905== 24 bytes in 1 blocks are definitely lost in loss record 1 of 1
==3905==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==3905==    by 0x10DB75: init_liste (liste.c:20)
==3905==    by 0x118686: main (test_liste.c:29)
==3905==
==3905== LEAK SUMMARY:
==3905==    definitely lost: 24 bytes in 1 blocks
==3905==    indirectly lost: 0 bytes in 0 blocks
==3905==      possibly lost: 0 bytes in 0 blocks
==3905==    still reachable: 0 bytes in 0 blocks
==3905==         suppressed: 0 bytes in 0 blocks
==3905==
==3905== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
==3905== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Ce résultat m'indique que j'ai fait un malloc à la ligne 20 qui n'a pas été free.
Après correction de l'erreur :

```
==4420== HEAP SUMMARY:
==4420==     in use at exit: 0 bytes in 0 blocks
==4420==   total heap usage: 5 allocs, 5 frees, 1,120 bytes allocated
==4420==
==4420== All heap blocks were freed -- no leaks are possible
==4420==
==4420== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==4420== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Références

[1] ECMA-404. https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[2] Minecraft | Site Officiel. https://www.minecraft.net/fr-fr/.

[3] Organisation des tâches. https://github.com/Checkam/WorldOfDungeons/projects.

[4] Perlin Noise in C. https://gist.github.com/nowl/828013.

[5] RFC-4627. https://www.ietf.org/rfc/rfc4627.txt.

[6] SDL touches keycode. https://wiki.libsdl.org/SDL_Keycode.

[7] SDL2. https://www.libsdl.org/.

[8] Terraria. https://terraria.org/.

[9] Cochoy, J. Introduction au bruit de Perlin. https://cochoy-jeremy.developpez.com/tutoriels/2d/introduction-bruit-perlin/. (10 février 2011).