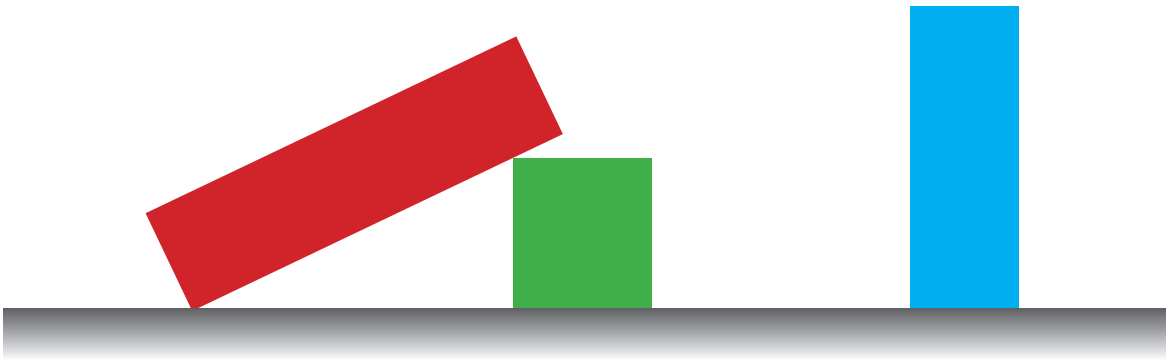
ESIPE imac ingénieur

THOMAS WAS DEAD

(IN AN EXPLOSION)

Maëlle Carlier
Yohan Le Breton



Sommaire

1) Le structures de données utilisées

Structure générale du jeu

Notre jeu est constitué d'un **"menu"** et de plusieurs **niveaux**.

Les informations propres à chaque niveau sont contenues dans un **fichier**. (Fichier "0" pour le menu, fichier "1" pour le niveau 1…). Le niveau 0 permet d'atteindre les niveaux qu'autorise notre progression dans le jeu. Il permet aussi de recommencer le jeu du début en supprimant la progression.

Nous avons utilisé des **tableaux** comme structures de données.

Un niveau est **défini** par :
- un tableau de structure Personnage
- un tableau de structure Obstacles
- un tableau de textures
- une musique
- un tableau complémentaire contenant les identifiants et les positions de début et de fin des obstacles en mouvements.

Les structures de données

Les structures de données sont définies dans le fichier obstacles_persos.h

Structure Obstacle

Nos obstacles ont tous un **identifiant** différent, cet attribut nous permet de les identifier.

Nous avons **6 type** d'obstacles

- **0 = objet traversable** : On ne calcule pas les collisions entre les obstacles de ce type et les personnages.

- **1 = obstacle standard** : Collisions entre les personnages et ces obstacles.

- **2 = personnage** : Obstacle qui correspond à un personnage.

- **3 = objectif** : C'est un obstacle traversable ayant le même identifiant et les mêmes dimensions qu'un obstacle personnage. Si tous les Obstacles personnages sont en collision avec leurs objectifs, le niveau est considéré comme terminé.

- **4 = ennemis** : Si un personnage entre en collision avec un personnage de ce type, le joueur perd et doit recommencer le niveau.

- **5 = porte vers un autre niveau** : Ces obstacles sont présents au niveau 0, ils permettent de lancer un niveau lorsque le personnage le touche.

```
typedef struct Obstacle
Obstacle;
struct Obstacle
{
   int id;
   int type;
   int texture;
   float x, y, z;
   int width, height;
};
```

Le numéro de texture correspond à l'image qui sera chargé sur l'obstacle. Par exemple un obstacle du niveau 1 qui a comme numéro de texture 1, aura comme texture l'image 1.jpg ou 1.png contenue dans le fichier "1/images".

Structure Personnage

Un **personnage** est un **obstacle** avec des attributs en plus : une vitesse, une position de fin, une puissance de saut, des booléens qui permettent d'identifier s'il y a une collision et de bloquer le mouvement du personnage, une durée de saut et une distance avec le sol.

```
typedef struct Personnage Personnage;
struct Personnage
{
 Obstacle * obs;
 float vx, vy;
 float xfin, yfin;
 int jumpPower;
 int touchGround, touchTop, touchLeft, touchRight;
 int jumpDuration;
 float groundDistance;
};
```

Utilisation de tableaux

Nous avons fait le choix d'utiliser des tableaux, nous avons donc des constantes qui définissent la taille de nos tableaux.

Par exemple dans le niveau 0, il n'y a qu'un personnages, donc les deux dernières cases du tableau **persos[]** contiennent **"NULL"**. Comme beaucoup de fonctions parcourent l'ensemble du tableau, nous avons dû être rigoureux et mettre de nombreux tests pour ne pas tenter d'accéder à des données de pointeurs nuls.

Pourquoi des **tableaux** et pas des **listes chaînées** ?
Une utilisation des tableaux n'est peut être pas la structure la plus optimisée. En effet lors du parcours des tableaux, comme expliqué au dessus, nous parcourons le tableau en entier alors que celui-ci peut n'être qu'à moitié plein. Cependant nous avons préféré cette structure aux listes chaînées car c'est une structure plus intuitive, qui nous a semblée plus simple à utiliser.
De plus, nous n'avions pas initialement pensé à travailler avec des listes chaînées, nous n'avons pas voulu perdre de temps à changer nos tableaux en listes chaînées comme nos tableaux ne sont pas de très grande taille. Cependant pour un développement plus poussé du jeu, ce choix pourrait poser problème.

Organisation des modules

Notre programme est divisé en 1 fichier principal et 3 modules.

- **main.c** : programme principal.
- **obstacles_ persos.h/.c** contiennent les structures et les fonctions relatives à la création des obstacles et des personnages ainsi que les fonctions gérant les collisions et et les déplacements.
- **fonctionnement.h/.c** : contiennent les fonctions pour lancer, dessiner, valider, mettre en pause un niveau.
- **textures_son.h/.c** : contiennent toutes les fonctions relatives aux images, textures, à la musique et aux bruitages.

2) Les déplacements et collisions

Toutes les fonctions detaillées dans ce paragraphes sont dans le fichier **obstacles_ persos.c**

Déplacement et physique des personnages

Un **personnage** a à la fois une position **(x, y)** et une vitesse **(vx, vy)**.

À chaque tour de boucle la main, on réalise les actions suivantes :

On écoute **les évènements du clavier** pour savoir si le joueur veut déplacer le personnage.

Si oui, et que par exemple le joueur veut se déplacer vers la droite, on ajoute une valeur positive à **vx** (une valeur négative si l'on veut aller à gauche)

On teste les **collisions** entre le personnage et tous les objets de la scène, et si collision il y a, on garde en mémoire quels côtés du personnage sont concernés. (fonction **CollisionsPersonnage**)

On met à jour les **vitesses** pour prendre en compte la gravité et les frottements du sol. **vx** est multiplié par un coefficient inférieur à 1, et on soustraie à **vy** un coefficient qui dépend du temps depuis lequel le personnage n'a pas touché le sol (pour avoir une chute non linéaire). (fonction **updateVitesses**).

On met à jour les **coordonnées du personnage** en fonction de la vitesse et des collisions, par exemple si le joueur veut aller à droite et que l'on détecte une collision sur le bord droit du personnage, on passe la vitesse **vx** à **zéro**. et donc la position sur l'axe des x ne change pas. Par contre si l'on ne détecte pas de collision, **x** est mis à jour, et on a :
x += vx * k . (fonction **movePersonnage**).

Une fois les variables de position à jour, il ne reste plus qu'à **dessiner** le personnage

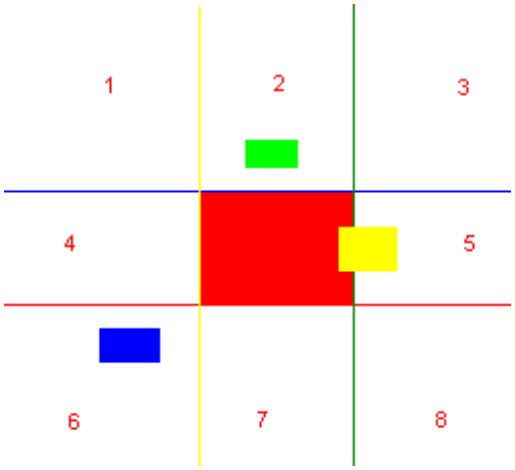
Gestion des collisions

Le but de notre **détection de collision** est de déterminer à chaque tour de boucle quels sont les bords, pour chaque personnage, qui entrent en collision avec un des objets de la scène.

Notre fonction **CollisionsPersonnage** prend en paramètre un pointeur sur un personnage (**p**), ainsi que le tableaux des obstacles **(obs[])**

Tout d'abord la fonction initialise à zéro les quatre points de collisions possibles du personnage **(touchTop, touchLeft, touchRight, touchGround)**.

Ensuite, à l'aide d'une boucle for, on passe en revue chaque objet de la scène pour tester les quatre points de collisions possibles. Par exemple pour que **touchRight** passe à , (Collision entre les rectangles rouge et jaune sur le schéma) il faut que l'obstacle soit entre la borne supérieure et inférieure du personnage mais aussi qu'il soit entre la borne gauche et droite de ce personnage.



Les obstacles qui bougent

Nous avons décidé d'ajouter des obstacles qui se déplacent. La fonction **deplaceObstacleContinue()** se charge des déplacements des obstacles mais aussi des personnages qui peuvent être poussés par ces obstacles. Le tableau **caractObsQuiBougent[]** contient les informations des obtsacles qui bougent : identifiant, sens du mouvement (1 aller, 2 retour), xDeb, yDeb, xFin,yFin. Pour chercher l' identifiant d'un obstacle à déplacer, on parcourt le tableau de 6 en 6.

Détail de la fonction **deplaceObstacleContinue()**

-On regarde si l'identifiant est contenu dans le tableau **caractObsQuiBougent[]**. S'il n'y est pas c'est que l'obstacle est immobile.

-Sinon on regarde le sens de déplacement de l'obstacle et on lance la fonction **deplaceObstacle()** qui prend en argument la position vers laquelle se déplace l'obstacle.

La fonction **deplaceObstacle()** s'occupe de déplacer l'obstacle et renvoi 1 si l'obstacle a atteint la position vers laquelle il se dirige, 0 si le déplacement est en cours. Si le déplacement est fini on change le sens de déplacement de l'obstacle et celui-ci repart dans l'autre sens.

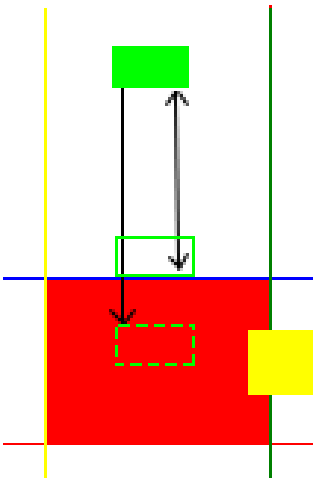
La fonction teste aussi si cet obstacle entre en collision avec un personnage en utilisant la fonction **detectionCollisionPersoObs()**. Cette fonction a le même fonctionnement que **CollisionsPersonnage()**, sauf qu'elle ne teste les collisions qu'entre un objet et un personnage et qu'elle ne change pas les attributs du personnage. Elle renvoie un entier selon la nature de la collision. S'il y a collision avec un personnage, la fonction le déplace.

Il faut ensuite regarder si ce personnage déplacé n'entraîne pas le déplacement d'un autre personnage. C'est le rôle de la fonction **persoPoussePerso()**. Enfin il faut regarder si ce deuxième personnage poussé n'entraine pas le déplacement du troisième personnage en rappelant la fonction **persoPoussePerso()**.

Problèmes rencontrés

Les **collisions** nous ont posés pas mal de problèmes. Notamment dû au fait que lorsque le personnage se déplace trop vite, celui-ci se déplace de plusieurs points par boucle, et il se retrouve à l'intérieur d'un obstacle **avant** que l'on ne détecte la collision.

Pour régler ce problème lorsque le personnage tombe sur un obstacle, nous avons décidé de calculer à chaque tour de boucle, si le personnage est en chute, **sa distance avec l'obstacle le plus proche** se trouvant en dessous de lui. Et ainsi, si un obstacle est plus proche que le prochain emplacement du personnage, la vitesse de ce dernier est **ajustée** pour que le personnage se place exactement sur le haut de l'obstacle.



Le **déplacement** des obstacles, a été une partie un peu délicate car il a fallu penser à tous les cas de collisions entre personnages (un personnage pousse un autre qui en pousse un autre…) Nous n'avons pas géré les cas où un personnage poussé entre en collision avec un autre obstacle (lorsqu'il est pris en tenaille, ou écrasé au sol). Nous avons donc construit nos niveaux pour que ce cas de figure ne se présente pas.

3) Gestion de la caméra

Les fonctions de gestion de la caméra sont issues du fichier **obstacles_ persos.c**

Suivi du personnage

La **position** de la caméra est gérée à l'aide de deux variables : **CenterX** et **CenterY** . Ce sont les coordonnées du point de la scène que l'on veut mettre au centre de la fenêtre d'affichage. À chaque tour de la boucle principale, on commence par réinitialiser la position de l'affichage avec **glLoadIdentity()**, puis nous mettons à jour **CenterX** et **CenterY** grâce à notre fonction **moveCameraSuivrePerso()**, et enfin ce n'est pas la caméra qui bouge mais l'ensemble de la scène grâce à la fonction **glTranslatef()** utilisant les coordonnées **CentreX** et **CentreY**.

Détail de la fonction **moveCameraSuivrePerso()**, cette fonction **centre l'affichage sur un personnage**:
On calcule **Dx** et **Dy**, la distance entre le personnage et le centre de la fenêtre d'afichage.
On définit une **zone de côté Lx, Ly**, proportionnelle à la taille de la fenêtre, qui sera la zone de liberté du personnage où la scène n'a pas besoin d'être déplacée.
Si le personnage sort de la zone de liberté, il faut déplacer la caméra et donc modifier **CenterX** et **CenterY**
On soustrait à **CenterX** et **CenterY** une fraction de **Dx** et **Dy** (Dx / 10 par exemple). Cela aura pour effet de rapprocher **CenterX** et **CenterY** du centre de la fenêtre d'affichage de manière progressive. Plus la distance entre les deux est grande, plus le déplacement sera rapide.

L'attribut z et l'effet de parallaxe

Chaque obstacle possède **un attribut Z**. Les obstacles sont **triés** dans le tableau **Obstacles[]** selon un **z croissant.** Nous avons utilisés un tri à bulle, ce n'est pas la technique de tri la plus rapide, mais nous l'avons préférée car elle a été simple à mettre en place. Lors de la boucle d'affichage des obstacles, les obstacles seront dessiné dans la scène selon leur z et ainsi un obstacle avec un z élevé sera dessiné avant les obstacles qui doivent se trouver devant lui.

De plus cet attribut permet de donner un **effet de profondeur** car les obstacles bougent en fonction de leur Z associé :

Dans la fonction **moveCameraSuivrePerso()**, à chaque fois qu'un déplacement du centre de la caméra est effectué, chaque obstacle est déplacé de cette même distance multiplié par leur Z.
obstacles[i]->x -= obstacles[i]->z * deplacementCentreX * 0.05;

Et donc une image qui sert d'arrière-plan avec un Z élevé, sera presque immobile lorsque la caméra se déplace, et les personnages avec un Z de zéro ne seront pas affectés.

4) L'utilisation des fichiers textes.

Les fonctions d'initialisations se trouvent dans le fichier **obstacles_ persos.c**

Initialisation des structures

Pour initialiser les obstacle :
La fonction **initializeObstaclesFromFile()** va chercher les informations contenues dans le fichier **Obstacles.txt**, elle parcourt le fichier caractère par caractère. Si elle trouve le caractère **"="** elle scanne les chiffres jusqu'au caractère **";"** si elle a scanné **7** nombres c'est que le personnage est **immobile**, elle appelle la fonction **createObstacle()** et stocke l'obstacle dans le tableau **Obstacles[]**. Si elle a scanné **9** nombres, c'est que l'obstacle est **mobile**, elle crée l'obstacle et stocke son identifiant et ses positions de début et de fin de mouvement dans le tableau **caractObsQuiBougent[]**.

Pour initialiser les personnages ont utilise la fonction **initializePersoFromFile()** qui fonctionne sur le même principe. La fonction récupère les informations pour créer les personnages et leurs obstacles objectifs.

Sauvegarde de la progression

Dans le dossier 0 se trouve un fichier texte : **Progression.txt** dans lequel se trouve le numéro du **niveau** jusque auquel le joueur est allé. Ainsi depuis le niveau 0 seul les niveaux inférieurs ou égaux à la progression sont accessibles. Lorsque l'on fini un niveau, si celui-ci est égal à la progression, celle ci est incrémentée et on a accès au niveau suivant. On peut aussi **réinitialiser** la progression à partir du niveau 0 en touchant la porte "**restart**". La progression est gérée à partir des fonctions **LireProgression()**, **AugmenterProgression()**, **RestartProgression()** du fichier **fonctionnement.h**

Lancement des niveaux

Pour lancer un niveau on utilise la fonction **ChangerNiveau()** du fichier **fonctionnement.c**. Cette fonction libère les structures du niveau précédent : les obstacles, les personnages, les textures, la musique. Elle initialise depuis le fichier du nouveau niveau les nouvelles structures, elle recentre la fenêtre d'affichage et elle trie les obstacles selon leurs z.

Pour lancer un niveau depuis le niveau 0, le personnage doit entrer en collision avec la porte du niveau qu'il veut lancer. On teste les collisions entre le personnage et les portes grâce à la fonction **lancerNiveauDepuisPorte()** du fichier **fonctionnement.c** et on lance le niveau en conséquence selon la progression du joueur.

5) Autre

L'intégration de vidéos

Nous avons **deux types de vidéos** : les vidéos au lancement et à la fin du jeu, ainsi que la vidéo du volcan pour l'arrière-plan du niveau quatre.

Le **volcan** est une série de **15 images** png que nous appliquons à tour de rôle comme texture sur un même obstacle. L'opération se trouve dans la fonction **drawObstacleTexture()** du fichier **fonctionnement.c**, et pour afficher les images à la cadence voulue, nous utilisons une simple opération qui change le numéro de texture de l'obstacle tous les six tours de boucle.
(o->texture = (nbLoop/6)%15 + 20)

Pour **les vidéos d'introduction et de fin**, nous avons créé les vidéos sur After Effects avant de les exporter en une suite d'images png accompagnée d'un fichier audio. Puis nous avons créé une fonction **launchVideo()** qui se trouve dans le fichier **fonctionnement.c**. Cette fonction prend en paramètre le nom du dossier où se trouve les images et la musique, et elle affiche les images les unes après les autres, à une cadence de 25 images par seconde, tout en jouant le fichier audio associé.

Cette méthode montre cependant des **limites**, car même pour afficher une vidéo de quelques secondes, le nombre d'images à charger est important, et nous avons donc dû réduire le format de nos images pour que le programme ne soit pas trop long à démarrer.

Mise en pause du jeu.

Dans le fichier **main.c** on a une variable **pause**.

Si **pause** vaut :
- **0** c'est que l'on joue.
- **1** c'est que le jeu est en pause avec une image de pause.
- **2** c'est que le jeu est en pause et on a gagné le niveau.
- **4** c'est que le jeu est en pause et on est game over.

Au début du main on teste la valeur de la variable pause et selon sa valeur on va lancer le déplacement des personnages ou le bloquer et dessiner un obstacle de pause avec une texture différente selon l'état de la pause (game over, victoire...).

Les textures

Dans chaque fichiers niveau, il y a un dossier images qui contient des images du type : **1.jpg**, **2.png**…
Nous avons fait ce choix pour simplifier la gestion des images et des obstacles. Un obstacle a un **attribut texture**. Si son attribut vaut **0**, il sera dessiné **sans texture** avec une couleur définie dans la fonction **drawObstacle()**. Si son attribut vaut **1** il sera dessiné avec la texture **1.jpg**, s'il vaut 2 avec la texture **2.pn**g… grâce à la fonction **drawObstacleTexture()**. Les fonctions de dessins sont dans le fichier **fonctionnement.c**.

Les fonctions qui gèrent la création des textures sont dans **textures_son.c**.

Avant chaque niveau on lance la fonction **initialiseTextures()**.
Cette fonction appelle la fonction **load_images()** qui charge les images du niveau dans un tableau. Cette fonction charge les images de format **PNG** et **JPG**.
Ensuite elle appelle la fonction **transformImageEnTexture()** qui remplit un tableau de texture à partir du tableau d'image. Le numéro de l'image correspond à l'indice de la texture dans le tableau de textures.
Enfin une fois que les images ont été transformées en textures, on peut libérer l'espace mémoire allouée grâce à la fonction **FreeImages()**.
Il ne faut pas oublier de faire appelle à la fonction **glDeleteTextures()** lorsque l'on n'a plus besoin des textures.

La musique et les bruitages

Les fonctions qui gèrent la musique et les bruitages sont dans le fichier **textures_son.c**.
Les sons et la musique sont gérés grâce à la bibliothèque **SDL_mixer**.
Les **bruitages** (de saut, de fin de niveau et quand on meure) sont communs à tout le jeu, ils sont initialisés au début du main et les fichiers sont dans le dossier **"0/musique"**. Ce sont des bruitage de type **Mix_Chunk**.
La musique change à chaque niveau, elle est contenue dans le fichier musique du niveau et elle est gérée par les fonction de base de **SDL_mixer**

6) Problèmes rencontrés

- Nous avons eu des difficultés avec les collisions (détaillées dans le **paragraphe 2**).
- On a eu un peu de mal à adapter les fonctions que l'on avait vu pour la gestion d'une image et d'une texture, en fonctions qui gèrent plusieurs images et plusieurs textures.
- Nous avons eu des difficultés pour **organiser** notre programme (faire un **makefile**) et plusieurs **modules** qui s'appellent les uns les autres.
- Enfin nous avons eu quelques problèmes de compatibilité entre **Linux** et **Mac**. Certaines parties du code marchaient sous **mac** mais pas sous **Linux**.
- Pour le gestion de la **mémoire**, nous avons voulu utiliser **Valgrind** mais il semble que juste en initialisant la SDL et en la quittant, valgrind détectait des fuites de mémoires que nous ne pouvions pas gérer. Nous avons donc pris soin de libérer la mémoire que nous avons alloué et avons attribués à un bug de la SDL les fuites de mémoires détectées par valgrind dans notre programme.

7) Ce que l'on aurait fait si on avait eu encore plus de temps

- Un mode multijoueur (deux joueurs sur un seul clavier controllant chacun un personnage).
- Des animations sur les personnages (lorsqu'il meurt ou lorsqu'il saute)
- Un code avec plus de modules et des fonctions mieux organisées.
- Plus de niveaux.