UNIVERSITÉ DE NAMUR

Programmation 2 : projet *Jumping Bananas* (Hadrien BAILLY)

IHDCB143

Programmation 2

Introduction

Objectif

Ce projet a pour visée le développement et la mise en pratique des concepts vus dans le cadre des cours de programmation de première année de baccalauréat.

Il s'agira ainsi de construire et développer un programme complexe en langage C, en proposant de la programmation procédurale/événementielle, de la gestion d'affichage via OpenGL (en autre par scrolling) et des flux d'entrées et sorties vers des fichiers textes et picturaux.

Outre l'implémentation du code, le projet s'accompagne du présent rapport ainsi que d'une présentation.

Contexte

Ce rapport présente le processus de conception et d'implémentation d'un projet inspiré du jeu *Jumping Bananas* de Agame : un jeu de type *plateformer* durant lequel le joueur doit collecter des objets tout en évitant le contact avec des ennemis.

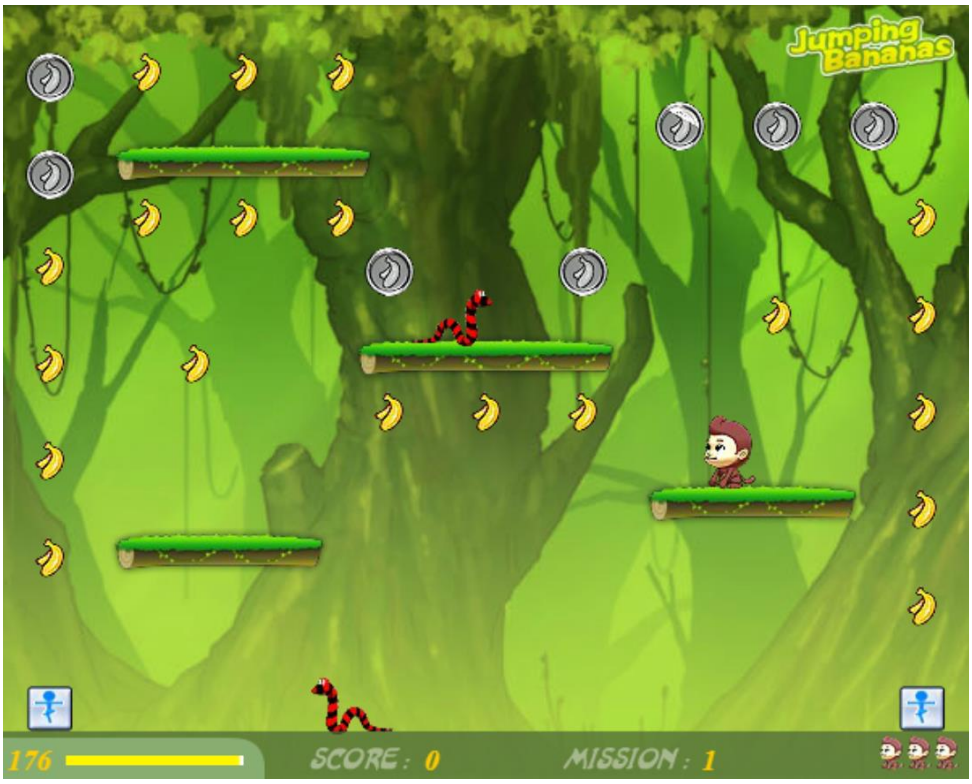

Figure 1 *Capture du jeu Jumping Bananas (JB)*

Jumping Bananas

Une partie typique de *Jumping Bananas* prend la forme d'une succession de niveaux (appelés *missions*) au sein de chacun desquels le joueur doit collecter l'ensemble des bananes et pièces avant la fin d'un délai imparti de 180 secondes. Le joueur incarne ainsi un singe pouvant se déplacer horizontalement et sauter pour atteindre des plateformes et récolter les précieux objets.

Figure 2 *Avatar (JP)*

La récupération des objets et le temps restant imparti alimentent un total qui permet au joueur de viser un meilleur score au fil des différents niveaux réalisés.

Figure 3 *Score items (JB)*

À côté des pièces et bananes, le joueur pourra également trouver différents objets pour l'aider dans son aventure : double saut, vol et survol, invulnérabilité, … Ces objets, que l'on appellera *power-up*, ne rapportent pas de points pour le score mais augmentent temporairement les capacités de l'avatar. Il existe également un objet permettant de restaurer un point de vie.

Figure 4 *Power-ups (JB)*

Figure 5 *Life-up (JB)*

Sur son chemin, il pourra rencontrer différents ennemis (serpents, araignées, hérissons, …) qui tenteront de le rejoindre pour l'attaquer : au moindre contact, le joueur perd un point de vie et est renvoyé à son point de départ. S'il perd l'ensemble de ces points de vie (trois dans la version originale), la partie est terminée. Ne possédant pas de capacité offensive, le joueur n'a d'autre choix que d'éviter les ennemis en sautant au-dessus ou en restant le plus loin possible de ceux-ci.

Figure 6 *Enemies (JB)*

Outre la perte de vie, la fin du chronomètre provoque également la fin de la partie, si le joueur n'a pas su collecter tous les objets dans le temps imparti.

Figure 7 *Game Over (JB)*

Le joueur est alors amené devant l'écran des meilleurs scores.

Contraintes

Au-delà de la recréation du jeu *Jumping Bananas*, ce projet doit répondre aux contraintes suivantes :

- Le joueur doit pouvoir choisir un niveau de difficulté.
- Le joueur doit pouvoir choisir un jeu à défilement (*scrolling*) horizontal ou vertical.
- Le joueur doit pouvoir suivre son état :
  - Barre de vie
  - Score selon le nombre de points acquis et d'ennemis vaincus
  - Temps écoulé
- Le joueur doit pouvoir se situer sur un tableau de meilleurs scores.
- Le joueur doit pouvoir bouger et sauter.
- Le joueur doit pouvoir attaquer ses ennemis au moyen de projectiles.
- Les ennemis touchés par un projectile doivent se changer en bulles temporairement.
- Le joueur doit pouvoir se servir de ces bulles comme plateforme ou les convertir en points.
- Les ennemis doivent se déplacer de façon autonome.
- Lorsque le joueur est touché par un ennemi, il doit perdre un point de vie et mourir si son solde de points atteint zéro.
- Si le temps imparti atteint zéro, les ennemis doivent entrer dans un état *agressif*, c'est-à-dire prendre une teinte rouge et se déplacer plus rapidement.
- Le jeu doit se terminer lorsque le joueur n'a plus de vie.

Outils de développement

Pour réaliser le jeu, les outils suivants doivent être utilisés :

- OpenGL et la librarie GLUT.
- Git et GitHub.
- Le langage de programmation C.
- Les systèmes d'exploitation Linux ou Mac.

Le programme doit répondre à la norme ANSI C1999 et ainsi pouvoir être compilé sans erreur au moyen d'un makefile comprenant la commande `gcc -Wall -std=c99 *.c -o jumpingBananas`. Le programme doit en outre être structuré en modules pour faciliter sa compréhension et son usage, selon les bonnes pratiques de programmation.

Le code du projet devra être posté sur Github : UNamurCSFaculty/1819_IHDCB132_Bailly_Hadrien.

Échéances

Phase 1 : Analyse et spécification

Phase 2 : Ecriture du programme

Phase 3 : Amélioration de la stratégie automatique de jeu

Jumping Bananas

Conception

Cette implémentation diffère partiellement du jeu *Jumping Bananas* original et suit les règles suivantes : le joueur incarne un personnage capable de bouger, sauter et lancer des projectiles. Ces projectiles permettent d'attaquer à distance les monstres que le joueur rencontrera sur son parcours.

Le but du joueur sera de traverser une carte déroulante jusqu'à la porte de sortie, en collectant le plus d'objets tout en évitant le contact avec les ennemis.

Les monstres errent de façon aléatoire sur la carte jusqu'à ce qu'ils aperçoivent le joueur et se lancent à sa poursuite. S'ils venaient à le perdre de vue, le monstre retourne alors à un mode de déplacement aléatoire. Si, au contraire, ils parviennent à toucher le joueur, ce dernier perd un point de vie et est alors ramené à son point de départ et doit re-parcourir tout le trajet. Si son niveau de vie venait à atteindre zéro, il meurt et le jeu atteint le gam

Algorithmique

Dans cette première partie seront présentés les différents modèles algorithmiques qui vont guider la conception du jeu.

Menus

Ce premier algorithme représente l'enchaînement des différents écrans du programme, selon les actions portées par le joueur.
À l'exécution du programme, le menu principal / écran-titre est chargé en toutes circonstances.
Le joueur peut alors effectuer différents choix :

- Ouvrir un second menu pour
  - consulter les règles (**rules**),
  - parcourir les références (**credits**), ou encore
  - visualiser les meilleurs scores (**highscores**).

- Démarrer une nouvelle partie en chargeant le premier niveau.

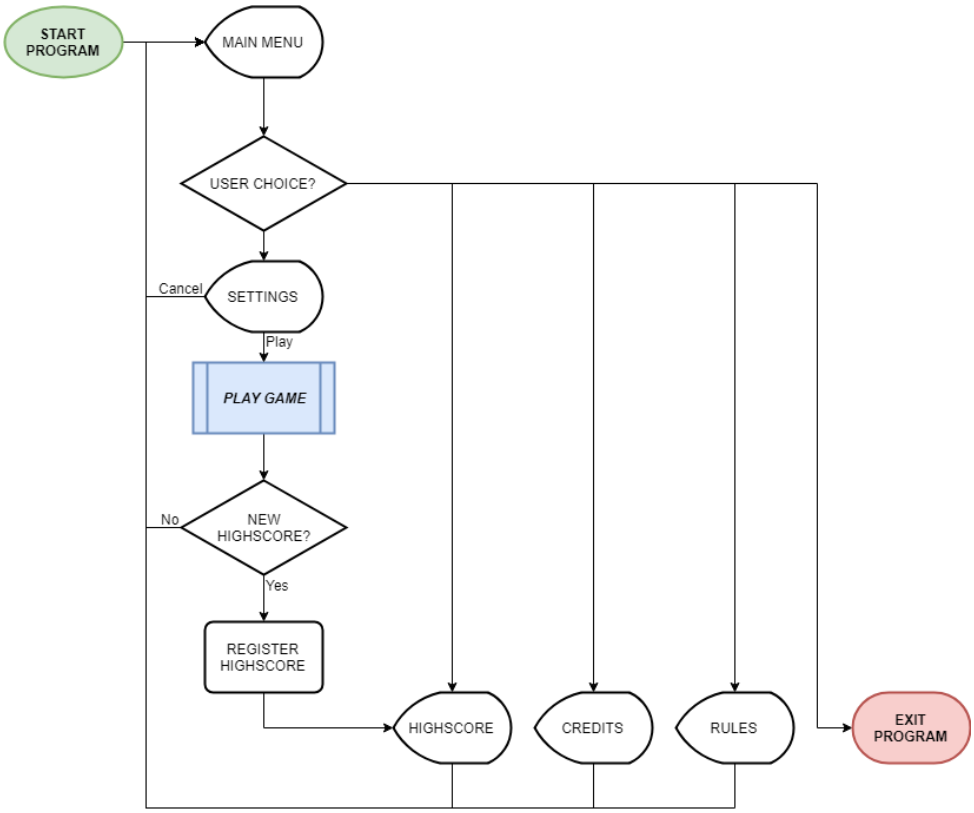

Figure 8 *Algorithme de présentation des menus*

Séquence de jeu

Lorsque le joueur lance une nouvelle partie, l'algorithme suivant est exécuté en boucle :

1. Choix d'une action par le joueur : bouger, attaquer ou ne rien faire.
2. Résolution des conséquences découlant du choix du joueur :
    a. Phase de **déplacement**
    b. Phase d'**attaque**
3. Exécution de la phase des **projectiles** éventuellement tirés en phase 2.b.
4. Application de la phase des **monstres**.
5. Phase de **résolution des statuts** affectant le joueur : power-up.
6. Nouvelle itération jusqu'au *Game Over* (mort du joueur ou fin de niveau).

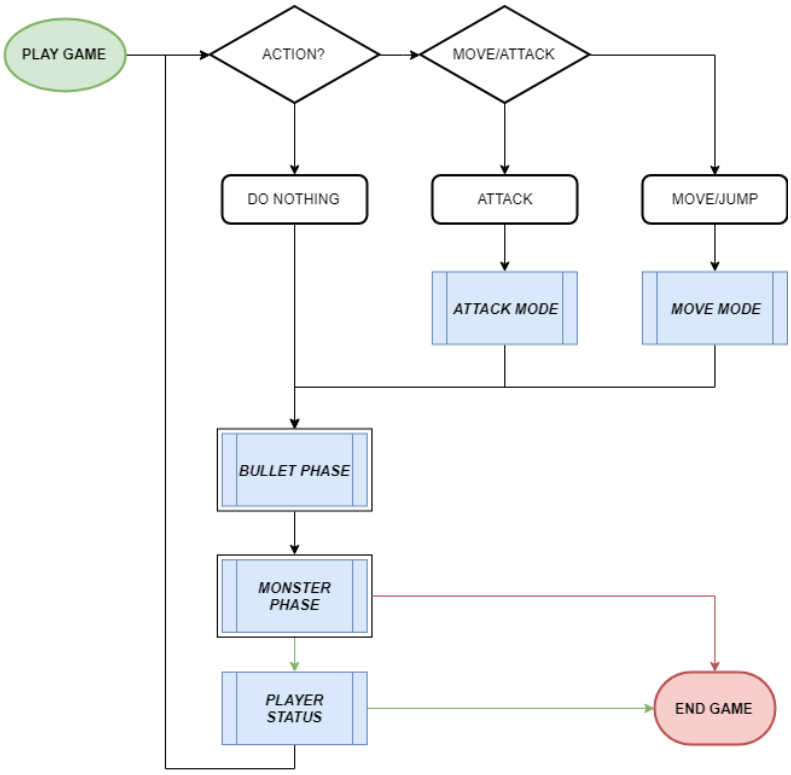

Figure 9 *Algorithme d'une séquence de jeu*

Phase de déplacement

Lorsque le joueur enfonce une touche de déplacement, le programme détermine tout d'abord le sens du déplacement : sens horizontal vs sens vertical.

Sens horizontal

Si le joueur décide d'effectuer un déplacement horizontal, le programme vérifie si le joueur regarde dans le bon sens. Si oui, il contrôle alors que le chemin est libre (pas d'obstacle) et permet au personnage d'avancer. Si non, le personnage ne fait que changer de direction sans se déplacer).

Après déplacement, le programme s'assure que le personnage repose sur une plateforme. Dans le cas contraire, il place celui-ci en position de chute.

Sens vertical

Si le joueur n'est pas en situation de chute ni sis en dessous d'un plafond, alors le programme lui permet d'effectuer un saut temporaire. Une fois ce dernier effectué, si le joueur n'a pas atterri sur une plateforme, alors il est placé en situation de chute.

Une fois le déplacement effectué, le joueur peut collecter les objets sur son chemin, qui augmentent son score et lui confèrent d'éventuelles améliorations.

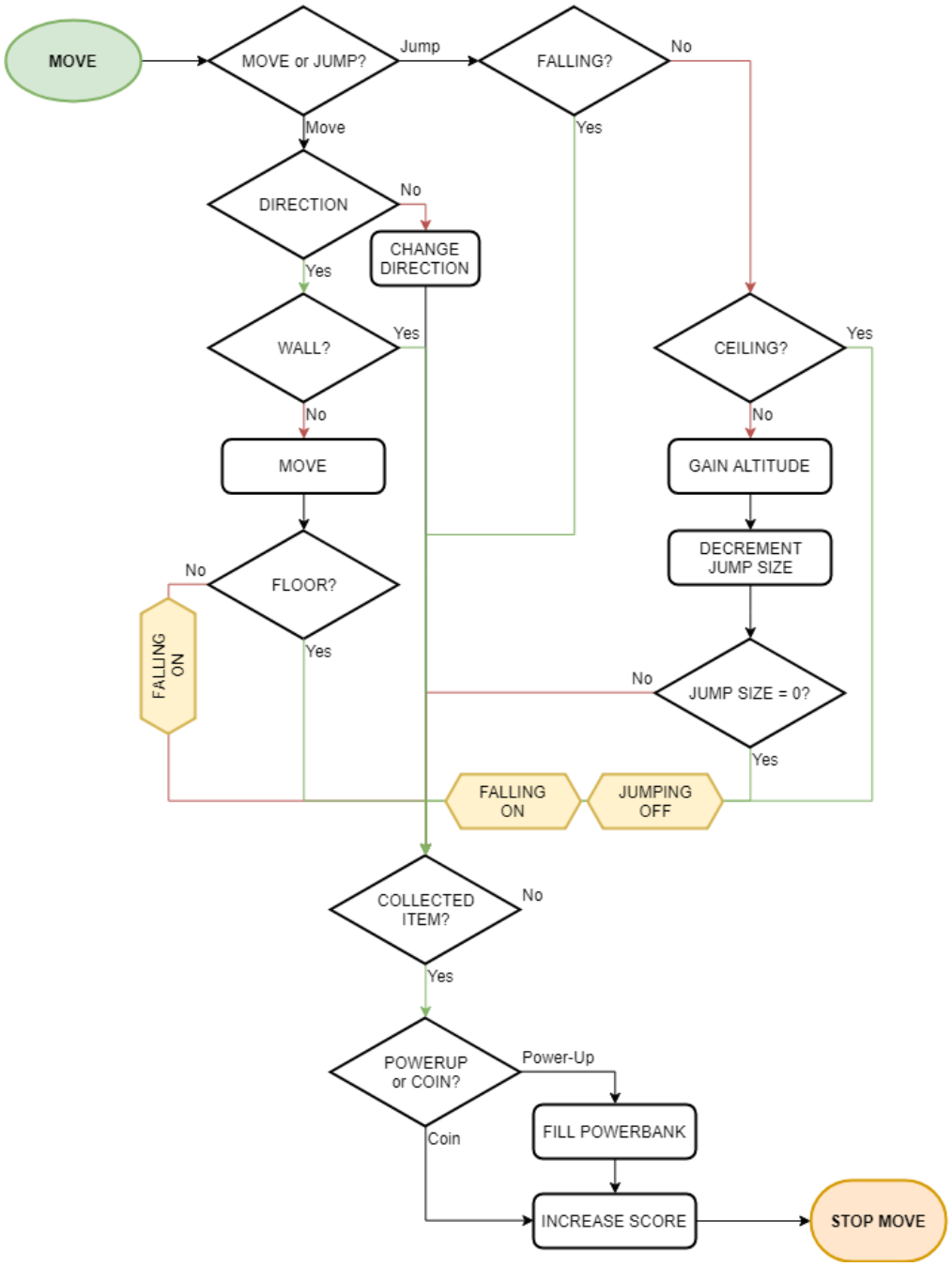

Figure 10 *Algorithme des déplacements*

Phase d'attaque

Si le joueur décide d'effectuer une attaque, la première action s'attèle à vérifier la distance du monstre en face du joueur. Si ce dernier est à portée de bras, le personnage effectuera une attaque au corps-à-corps. Le monstre perd alors un – ou plusieurs – points de vie. Si le niveau de vie du monstre atteint zéro, ce dernier devient étourdi et peut alors être achevé.

> Seul un monstre déjà étourdi peut être vaincu.

Si aucun monstre n'est à portée, le joueur peut alors lancer un projectile et tenter d'atteindre un monstre à distance. Un lancer de projectile ajoute un nouvel objet, doté de sa propre durée de vie et d'un potentiel de dégât. Seul un nombre limité de projectiles, donné par la constante `MAXBULLETS`, est autorisé et, si atteint, provoque la disparition du plus ancien projectile.

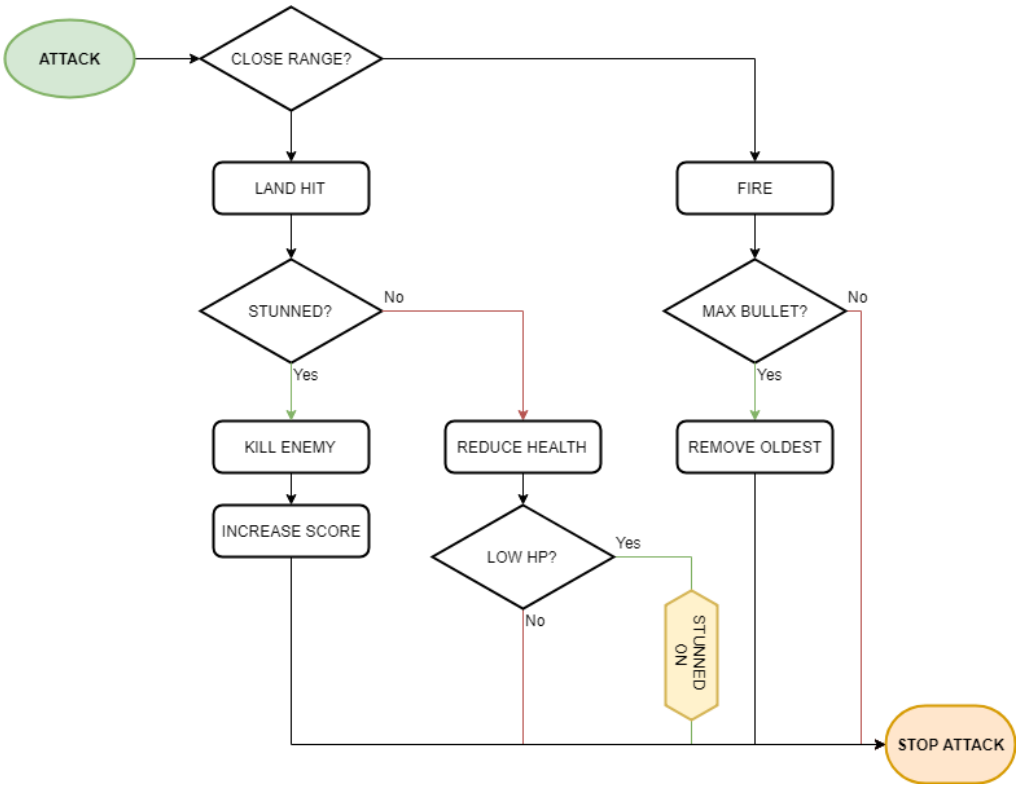

Figure 11 *Algorithme de la phase d'attaque*

Phase des projectiles

Si le joueur a effectué au moins un lancer de projectiles, alors une phase de résolution est enclenchée.

Durant celle-ci est vérifié pour chaque projectile s'il touche un adversaire (auquel il réduit alors les points de vie de l'ordre de son dégât et provoque éventuellement l'état d'étourdissement) ou s'il poursuit sa course. Les projectiles arrivés à échéance ou ayant percuté une cible sont alors retirés du plateau.

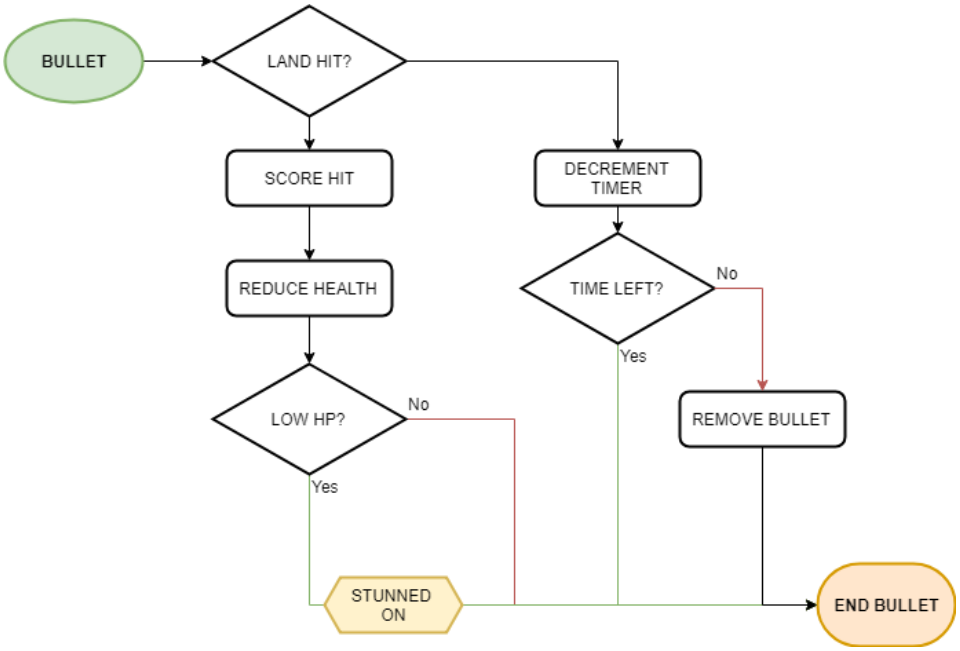

Figure 12 *Algorithme de la phase des projectiles*

Phase des monstres

Une fois les phases du joueur terminées, on vérifie l'état d'étourdissement pour chaque monstre et effectue la séquence applicable :

- Si assommé, décrément du compteur d'état et fin de l'état si applicable.
- Sinon, vérification de la proximité du joueur. Un joueur en vue déclenche un mode d'attaque augmentant la vitesse du monstre et incite ce dernier à poursuivre le joueur si possible.
  Si, en plus, le joueur est à portée, alors le monstre tente une attaque et déclenche le mode défense du joueur.

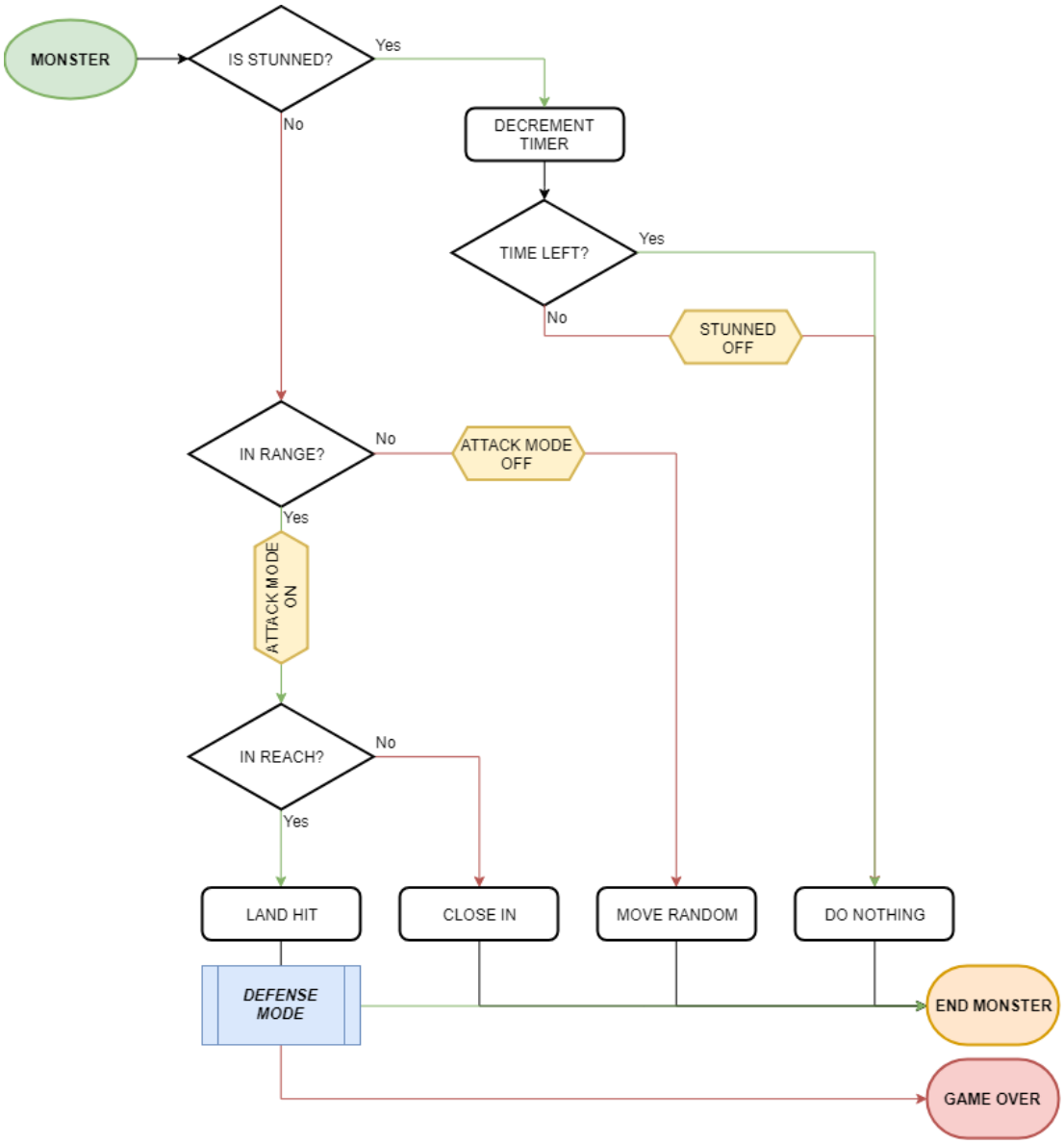

Figure 13 *Algorithme du tour des monstres*

Phase de défense

Si le joueur est attaqué par un monstre, il entre en phase défensive. Durant cette phase, l'invulnérabilité du joueur est testée.

- Si elle est active, le joueur ne subit aucun dégât.
- Sinon, le joueur perd un point de vie. La perte d'un point de vie, si elle n'est pas mortelle, déclenche une invulnérabilité temporaire. Si les points de vie tombent à zéro, le joueur décède et c'est *Game Over*.

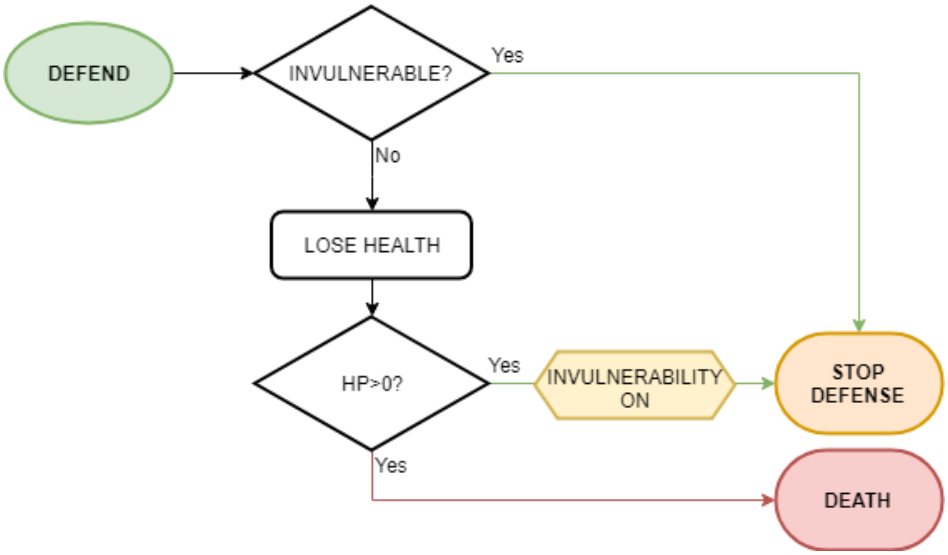

Figure 14 *Algorithme de la phase de défense*

Phase de résolution des statuts

Si le joueur survit à/dépasse la phase des monstres, il reste à évaluer son état : chute, power-ups et conditions de victoire.

- *Power-ups* : on décrémente les compteurs. S'ils sont à zéro, le power-up est désactivé.
- Conditions de victoire : si le niveau est complété et qu'il n'est pas le dernier, on charge le suivant, sinon sortie du jeu.

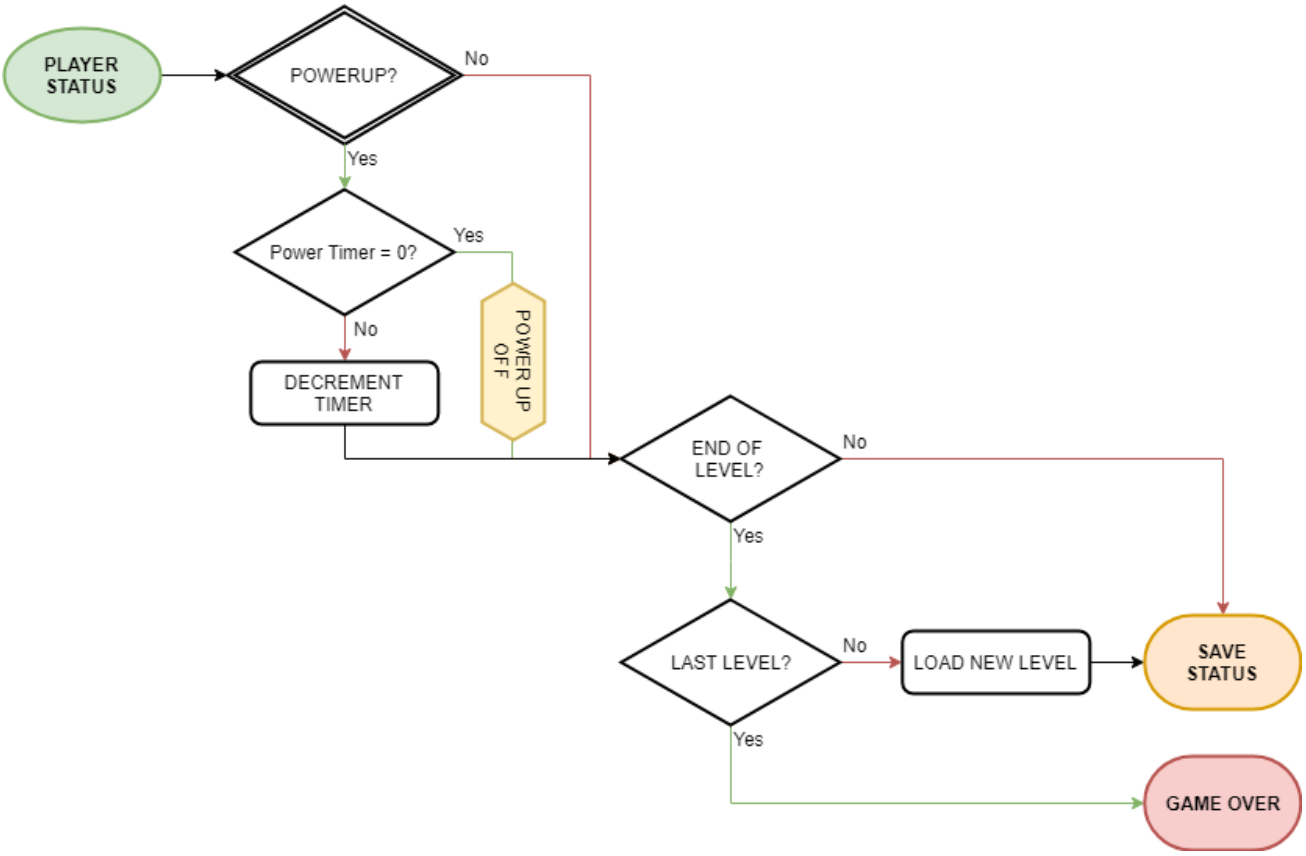

Figure 15 *Algorithme de la phase vérification des statuts*

Structures de données

Pour appliquer ces différents algorithmes, nous allons utiliser différents objets : des constantes, énumérations et structures de données, qui permettront d'organiser le code et de faciliter son maintien.

Constantes

Plusieurs éléments vont être déclarés constants pour l'ensemble des parties, principalement des limites :

- **`MAXWIDTH`** : la largeur maximale autorisée des niveaux.
- **`MAXHEIGHT`** : la hauteur maximale autorisée des niveaux.
- **`SCROLLBASE`** : l'espace minimum avant le début du scrolling.
- **`MAXLEVEL`** : le nombre maximum de niveaux pouvant être chargés.
- **`MAXMONSTERS`** : le nombre maximum de monstres pouvant se trouver simultanément sur le plateau.
- **`MAXPOWERUPS`** : le nombre maximum de power-ups pouvant être collectés par le joueur.
- **`MAXBULLETS`** : le nombre maximum de balles tirées par le joueur pouvant coexister sur le plateau.
- **`BULLETTIMER`** : le temps de vie d'une balle.
- **`BULLETSPEED`** : la vitesse standard des balles.
- **`BULLETATTACK`** : la force standard des balles.
- **`UNIT`** : l'unité de distance dans le jeu (pour la mesure des mouvements en rapport avec les *sprites*).
- **`DISPLAYWIDTH`** : la taille d'affichage.
- **`DISPLAYHEIGTH`** : la hauteur d'affichage.
- **`BOTTOMBANNER`** : la taille de la bannière inférieure pour l'affichage des points et vies du joueur.
- **`FRAMERATE`** : le taux de rafraichissement de l'écran.
- **`ANIMSPEED`** : la vitesse d'animation des personnages et objets.
- **`FLASHINGRATE`** : la fréquence de scintillement.
- **`MOVERATE`** : le taux de mise à jour des déplacements.
- **`MOVETIMER`** : le délai standard de saut.
- **`MAXSCORE`** : le nombre de scores maximum à stocker.
- **`DEFAULTTIMER`** : le temps de saut standard.
- **`MAXPSEUDO`** : la longueur maximale d'un pseudo de joueur.
- **`MAXSTRING`** : la taille maximum des messages pouvant être repris dans le journal d'erreur (*debug*) et dans la plupart des champs de type texte.

L'idée sous-jacente au choix de ces différentes constantes est de faciliter la gestion de ces facteurs durant le développement, en permettant des adaptations fluides (un seul élément de code à changer) tout en gardant une cohérence dans les échelles et les contraintes.

Énumérations

Parallèlement aux constantes, plusieurs énumérations seront déclarées afin de lister explicitement certains choix possibles :

- ***State*** : les différents états possibles du programme (menu ou jeu).
- ***Direction*** : les différentes directions pour les déplacements ou l'affichage.
- ***Colour*** : les différentes couleurs pour l'affichage.
- ***Type*** : les différents types de *sprites*.
  - ***Hero***
  - ***Mob***
  - ***Item***
  - ***Tile***
  - ***Back(ground)***
  - ***Button***
- ***Anim*** : les différentes animations des *sprites*.
- ***Effect*** : les différents effets associés aux objets (qu'ils soient un *power-up* ou un simple collectable).
- ***Menu*** : les différents menus disponibles.
- ***Error Level*** : le degré de sévérité des alertes émises pendant l'exécution.

À nouveau, le choix posé ici a pour but d'augmenter la lisibilité et l'efficacité du code en encapsulant différents concepts.

Structures

L'encapsulation des concepts est ici poussée plus loin encore en rassemblant les données nécessaires à l'exécution du programme dans des structures logiques.

- ***Game*** : structure principale du jeu, elle contient l'ensemble des informations sur la partie en cours, le temps écoulé, ainsi qu'une référence vers le contexte actuellement chargé.

```
Struct Game {
  Resources* resources;
    Support* support;
    Context* context;
       Menu* menu;
      state  state;
};
```

- ***Resources*** : les ressources représentent les différents éléments réutilisables du jeu, tels que le bestiaire, les cartes de niveaux, les *sprites* ou encore les polices d'écriture.

```
Struct Resources {
    Level* levels;
    Asset* assets;
  Monster* monsters;
     Item* items;
     Font* fonts;
};
```

- ***Support*** : le support représente les éléments contextuels qui n'influencent pas le jeu fondamentalement, à savoir les meilleurs scores et les crédits.

```
Struct Support {
    Score*  scores;
     char** credits;
};
```

- ***Score*** : le score est une structure qui a pour fonction de retracer le score du joueur, selon la valeur des objets et des monstres tués, ainsi qu'un compteur d'ennemis abattus. Elle contient également une chaine de caractère pour stocker le résultat du joueur en cas de meilleur score. Cet objet fait à la fois partie du support et du contexte (voir ci-après).

```
Struct Score {
   char  pseudo[MAXPSEUDO];
    int  points;
    int  kills;
  Score* next;
};
```

- ***Context*** : le contexte représente les informations sur le jeu en cours, des monstres actifs aux objets collectés et projectiles tirés, en passant par le score et l'état de la partie.

```
Struct Context {
      Score* score;
      Level* level;
     Player* player;
    Monster* monsters;
     Bullet* bullets;
       Item* items;
       bool  isValid;
       bool  isSuspended;
       bool  isWon;
};
```

- ***Level*** : cette structure vise à renseigner le niveau actuellement chargé, ses dimensions et son éventuel décalage lorsque le *scrolling* est applicable.

```
Struct Level {
   char   name[MAXSTRING];
    int   index;
    int   width;
    int   height;
    int   scrollx;
    int   scrolly;
   char** map;
  Level*  next;
};
```

- ***Player*** : cette structure rassemble les caractéristiques sur le joueur, à savoir sa position sur le plateau de jeu, ses valeurs de santé, de force et de vitesse de base.
  Comme le jeu comporte des éléments pouvant améliorer ou réduire les valeurs de santé/vitesse/force, des *trackers* sont également nécessaires pour conserver les valeurs d'application par rapport aux valeurs de base.

```
Struct Player {
  Coordinates* coord;
        Power* powerbank;
          int  baseHealth;
          int  currentHealth;
          int  baseSpeed;
          int  currentSpeed;
          int  baseAttack;
          int  currentAttack;
       Sprite* sprites;
};
```

  En plus de posséder des caractéristiques de base, le joueur peut collecter des objets qui doivent également être conservés : c'est le rôle du *power bank*.

- ***Power-Up*** : on retrouve dans le power *bank* les power-ups : des objets spécifiques qui, lorsque ramassés par le joueur, lui confèrent des augmentations (ou réduction) temporaires de caractéristiques. L'effet qu'ils apposent à ce moment est actif pour une durée limitée. La structure comprend aisn i différents éléments qui vont permettre de suivre l'état de ce dernier selon son action, sa durée restante et l'effet donné.

```
Struct Power {
  effect  type;
    bool  active;
     int  timer;
   Power* next;
};
```

- ***Item*** : les objets en général sont dotés d'une position, d'un type (pour savoir s'ils doivent rejoindre le *power bank* ou non) et leur valeur pour le score du joueur.

```
Struct Item {
  Coordinates* coord;
         char  name[MAXSTRING];
          obj  type;
       effect  effect;
          int  value;
       Sprite* sprites;
         Item* next;
};
```

- ***Monster*** : aux côtés du joueur, on retrouve également des monstres, des personnages semblables en caractéristiques mais qui rapportent un certain nombre de points lorsque vaincus. Ils possèdent un certain nombre de traits supplémentaires pour mener à bien leur mission de monstre.

```
Struct Monster {
  Coordinates* coord;
         char  name[MAXSTRING];
          mob  type;
          int  baseHealth;
          int  currentHealth;
          int  baseSpeed;
          int  currentSpeed;
          int  baseAttack;
          int  currentAttack;
          int  sight;
          int  reach;
         bool  isStunned;
         bool  isAttacking;
          int  timer;
          int  scoreValue;
       Sprite* sprites;
      Monster* next;
};
```

- ***Bullet*** : le jeu permet au joueur de lancer des objets en direction des monstres, qu'on appellera ici génériquement des balles. Celles-ci possèdent une base de dégâts, un temps limité de vie et une position permettant de déterminer s'il y a impact.

```
Struct Bullet {
  Coordinates* coord;
          int  baseAttack;
          int  currentAttack;
          int  baseSpeed;
          int  currentSpeed;
          int  timer;
       Bullet* next;
       Sprite* sprites;
};
```

- ***Coordinates*** : les coordonnées ont une double vocation : suivre les déplacements des personnages/objets et déterminer leur emplacement d'affichage/visualisation.
  - Elle se compose d'une sous-structure ***Moving***, qui suit le momentum associé aux coordonnées.

```
Struct Coordinates {
       float  x,y,w,h;
   direction  look;
      Moving* moving;
};
```

```
Struct Moving {
  bool left;
  bool right;
  bool up;
  bool down;
   int timer;
};
```

- ***Asset*** : on se servira des assets pour stocker les animations et les *sprites* qui les composent. Ceci permettra par la suite de les retrouver et de les manipuler plus facilement.
    - Une sous-structure ***sprite*** contiendra ensuite les différentes images des animations.

```
Struct Asset {
    char  path[MAXSTRING];
    type  type;
     int  name;
    anim  anim;
     int  frames;
  Sprite* sprites;
   Asset* next;
};
```

```
Struct Sprite {
     char  path[MAXSTRING];
      int  frame;
      int  width;
      int  height;
   GLuint  texture;
   Sprite* next;
};
```

- ***Font*** : il s'agit d'une structure technique qui permet de référencer une police de caractère lue par FreeType et de la stocker pour utilisation subséquente.

```
Struct Font {
     char  name[MAXSTRING];
  FT_Face  face;
      int  index;
     Font* next;
};
```

- ***Menu*** : il s'agit d'une structure technique qui suit l'état des menus, à savoir le menu actuellement sélectionné et l'endroit où se situe le focus de l'utilisateur.

```
Struct Menu {
  int id;
  int focus;
};
```

- ***Key*** : il s'agit d'une structure technique qui suit l'état des touches du clavier, si celles-ci sont enfoncées et si leur action a déjà été prise en compte.

```
Struct Key {
  bool pressed;
  bool inactive;
};
```

- ***Log*** : cette structure ne fait pas partie du jeu à proprement parler. Elle existe dans le but de structurer les messages renvoyés par le code pendant une exécution (qu'elle soit à succès ou non), afin de faciliter le *debugging*.

```
struct Log {
        char* time;
  errorLevel  level;
        char* location;
        char* msg;
};
```

La réalisation de ces structures a suivi plusieurs règles :

- Toutes les structures commencent par une majuscule, de sorte à les distinguer des variables standards lors des déclarations.
- Toutes les structures pouvant être enchaînées contiennent un lien vers la structure suivante sous le nom *next*.
- Toutes les structures sont déclarées au moyen de **`typedef`** permettant de transformer ces structures en variables « standards » et d'ainsi simplifier visuellement le code

Interface

Le jeu se déroule dans une fenêtre simple, de taille relativement restreinte (selon les dimensions qui ont été déterminées dans les constantes **`DISPLAYWIDTH`** et **`DISPLAYHEIGTH`**).

Le premier écran présenté à l'utilisateur est bien entendu l'écran d'accueil, qui présentera le titre du jeu et donnera accès aux différents sous-menus, tels que précisé dans le workflow **Menus**. L'utilisateur peut ainsi utiliser les touches Z et S pour choisir l'un des menus et appuyer sur Enter pour le sélectionner.

Figure 16 *Écran principal*

Règles

Sur l'écran des règles, sont présentés les contrôles principaux de jeu, qui seront utilisés par le joueur pour déplacer le personnage.

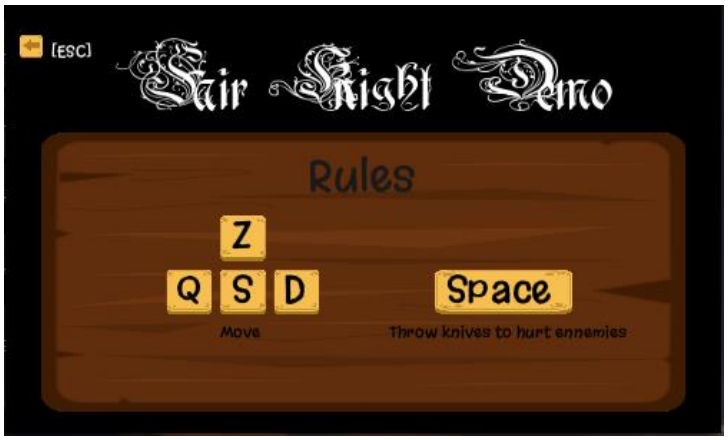

Figure 17 *Écran des règles*

Scores

Le tableau des scores reprend la liste des meilleurs résultats, classés par valeur puis nombre d'ennemis tués, associés à un pseudo sur 10 caractères.

Figure 18 *Écran des scores*

Credits

Le menu des crédits reprendra les crédits (non exhaustifs) vers les ressources utilisées.

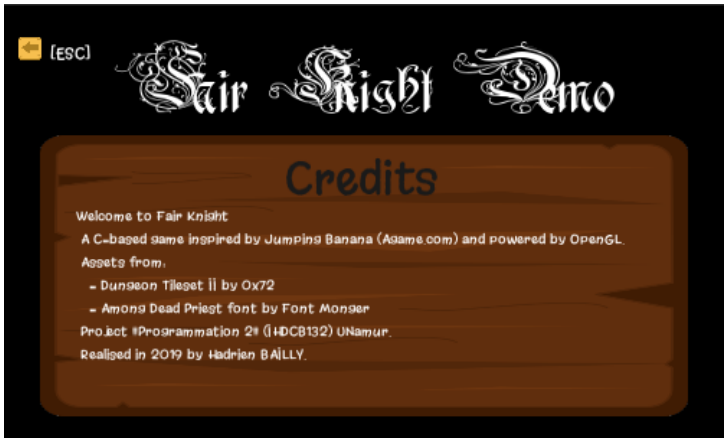

Figure 19 *Écran des crédits*

Jeu

L'écran de jeu est divisé en 4 zones :

1. La zone principale de **jeu**, où le personnage évoluera et effectuera ses actions.
2. La zone de la **jauge de vie** (points totaux et points restants).
3. La zone de **score**, avec les points obtenus, et les ennemis vaincus.
4. Les boutons de **pause** et de **sortie**.

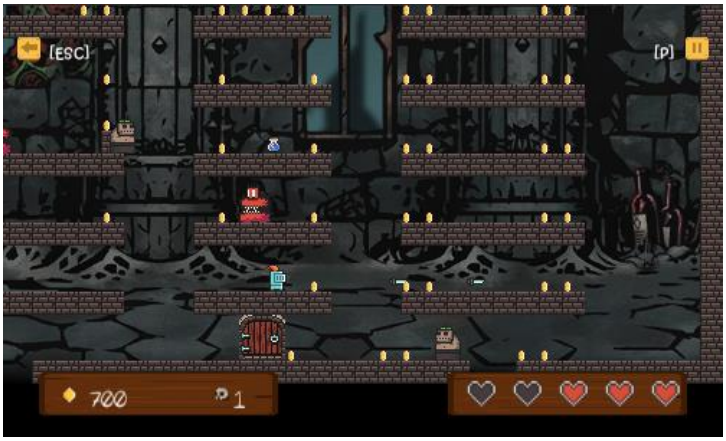

Figure 20 *Écran de jeu*

Lorsque la partie est terminée, un écran devait donner au joueur son score et lui permettre de l'inscrire parmi les meilleurs scores (si applicable). Cet écran n'a malheureusement pas pu être implémenté.

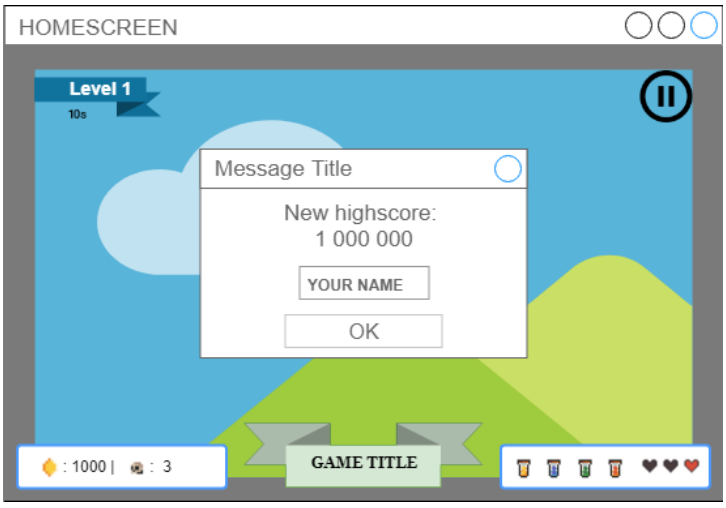

Figure 21 *Écran d'inscription de score*

Implémentation

Dans cette section, nous allons revenir sur le code réalisé sur base des notions présentées ci-avant. Nous discuterons ainsi de la structure générale des fichiers de code, leur organisation en répertoires et sur l'implémentation effective des algorithmes.

Précisons tout d'abord que le jeu a été codé intégralement en anglais, pour des raisons de commodité, de cohérence et de simplicité essentiellement. En effet, étant d'une part la langue utilisée par le langage C et offrant d'autre part des formulations souvent plus brèves, l'anglais s'est naturellement imposé comme la langue de choix pour la réalisation du code.

En second lieu, il est important de souligner que, si les algorithmes ont fourni une base de travail et une structure au code, toutes les fonctionnalités qu'ils mentionnent n'ont pas été implémentées. Nous reviendrons sur les raisons de ces écarts au cours de la section **Retour sur image**.

Structure des fichiers

Le code du projet s'est organisé autour de plusieurs catégories de fichiers :

- Le code principal (*main*), qui comprend l'ensemble de la logique propre au jeu.
- Le code d'affichage (*rendering*), qui contrôle les éléments liés à GLUT et à l'affichage des images à l'écran.
- Le code-ressources (*resources*), qui reprend l'accès aux éléments nécessaires pour l'affichage (images et polices).
- Le code de soutien (*support*), qui contextualise le jeu sans en changer fondamentalement le fonctionnement.
- Le code utilitaire (*tools*), qui fournit des fonctions génériques pour faciliter le travail de codage.

On peut ensuite distinguer deux types de code :

- Les codes de type **constructeurs**, qui s'occupent de la gestion des structures. Ceux-ci s'inspirent fortement des méthodes des langages orientés objet (tel Java). On y retrouve ainsi des constructeurs et des destructeurs, des *setters* et des *getters*, ainsi que des fonctions permettant de manipuler le contenu des objets de façon abstraite.
- Les codes de type **procéduraux**, qui vont utiliser les fonctions des constructeurs pour élaborer la logique du jeu.

Une attention particulière a été portée à la structuration cohérente des fichiers, l'ordre et le choix des noms de fonction.

- Ainsi, les fonctions objet `init` (*initialisation*) permettent d'instancier un objet avec des valeurs initialisées sur un défaut.
- Les fonctions typées `set` et `get` (*functions*) permettent soit de mettre à jour soit d'obtenir une information depuis un objet ou une collection d'objet.
- Les fonctions booléennes `is` (*tests*) permettent de vérifier un état.
- Enfin, les fonctions `destroy` (*destroy*) permettent de supprimer proprement un objet de la mémoire.

Pour le reste des fonctions, le choix du nom s'est attaché à en décrire le rôle de façon explicite, de sorte à ce qu'il ne soit pas nécessaire d'en connaître le fonctionnement pour en comprendre l'utilité (par exemple, `loadAssetSprites` qui permet de charger les *sprites* correspondantes à une ressource).

En toute occasion, le nom fait référence à la, ou les, structure(s) manipulées quand applicable, avec une terminaison au pluriel lorsqu'elles permettent de réaliser une opération sur une liste chaînée de cette structure.

Par ailleurs, des fonctions *logger* ont été disposées tout au long du code. Ces fonctions proviennent de l'utilitaire *debug.h* et ont pour vocation de documenter le déroulement du code et d'en faciliter le *debugging* et, par ricochet, servent également de commentaire utile à certains passages du code.

Structure des répertoires

Pour organiser les fichiers, les répertoires ont été découpé selon un premier niveau général, qui distingue le type (au sens large) des fichiers :

- Les fichiers de type **ressources** (*assets*), principalement des images mais également des fichiers connexes, comme des fichiers de propriétés ou les cartes des niveaux.
- Les fichiers de type **documentation** (*doc*) qui reprennent les fichiers extérieurs au code qui en documentent le fonctionnement (tel que le présent rapport).
- Les fichiers de type **code** (*code*) qui constituent le cœur du programme.

Dans un second temps, au niveau du répertoire du *code*, les sous-dossiers poursuivent la logique des catégories de code. On retrouvera ainsi les blocs *main, rendering, resources, support, tools* et également un bloc *include* pour les librairies externes.

Code Review

Revenons maintenant sur les différentes parties du code pour en expliquer le principe de fonctionnement. Au cours des paragraphes suivants, nous allons présenter le fonctionnement global du jeu, puis revenir sur le code avec pour fil conducteur la découpe des répertoires.

Principe général

Hors des répertoires, nous trouverons tout d'abord la fonction *main*, qui permet de démarrer le programme, ainsi qu'un *header lib.H* dont l'utilité est de fournir un point central vers les constantes et un raccourci pour les librairies standard de C.

La fonction *main* permet de démarrer les deux processus du programme : le processus principal exécutant la logique du jeu et le processus d'affichage de GLUT. Le processus principal se charge de récupérer les différentes ressources (graphiques et logiques) qu'il passe éventuellement à GLUT, gère la mémoire et manipule les données. Le processus GLUT commence

quant à lui par enregistrer un ensemble de fonctions qui vont lui permettre de dialoguer avec le processus principal, afin de permettre un input/output au cours de l'exécution.

Le point de départ du processus principal est bien entendu la structure *Game*. Celle-ci fournit un point central auquel on va pouvoir raccrocher les trois composants du jeu : les ressources, le code support et bien entendu le contexte de jeu. La première étape de l'exécution est ainsi de lire et de référencer les différentes ressources utilisées par le jeu, qu'elles soient graphiques (images du jeu et police) ou logique (propriétés des monstres et objets, cartes des niveaux), ainsi que les éléments textuels. Une fois que ces éléments sont présents, le processus GLUT est appelé et lance l'affichage de l'écran d'accueil.

Le contexte est ensuite construit lorsque l'utilisateur lance une partie de jeu : la carte du premier niveau est lue et les différents éléments du jeu sont générés en mémoire. On procède alors aux différentes phases de jeu et on interagit avec la mémoire en fonction des apparitions et disparitions des objets.

Une fois la partie terminée, le contexte est détruit et libère la mémoire. L'utilisateur est alors ramené à l'écran d'accueil. Lorsqu'il sort du programme, le reste des éléments sont également détruits.

Main

Au sein du code *main* se trouvent tous les éléments liés à la logique du jeu : on y retrouve ainsi le plateau de jeu, les structures des différents personnages et objets dispersés sur celui-ci, le détail de leur position et de leur état dans le jeu, et de manière plus générale, le **contexte** de jeu.

On retrouve dès lors tous les constructeurs liés aux éléments de jeux : niveau, joueur, monstre, objets, power-ups, balles, et coordonnées. Ces constructeurs sont ensuite appelés par le *builder*, le programme de création du contexte.

Le *builder* démarre par la lecture de la carte du niveau à charger. Sur cette carte aux dimensions précisées dans le fichier de configuration `/assets/level/levels`, se trouvent le type et les coordonnées des différents éléments à inscrire dans le contexte : le joueur, les différents monstres habitant la carte, les objets collectables, les plateformes et la sortie. Durant la lecture, le programme procède à l'instanciation et au chaînage de ces éléments dans la structure de contexte.

Pour construire la carte, le *builder* repose bien entendu sur les ressources rendues disponibles au démarrage du programme (voir ci-après *Resources*) : sans ces ressources, il aurait fallu coder en dur les caractéristiques et celles-ci n'auraient plus été modifiables une fois le code compilé. Notons au passage que le *builder* possède deux fonctions supplémentaires qui lui permettent soit de recharger le niveau courant, soit de charger le niveau suivant (en cas de victoire, par exemple).

Une fois que le contexte a été construit, celui-ci doit encore être rendu vivant par le déroulement d'actions. C'est le rôle du *Phaser* (*phase*), qui va régler le comportement des différents objets autonomes et mobiles : le joueur, les monstres et les balles. On retrouve ici les fonctionnements décrits au point précédent **Algorithmique**. Grâce à eux, le joueur pourra se mouvoir, collecter des pièces et lancer des balles ; les monstres pourront se déplacer, soit aléatoirement, soit se mettre en chasse du joueur ; les balles pourront voler jusqu'à atteindre leur cible ou disparaître.

La gestion des déplacements se fait par le biais du code *motion.h*, qui traite des collisions de deux manières : soit par modulo de l'unité du jeu pour déterminer la case couramment utilisée par le joueur, soit par collision bi-axiale entre les différents personnages et objets (sur base d'une *hitbox*).

Un morceau de code non utilisé est présent dans ce répertoire, il s'agit du code *power.h* qui, s'il avait été possible de le réaliser, permettrait au personnage de gagner des pouvoirs temporaires tels qu'un boost de vitesse, des restaurations de santé, etc. Ce code aurait alors impliqué une étape supplémentaire dans le *Phaser* pour gérer l'utilisation de ces pouvoirs.

Rendering

Le code *rendering* a trait à tous les éléments nécessaires pour le bon fonctionnement de GLUT et à l'**affichage** du jeu. C'est donc sans surprise que le code principal se trouve dans *display.h*, qui initialise GLUT et appelle tous les autres fichiers.

Pour fonctionner, GLUT enregistre en effet au démarrage plusieurs fonctions pour lui permettre de gérer l'input et l'output de données à l'écran. Il faut ainsi définir les fonctions suivantes :

- Une fonction d'affichage (*draw* et *UI*)
- Une fonction de rafraichissement (*update*)
- Une fonction de capture du clavier (*keyboard* et *controls*)

Ce à quoi a été rajouté un constructeur/contrôleur pour les menus (*menu*).

Les fonctions d'affichage sont sans conteste les plus élaborées et les plus diverses, et permettent d'afficher des images qui auront été préalablement chargées. Elles permettent également de dessiner du texte sur demande au moyen de police de type .ttf (*True Type Font*). Ces fonctions se présentent sous plusieurs variantes afin de faciliter le code d'affichage tout en permettant une grande flexibilité en définitive. Il est en effet possible de choisir l'une ou l'autre méthode selon que l'on souhaite dessiner de manière centrée, de manière préconfigurée, etc.

Deux librairies externes ont été nécessaires pour parvenir à cette fin :

- **SOIL** (*Simple OpenGL Image Library*), pour le chargement d'image et
- **FreeType** pour le chargement de fichier de police.

Ces deux librairies ont permis la prise en charge de la partie très complexe et poussée de gestion des fichiers au format spécialisé, à travers des interfaces plus ou moins simplifiées et documentées (voir ci-dessous ***Include***).

Une fois les images chargées, l'affichage devait encore être mis à jour pour les animer. Plusieurs séquences de rafraichissement ont ainsi été mises en place : une première à 60 FPS, puis d'autres à des temps plus longs pour les autres traitements comme les animations ou la gestion des déplacements.

Enfin, il fallait encore permettre à l'utilisateur d'interagir avec le jeu, en capturant ses interactions au moyen du clavier. Pour cela, GLUT offre deux fonctions : une qui détecte la pression d'une touche, une seconde son relâchement. A chaque touche a alors été associée une action, en fonction de l'état de déroulement du programme et des actions possibles.

Resources

La partie *resources* s'attèle à enregistrer et organiser les éléments recyclables dans l'exécution du code afin d'optimiser la consommation de mémoire et de faciliter l'utilisation de ces ressources.

Les ressources sont chargées au démarrage du programme et parcourent ainsi plusieurs fichiers de propriétés :

- */assets/fonts/fonts*, qui détaille les polices disponibles.
- */assets/sprites/sprites*, qui liste les images disponibles et leur type.
- */assets/text/monster*, qui donne la liste et les caractéristiques monstres.
- */assets/text/items*, qui reprend les objets disponibles.

Sur base de ces fichiers de propriétés, le programme peut construire les noms de fichiers nécessaires pour charger effectivement ces ressources. Il peut ainsi récupérer les différentes images et reconstruire les animations, charger les polices via FreeType, et construire le bestiaire et l'inventaire nécessaires au *builder* de niveau.

Afin de pouvoir retrouver plus facilement les images, une couche a été ajoutée aux *sprites* pour les organiser selon une forme de bibliothèque. Ce sont les *assets*, qui permettent d'identifier chaque image selon un certain nombre de caractéristiques et qui sont ainsi associées aux différents sous-répertoires des *sprites assets*.

Support

Comme précisé ci-avant, la partie *support* n'est pas fondamentale pour le jeu mais permet d'ajouter du contexte (au sens large) au jeu : sont ainsi ramenés depuis des fichiers la liste des crédits (*/assets/text/credits*), ainsi que le tableau des meilleurs scores (si existant, */assets/score/scores*).

Ces fichiers sont également chargés au démarrage, tandis que les scores sont réécrits à chaque fois qu'une nouvelle entrée est ajoutée au tableau.

Tools

Enfin, les fonctions *tools* sont des codes écrits pour faciliter le développement.

Le code *debug.h* permet ainsi de générer des journaux d'application à la fois à l'écran et dans un fichier */assets/debug.log* lorsque des évènements se produisent dans le jeu.

```
01:46:22 [INFO] game: game starting...

01:46:22 [INFO] context: initialising context...

01:46:22 [INFO] score: initialising score...
01:46:22 [INFO] score: score initialised.

01:46:22 [INFO] player: instanciating player...
01:46:22 [INFO] player: player instanciated.

01:46:22 [INFO] context: context initialised.

01:46:22 [INFO] builder: building context...

01:46:22 [INFO] resources: loading level...
01:46:22 [INFO] resources: Level 1 loaded.
01:46:22 [INFO] level: loading map...
01:46:22 [INFO] level: opening file Jungle-1.map...
01:46:22 [INFO] level: Jungle-1.map successfully opened.
01:46:22 [INFO] level: 31 rows loaded
01:46:22 [INFO] level: Jungle-1.map file closed.

01:46:22 [INFO] builder: locating player...
01:46:22 [INFO] builder: player coordinates: [3,2].
01:46:22 [INFO] builder: map exit set.
01:46:22 [INFO] builder: context built.
                ~ spawned 6 monsters
                ~ placed 102 items
01:46:22 [INFO] controls: pausing game.
```

Figure 22 *Log d'application*

On distinguera ainsi plusieurs niveaux de journaux :

- TRACE, qui correspond à des logs très détaillés (pour un *debugging* précis).
- INFO, qui correspond à de l'information utile pour suivre l'exécution du programme.
- WARN, qui renvoie à une erreur ayant pu être corrigée ou à une action destructrice.
- ERR, qui relève une erreur durant l'exécution n'ayant pas pu être corrigée.
- FAIL, qui capture une erreur critique et met fin proprement à l'exécution du programme.

Le code *util.h* offre lui quelques fonctions purement pratiques, telles que la concaténation de deux chaînes de caractères ou la manipulation de fichiers (*open/close*).

Include

Pour réaliser ce code, deux librairies externes ont été employées : *SOIL* et *FreeType*.

- SOIL est une librairie de chargement d'image. Elle permet de lire de nombreux formats, dont le PNG, et d'en transformer le contenu en un objet texturable par GLUT. Cette librairie intervient dans le code *draw.h.*

- *FreeType* est une libraire de chargement de police. Elle permet d'écrire dynamiquement des chaînes de caractères à l'écran sur base de fichiers .ttf *TrueTypeFont* et offre ainsi plus de flexibilité pour l'affichage de texte, par rapport à la fonction standard de GLUT `glutBitmapCharacter`.

Ces deux librairies sont présentes dans le répertoire *include* plus pour des raisons de documentation que d'inclusion, puisque leur version de développement doit être installée préalablement à la compilation du code pour fonctionner.

Graphe des dépendances

Voici ici les liens de dépendance entre les différents fragments de code. Ne sont ici pas représentés exhaustivement les liens entre les parties *rendering* et *main* (représentées en bleu épais).

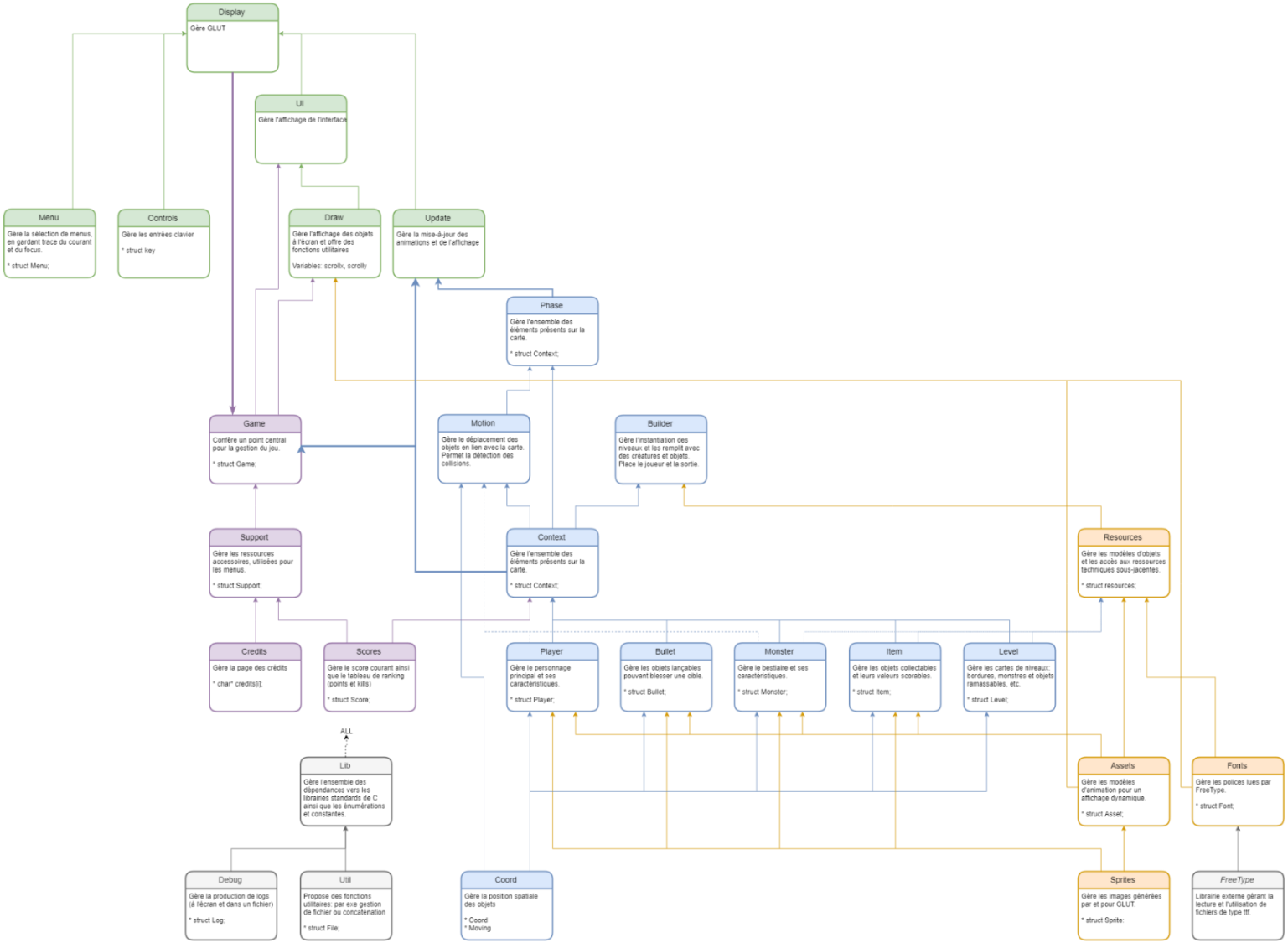

Figure 23 *Depencency Graph*

Conclusion

Retour sur image

Ce projet a été l'occasion de se confronter à un premier développement d'envergure, qui couvre de nombreux aspects fonctionnels : affichage, algorithmique, gestion du temps qui s'écoule, création d'un programme de bout en bout… Autant d'éléments qu'il n'est pas toujours possible de rencontrer dans une pratique plus « classique » du code.

Au cours de mes précédentes expériences de programmation, l'accent avait en effet toujours porté sur la réalisation d'une opération transactionnelle en vase clos, dont il fallait éventuellement optimiser le temps de réponse. Dans le cadre de *Jumping Bananas*, il fallut renverser l'approche : le code était désormais cyclique et l'exécution devait évoluer au fil des interactions avec l'utilisateur.

Cela m'a ainsi permis d'utiliser, de mieux comprendre le fonctionnement et de solidifier de nombreux concepts de la programmation : allocation dynamique de mémoire, structures de données, utilisation de pointeurs, voire de pointeurs vers pointeurs, … Il a fallu maîtriser tous ces concepts pour parvenir à éviter les problèmes de segmentation et les fins abruptes d'exécution.

J'ai également pu réaliser l'importance d'un code bien structuré, tant dans le code lui-même que dans la pratique d'écriture. Il fallait s'attacher à concevoir des interfaces puissantes mais toujours avec simplicité, avec des fonctions claires et unifonctionnelles. Mon regret est d'ailleurs de ne pas avoir pu saisir ces notions plus tôt, car elles auraient facilité grandement la réalisationdu projet et permis d'écrire un code plus robuste.

La réalisation de ce projet a en effet été marquée par de nombreuses refontes du code à la découverte de nouveaux concepts ou de bonnes pratiques. Cette avance par tâtonnements, qui conduit à des niveaux inégaux dans le code, est en partie liée à la complexité et la certaine imperméabilité de la librairie FreeGLUT3, dont on retrouve peu de tutoriels et de manuels de référence en C, et à la recherche de la librairie SOIL me permettant de faciliter l'utilisation des fichiers images. Il y avait également la méconnaissance des bonnes pratiques de programmation en C, qui a généré de la complexité là où il n'y aurait pas dû en avoir.

Pour cette raison, il n'a pas été possible d'implémenter toutes les fonctionnalités prévues par les algorithmes pensés en début de projet : étourdissement des monstres et corps-à-corps, collecte et utilisation de *power-ups*,, ou encore écran de sauvegarde de meilleurs scores. La base technique et algorithmique de ces fonctionnalités est néanmoins déjà présente, comme en atteste le fichier *power.h* et il serait possible de rajouter celles-ci facilement dans une seconde phase de développement.

Il reste également deux bugs qui n'ont pas pu être corrigés pour la remise du projet :

- Il peut arriver que le personnage ou un monstre reste accroché à un bloc de type #.
  Ce problème est causé par la méthode de gestion des collisions avec les limites par modulo par rapport à la map plutôt que par collision effective. C'est un choix qui a été fait durant le développement pour gérer les interactions avec le niveau de manière simple, mais qui montre ici ses limites.
- Lorsqu'un monstre est en mouvement, il n'est pas possible de lui lancer un projectile depuis son dos. Si la collision est bien détectée, le projectile ne parvient pas à blesser ou tuer le monstre touché.

Enfin, il aurait été souhaitable de revenir sur les fonctions de lecture de fichiers, que ce soit les images ou les fichiers csv. Le code reste pour le moment fort dépendant des énumérations de type (*hero,mob, obj, tile* ou *back*), et n'est pas très résistant aux incohérences entre les différentes portions de code ou à l'absence d'un fichier.

Développements possibles

Comme précisé ci-avant, il serait intéressant dans le cadre d'une continuation du développement, de travailler aux éléments suivants :

- développer un mécanisme de lecture des fichiers de propriétés pouvant mieux détecter et réagir à l'absence de ces fichiers, et également de se passer des énumérations en dur,
- continuer la partie sur l'utilisation des power-ups (ramassage, utilisation, écoulement), ou encore
- revoir le mécanisme de déplacement sur la carte.

On pourrait également tenter d'ajouter de nouvelles fonctionnalités :

- Dangers liés à l'environnement (sol à pointes).
- Dégâts continus
- Armes spéciales
- …

Bibliographie

0x72. (2019). *16x16 Dungeon Tileset*.
Récupéré sur Itch.io: https://0x72.itch.io/16x16-dungeon-tileset

0x72. (2019). *16x16 Dungeon Tileset II*.
Récupéré sur Itch.io: https://0x72.itch.io/dungeontileset-ii

B., M. (2019). *Marginally Hopeful (Font)*.
Récupéré sur Itch.io: https://disabledpaladin.itch.io/marginally-hopeful-font

Computer Hope. (2019). *Unix and Linux commands help*.
Récupéré sur Computer Hope: https://www.computerhope.com/unix.htm

Dummer, J. L. (2019). *Simple OpenGL Image Library*.
Récupéré sur Lonesock.net: https://www.lonesock.net/soil.html

Free Software Foundation, Inc. (2019). *GNU Make*.
Récupéré sur GNU Operating System: https://www.gnu.org/software/make/manual/html_node/

fsy. (2019). *GUI - Game Buttons fsy002*.
Récupéré sur Itch.io: https://fsy.itch.io/gui-game-buttons-fsy002

Hock-Chuan, C. (2019). *GCC and Make*.
Récupéré sur programming notes: https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html

Kilgard, M. J. (2019). *The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3*.
Récupéré sur OpenGL.org: https://www.opengl.org/resources/libraries/glut/spec3/spec3.html

Laurent, A. (2019). *Comprendre la programmation d'un jeu vidéo - Les animations 2D*.
Récupéré sur Developpez.com: https://alexandre-laurent.developpez.com/tutoriels/programmation-jeux/animations-2D/

noobtuts.com. (2019). *C++ 2D Pong Game*.
Récupéré sur noobtuts.com: https://www.noobtuts.com/cpp/2d-pong-game

Olsen, S. (2019). *NeHe Productions*.
Récupéré sur http://nehe.gamedev.net: http://nehe.gamedev.net/tutorial/freetype_fonts_in_opengl/24001/

OpenGL Help Network. (2019). *The Official Guide to Learning OpenGL, Version 1.1*.
Récupéré sur GlProgramming.com: glprogramming.com/red

O'Reilly. (2019, 03 01). *TGA File Format Summary*.
Récupéré sur FileFormat.Info: https://www.fileformat.info/format/tga/egff.htm

Pema. (2019). *OpenGL Colour Codes*.
Récupéré sur Pema's Vitual Hub: https://pemavirtualhub.wordpress.com/2016/06/20/opengl-color-codes/

Roelofs, G. (2019). *PNG - The Definitive Guide*.
Récupéré sur libpng.org: http://www.libpng.org/pub/png/book/chapter13.html

Simple OpenGL Image Library. (2019).
Récupéré sur Lonesock.net: https://www.lonesock.net/soil.html

Swiftless. (2019). *OpenGL Keyboard Interaction (Version 2.0)*.
Récupéré sur Swiftless.com: http://www.swiftless.com/tutorials/opengl/keyboard.html

The FreeType Project. (2019). *FreeType Tutorial / I*.
Récupéré sur freetype.org: https://www.freetype.org/freetype2/docs/tutorial/step1.html

The FreeType Project. (2019). *FreeType-2.10.0 API Reference*.
Récupéré sur freetype.org: https://www.freetype.org/freetype2/docs/reference/ft2-version.html

The Khronos Group Inc. (2019). *OpenGL 2.1 Reference Pages*.
Récupéré sur Khronos.org: www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/

Tutorials Point. (2019). *C Programming Tutorial*.
Récupéré sur tutorialpoints.com: https://www.tutorialspoint.com/cprogramming/index.htm

Tutorials Point. (2019). *The C Standard Library*.
Récupéré sur tutorialspoint.com: https://www.tutorialspoint.com/c_standard_library/index.htm

Unknown. (2019). *ASCII Tables*
Récupéré sur asciitable.com: http://www.asciitable.com

Vries, J. d. (2019). *Learn OpenGL*.
Récupéré sur https://learnopengl.com/

Table des illustrations

Outils de développement utilisés

Librairies

- freeGLUT3-dev
- libfreetype6-dev
- libsoil-dev

Programmes

- Linux Mint sur machine VirtualBox.
- Atom, *A hackable text editor for the 21st Century*, avec les modules suivants (non-exhaustif) :
  - linter (incl. linter-gcc/linter-gcc2),
  - atom-beautifier (incl. Uncrustify)
  - gcc-make-run
- GitHub & Gitkraken
- draw.io