Projet de Programmation : Bubble Bobble
Rapport Final

Geoffroy LARUE

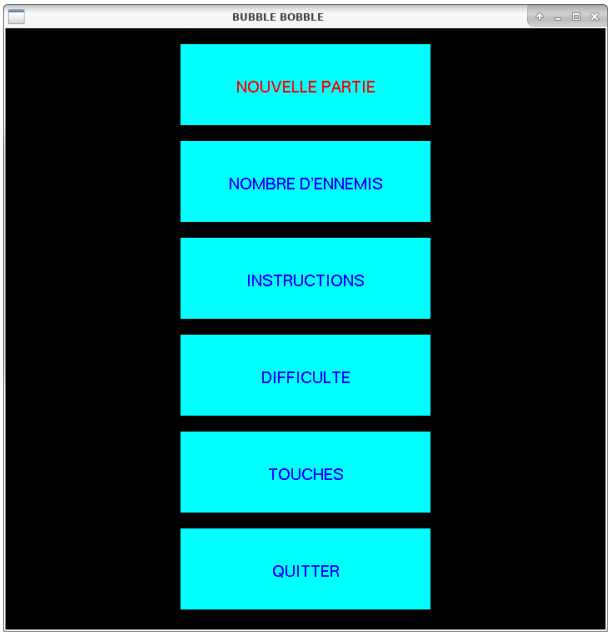
Contents

1 Introduction

Ce court rapport décrit notre implémentation du jeu "Bubble Bobble" en C avec les libraires GL, GLU et GLUT. Elle constitue pour nous une première expérience de programmation "graphique" en C et permet de découvrir la répartition en modules, ainsi que de mettre en pratique des structures de données approchées durant les cours théoriques et/ou via d'autres sources. L'objet de ce document est double. Pour l'auteur, il a servi à mettre au clair la façon de de mener à bien ce projet, Ensuite, il est destiné à faciliter la tâche des personnes chargées de "rentrer" dans le programme afin de l'évaluer.

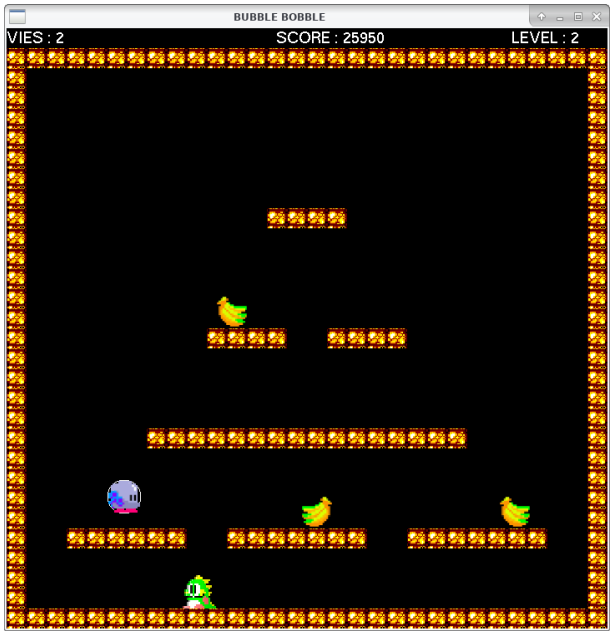
2 Ecrans

2.1 Menu



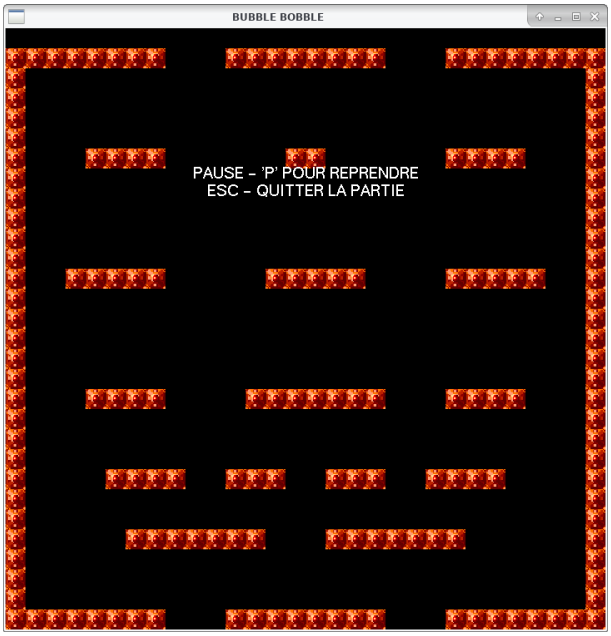
Cet écran est celui qui est affiché au lancement du programme. C'est également par ici que le programme repasse à la fin d'une partie.

2.2 Jeu



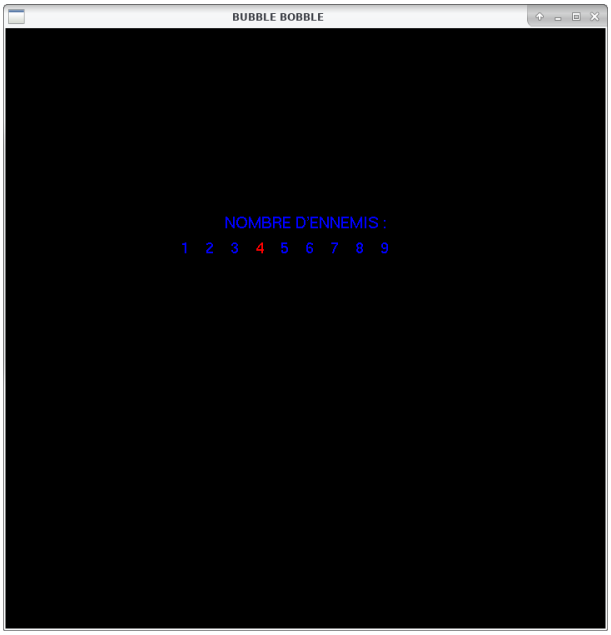
Il s'agit de l'écran du jeu à proprement parler. Au dessus sont affichés le niveau actuel, le nombre de vies restantes et le score actuel du joueur.

2.3 Pause



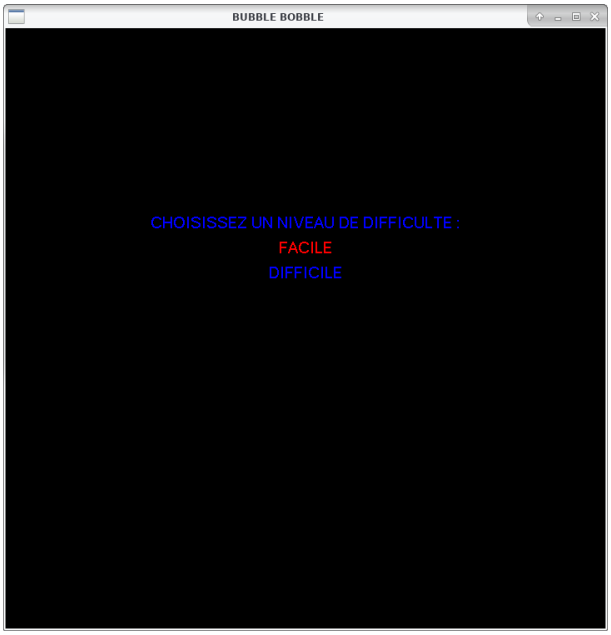
Nous donnons la possibilité au joueur de mettre le jeu en pause, mais nous masquons les personnages durant cette interruption.

2.4 Choix du nombre d'ennemis



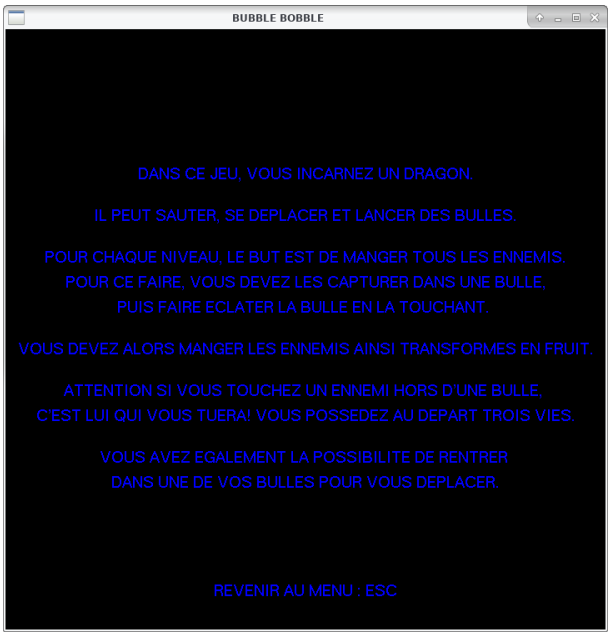
Par cet écran, le joueur peut choisir combien d'ennemis il affrontera durant la partie.

2.5 Choix du niveau



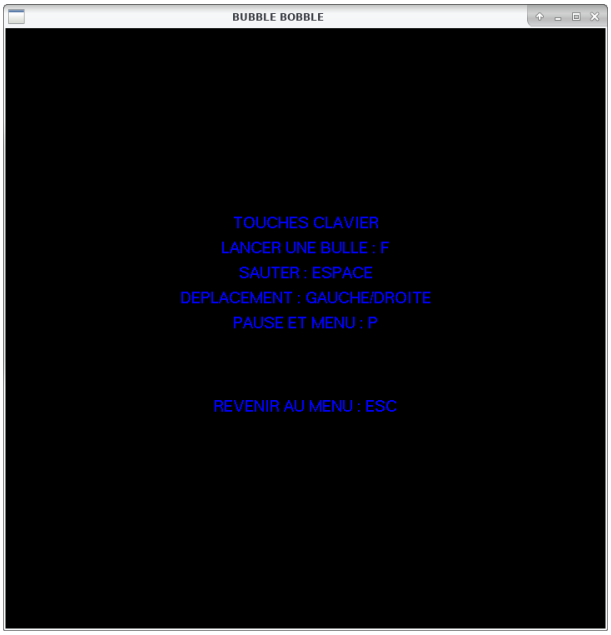
Cet écran, accessible par le menu avant une partie, permet de sélectionner la difficulté du jeu.

2.6 Règles du jeu



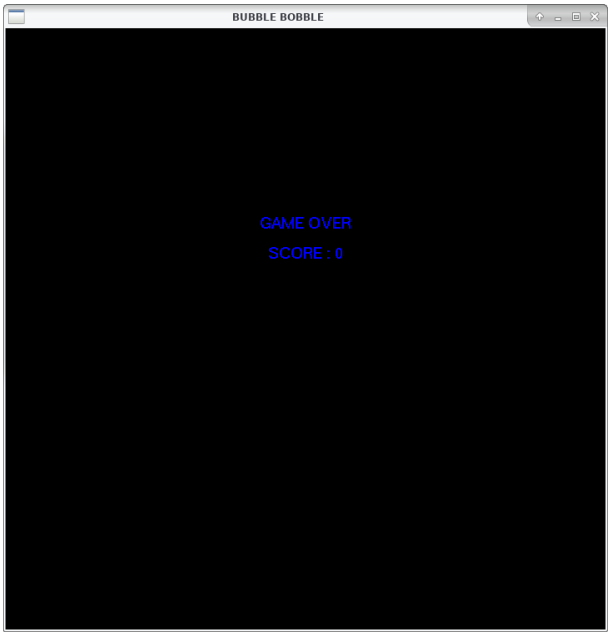
Cet écran, accessible depuis le menu, affiche simplement un texte qui explique le fonctionnement du jeu.

2.7 Touches



Dernier écran accessible depuis le menu qui permet de découvrir les touches clavier utilisées pour jouer.

2.8 Game over



Cet écran ne s'affichera que durant quelques secondes à la fin d'une partie. Le programme sera ensuite redirigé vers le MENU.

3 Modules et fonctions

NB : Les données qui apparaissent dans le fichier d'interface sont indiquées par un .h en exposant dans leur titre.

3.1 *main*

Ce fichier ne contiendra que la fonction "main" dans laquelle seront appelées toutes les fonctions nécessaires à glut, auxquelles sont associées si nécessaire leur implémentation réalisée dans le module *graphics*. Ce n'est pas un module à proprement parler et ne contient évidemment pas de fichier d'interface `.h`.

3.2 *graphics*

Ce module contient les éléments "structurels" du programme. On y gère plusieurs éléments :

- l'implémentation de glut, avec toutes les fonctions nécessaires à son fonctionnement.
- l'affichage et la gestion du menu
- les fonctions de rendu des différents écrans du programme et l'organisation entre eux (redirection du "flux" du programme).

3.2.A DONNEES

3.2.A.a Constantes

BUTTON_WIDTH On définit ici la largeur des boutons du menu principal.

BUTTON_HEIGHT On définit ici la hauteur des boutons du menu principal. Cette hauteur est calculée en fonction du nombre de choix (MAIN_MENU_ITEMS) et de la hauteur de fenetre (WINDOW_HEIGHT dans *map*) pour obtenir six boutons plus sept espaces d'1/5 de la hauteur entre chacun.

DEFAULT_FONT La police de caractère GLUT par défaut.

FONT_HEIGHT La hauteur de la police de caractère GLUT par défaut.

TIMER_FUNC_DELAY On définit ici la durée (en ms) entre chaque appel de la fonction d'affichage principal (`display()`) par la fonction `timer_game()`.

NEW_LEVEL_DELAY Cette constante définit la durée (en nombre de cycles d'une durée de TIMER_FUNC_DELAY ms) de l'affichage du numéro entre les niveaux dans `game()`.

GAME_OVER_DELAY Cette constante définit la durée (en nombre de cycles d'une durée de TIMER_FUNC_DELAY ms) de l'affichage de l'écran "Fin de partie" par `game_over_screen()`.

TIMER_SCREEN_DELAY Définit la durée d'affichage des écrans affichés via `timer_screen()`.

MAIN_MENU_ITEMS Nombre de choix dans le menu principal.

NUMBER_DIFFICULTIES Nombre de "niveaux" de difficulté.

3.2.A.b Types de données

enum State

```
typedef enum{
  MENU,
  PLAYING,
  GAME_OVER,
  GAME_WON,
  INSTRUCTIONS,
  ENEMIES,
  DIFFICULTY,
  SHORTCUTS,
  CHOOSE_LEVEL,
  PAUSE

}State;
```

Cette énumération, dont on définira une instance comme variable globale, contiendra l'état actuel du jeu. C'est grâce à elle que la boucle d'affichage principale (`display()`) détermine quelle fonction appeler.

struct Store

```
typedef struct{
  int difficulty_choice;
  int menu_choice;
  State game_state;
  int game_slide;
  int level_delay;
  int screen_delay;
  int score;
  int number_enemies;

}Store;
```

Simple structure de "rangement" pour les variables d'état du module, définies ci-dessous, dans la partie variables globales.

3.2.A.c Variables globales

static Store store C'est donc la seule véritable variable globale du module, instance de la structure Store décrite ci-dessus. Elle sera initialisé par la fonction `init_graphic_store()`;

State game_state C'est la variable du type `State` décrit ci-dessus, qui garde en mémoire l'état actuel du jeu.

int game_slide Cette variable nous sert à réaliser l'animation de "glissement" au début de chaque partie. Elle détermine la translation verticale à effectuer avant l'affichage de la carte et des personnages. Elle est initialisée à la hauteur totale de l'écran et redescend jusqu'à 0 durant l'animation. Elle est réinitialisée à la fin d'une partie pour que l'animation se répète à chaque fois.

int screen_delay Fonctionne un peu comme la précédente : elle est initialisée en début de partie et décrémentée jusqu'à zéro par chaque passage dans la fonction `timer_screen()`, et détermine le nombre d'itérations de cette fonction, et donc la durée de l'affichage des écrans qui passent par elle (GAME_OVER et GAME_WON).

int level_delay Permet de gerer la temporisation de l'affichage du message "NIVEAU <X>" à chaque nouveau niveau.

int score Stocke le score obtenu par le joueur lors de la dernière partie pour l'afficher en fin de partie.

int difficulty_choice Stocke le niveau de difficulté courant.

int menu_choice Stocke le choix courant dans le menu principal.

int number_enemies Stocke le choix du nombre d'ennemis pour la partie.

3.2.B FONCTIONS PRINCIPALES

3.2.B.a Interface (.h)

Ces fonctions sont nécessaires au fonctionnement de glut et données en paramètres dans le *main*. Elles sont prototypées dans le fichier d'interface, ainsi que la fonction `game_over()`, qui doit pouvoir être appelée depuis le module *game*.

void init(void)

void display(void) Cette fonction est la boucle principale du jeu, elle est rappelée en continu (toutes les x millisecondes) par la fonction `timer_game()`. Elle consiste en un nettoyage de l'écran et à l'appel, selon la valeur la variable `game_state`, de la fonction correspondante à l'état du jeu.

void timer_game(int time) C'est elle qui se charge de rappeler tous les x millisecondes (valeur de la constante TIMER_DELAY) la fonction d'affichage principale `display()`. Elle est donnée comme paramètre dans le main à la fonction `glutTimerFunc()`.

void game_over(void) C'est la fonction qui est appelée lorsque le joueur perd une partie. Ici on s'occupe de réinitialiser tous les éléments des entités (grâce à la fonction `init_entities()` du module *entity*) ainsi que les autres éléments nécessaires au début d'une nouvelle partie, comme par exemple les variables globales `store.game_slide` et `store.screen_delay`.

3.2.B.b Implementation (.c)

void menu_game(void) Fonction qui est appelée au démarrage du programme, et à la fin de chaque partie. De là on a accès à tous les écrans du jeu en cliquant sur le bouton approprié. Correspond à l'écran "Menu".

void game(bool pause) Correspond à l'écran "Jeu". Cette fonction est celle qui affiche le jeu à proprement parler. Elle lance l'affichage de la carte et des personnages. Son paramètre `bool pause` permet de mettre le jeu en mode "pause". Ce mode consiste en l'affichage de la carte, mais sans les personnages, dont le mouvement est arrêté.

void game_over_screen(void) Affiche durant quelques secondes l'écran indiquant que le joueur a perdu une partie, puis remet l'état du jeu sur "MENU". Correspond à l'écran "Game Over".

void game_over(void) Fonction appelée quand le joueur a perdu une partie. C'est ici qu'on nettoie et qu'on réinitialise les entités pour une prochaine partie.

void game_won(void) Gestion de la fin du jeu par le joueur (quand il a terminé les 10 niveaux).

3.3 *entity*

3.3.A DONNEES

3.3.A.a Constantes

MAX_ENEMIES$_{.h}$ Nombre maximum d'ennemis (utilisé par *graphics* pour l'écran du choix de nombre d'ennemis).

DEFAULT_NUMBER_ENEMIES$_{.h}$ Nombre d'ennemis par défaut.

DRAGON_START_X, DRAGON_START_Y Position du dragon au début d'une partie.

ENEMY_START_X, ENEMY_START_Y Position que prendra un ennemi au début d'un niveau

BUBBLE_LIFESPAN Cette valeur sert à initialiser la variable Bubble.lifespan. Elle détermine la durée de vie d'une bulle lancée par un dragon.

NUMBER_LIVES Nombre de vies du dragon.

NEW_LIFE_DELAY Le nombre de tours entre le moment où le joueur est replacé après la "mort" et avant qu'il puisse être atteint par les ennemis. Cela permet au joueur de ne pas mourir directement lors de son repositionnement si un ennemi se trouve justement à cet endroit là.

DEATH_STAGES Correspond au nombre de "tours" (itérations de `display()` entre le moment de la mort du dragon et son repositionnement pour poursuivre le jeu. Elle détermine la durée de l'animation de la "mort" de l'entité.

AGONY, REPLACED, ITEMIZED, DEAD, ALIVE Valeur des différents états que peut prendre la valeur `Entity.state`.

START_POINTS, POINTS_DECREASE, MIN_POINTS Utilisées pour le calcul des "points" qu'un ennemi va donner au dragon quand il les mangera. Respectivement la valeur de départ, la diminution à chaque tour de boucle du jeu, le nombre minimum de points.

ENTITY_SIZE Hauteur/largeur des carrés représentant les entités.

JUMP Nombre de tours de boucle de jeu que durera un saut.

MOVE_V, MOVE_H Longueur du déplacement (vertical et horizontal) en pixel des entités à chaque tour de boucle de jeu.

BUBBLE_V, BUBBLE_H Longueur du déplacement (vertical et horizontal) en pixel des bulles à chaque tour de boucle de jeu.

3.3.A.b Types de données

struct Bubble

```
typedef struct Bubble{
  int x;
  int y;
  int lifespan; // "duree de vie" courante (en tours de boucle de jeu) de la
  ↪ bulle
  char direction;
  bool empty; // Permet de savoir si une bulle est déjà "occupée" par une
  ↪ entité
```

```
  struct Bubble *next;

}Bubble, *BubbleList;
```

struct Entity

```
typedef struct Entity{
  int x;
  int y;
  char direction;
  bool moving;  // Vrai si entité en mouvement (pour les touches
  int jump_stage;// permet de savoir si le personnage est en train de
  ↪ sauter, et si oui, ou il en est dans le saut
  bool drag;    // vrai si l'entité est un dragon
  int state; // permet de gerer l'animation de la mort des entités, et la
  ↪ transformation des ennemis
  GLuint texture_id; // texture a appliquer a l'entite
  int score;         // score du dragon, valeur en points des ennemis
  int lives;         // nombre de vies (pour le dragon)
  Bubble *bubbledTo; // pointeur vers la bulle dans laquelle l'entité est
  ↪ enfermée.
  BubbleList bubbles;  // la liste des bulles que le dragon a lancéés
  struct Entity *next;
  struct Entity *previous;

}Entity, *EntityList;
```

L'implémentation de cette structure de donnée est inspirée d'une vidéo [1], et son utilisation dans le cadre d'un jeu vidéo inspirée d'un article ([2]). Nous avons opté pour utiliser la même liste chaînée pour le dragon et pour les ennemis. Cela nous amène à utiliser certaines variables dans la structure qui ne servent que pour le dragon, mais permettent de très grandement simplifier l'implémentation des différentes fonctions destinées à afficher, déplacer, sauter, entre autres.

struct Store

```
typedef struct{
  GLuint tex_bad_dead; // Texture des ennemis transformés en banane
  GLuint tex_bad;      // Texture des ennemis
  EntityList entities; // Liste chainée contenant les entités
  Entity *dragon;      // simple pointeur vers le dragon (idem que entities,
  ↪ mais rend le code plus clair)
  int enemies_alive;   // Nombre d'ennemis en vie
  int number_enemies;  // Nombre d'ennemis au total
  int difficulty;      // niveau de difficulté courant

}Store;
```

3.3.A.c Variables globales

static EntityList entities Ce pointeur contiendra l'adresse de la liste de toutes les entités (dragons et ennemis).

3.3.B FONCTIONS PRINCIPALES

3.3.B.a Interface (.h)

Toutes les fonctions suivantes sont dans l'interface car elles seront appelées par différentes fonctions du module *graphics*.

char get_lives(void) Récupère le nombre de vies restantes du joueur. Cette fonction sera utilisée par la fonction `game()` pour afficher les informations sur la partie en cours.

int get_scores(void) Idem que la précédente, mais pour le score actuel du joueur.

void create_entities(void); Crée les entités, cette opération est effectuée une première fois par la fonction `init()` au début du jeu. Elle est réutilisée par `set_number_entities()` en cas de modification du choix du nombre d'ennemis (via l'ecran ENEMIES).

void jump_drag(void); Fait sauter le dragon. Elle est appelée par une des fonctions de gestion des évènements claviers.

void launch_bubble(void); Fait lancer une bulle par le dragon en ajoutant une élément à la liste pointée par `Entity.bubbles`. Associée à la touche *ad hoc*.

void set_direction_drag(char direction) Dirige le dragon en attribuant la bonne valeur à `Entity.direction`. Associée aux touches gauche et droite.

void set_moving_drag(bool moving); Fonction qui active ou désactive le mouvement du dragon via la variable booléenne `Entity.moving`. Elle est associée au relâchement des touches de direction, afin d'éviter que le dragon ne continue à bouger quand elles sont relâchées.

void init_entities() Remet les entités à leur état initial. Appelée à la fin d'une partie pour remettre tout à zéro pour la prochaine partie (position des personnages, score, vies, bulles, ...).

void clean(); Libère toutes les allocations mémoires réalisées pour les entités et les bulles associées (via la fonction `clean_entities()`).

void set_number_enemies(int n); Recrée les entités du jeu avec n (entier positif) ennemis.

void manage_entities(); Cette fonction est courte mais se charge d'appeler les différentes fonctions de déplacement, de gravité, d'intelligence des ennemis pour chaque item de la liste chainée des entités à chaque passage dans la boucle de jeu.

3.3.B.b Implémentation (.c)

void gravity(Entity *e); Fonction qui gère la "gravité". Elle fait "tomber" les personnages en permanence, sauf lorsqu'ils sautent, qu'ils sont dans une bulle, ou qu'ils se retrouvent sur la "terre ferme" (déterminé grâce aux fonctions `isOnSolidGround()` et `isWall()` du module *map*).

bool touch_any_entity(Entity *e, EntityList l); Vérifie qu'une entité en touche une autre de la liste des entités grâce à `touch_entity()`.

bool touch_entity(Entity *e, Entity *e2); Teste si une collision a lieu entre deux entités particulières.

void kill_dragon(void); Gère la mort du dragon.

void display_entity(Entity *e); Affiche une entité à l'écran. C'est fonction est très importante. Selon l'état de l'entité, elle détermine la texture à appliquer et sa direction. C'est ici également qu'est gérée l'animation de la mort des entités.

void display_entities(void) Affiche les entités (dragon(s) et ennemis).

void move_entity(Entity *e); Déplace l'entité en modifiant sa position x. Si l'entité est "enfermée" dans une bulle, elle prend la position de la bulle en question, c'est donc le déplacement de la bulle qui la fait bouger.

void jump_entity(Entity *e); Fait sauter une entité.

void add_bubble(Entity *e) Crée une bulle en l'ajoutant à la liste chaînée de bulles pointée par `Entity.bubbles`. Appelée par `launch_bubble()`.

void move_bubble(Bubble *b); Gère le déplacement des bulles. Elle effectue un déplacement horizontal durant la moitié de la vie de la bulle (BUBBLE_LIFESPAN), puis vertical l'autre moitié du temps.

void clean_bubbles(Entity *e); Supprime (et libère la mémoire allouée) toutes les bulles attachées à une entité.

void display_bubble(Bubble *b); Se charge de dessiner une bulle (à partir de la fonction trouvée ici [3]) à l'écran.

3.3.C FONCTIONS PROPRES A L'INTELLIGENCE DES ENNEMIS

void randomize_entity(Entity *e); Cette fonction sert au niveau "facile" : elle fait sauter les ennemis au hasard selon une probabilité.

void entity_jump_ia(Entity *e); Quand le dragon se trouve au-dessus d'une entité, cette fonction va vérifier via `any_platform_over()` que l'ennemi se trouve en dessous d'une plateforme. Si c'est le cas, l'entité va effectuer un saut. Cette fonction est utilisée pour le niveau difficile.

void set_direction_ia(Entity *e); Cette fonction utilisée pour le niveau difficile amène les ennemis à se diriger vers le dragon quand elles se trouvent à son niveau.

void selected_ia(Entity *e); Va se charger de déterminer les fonctions à utiliser par `manage_entities` pour l'intelligence des ennemis en fonction du niveau de difficulté choisi.

3.4 *tga_small*

Les textures sont chargées à partir d'images TARGA (.tga). Nous avons choisi ce format pour la simplicité de sa structure. Il n'est plus très utilisé aujourd'hui, mais est parfaitement adapté pour les images très simples nécessaire à la reproduction d'un ancien jeu vidéo. Il contient un canal "alpha" qui permettra de masquer le "contour" de la texture.

Ce module, très largement inspiré d'un article sur le site de David Henry [4], comportera les fonctions nécessaires pour générer les textures à partir des images.

Un fichier image au format TARGA contient un en-tête (header) contenant les différentes différentes informations sur l'image (sa taille, le nombre de bits par pixel, taille, origine, type de compression), suivi des informations sur chaque pixel.

3.4.A DONNEES

3.4.A.a Constantes

ALPHA_R, ALPHA_G, ALPHA_B Canaux de la couleur qui sera transformée en canal alpha lors de la transformation de l'image en texture.

3.4.A.b Types de données

struct gl_texture_t

```
struct gl_texture_t
{
  GLsizei width;
  GLsizei height;
  GLenum format;
  GLint internalFormat;
  GLuint id;

  GLubyte *texels;
};
```

Nous utiliserons ce type de donnée pour stocker toutes les informations nécessaires à la génération d'une texture GLU.

struct tga_header_t

```
struct tga_header_t
{
  GLubyte id_length;
  GLubyte colormap_type;
  GLubyte image_type;

  short cm_first_entry;
  short cm_length;
  GLubyte cm_size;

  short x_origin;
  short y_origin;

  short width;
  short height;

  GLubyte pixel_depth;
  GLubyte image_descriptor;
};
```

Cette structure recevra toutes les informations sur l'image lues dans son "en-tête" (header).

informations sur l'image (sa taille, le nombre de bits par pixel, taille, origine, type de compression), suivi des informations sur chaque pixel.

3.4.B FONCTIONS PRINCIPALES

3.4.B.a Interface (.h)

GLuint loadTGATexture (const char *filename) Unique fonction nécessaire à l'extérieur du module, elle prend en argument un fichier TGA et renvoie un id de texture GLU.

3.4.B.b Implémentation (.c)

static void GetTextureInfo (const struct tga_header_t *header, struct gl_texture_t *texinfo) Récupère les informations à partir de l'en-tête de l'image.

static struct gl_texture_t *ReadTGAFile (const char *filename) Renvoie une structure gl_texture_t à partir d'un fichier image TGA.

static void ReadTGA8bits_add_alpha (FILE *fp, const GLubyte *colormap, struct gl_texture_t *texinfo) Fonction appelée par ReadTGAFile et qui va transformer en texture 32 bits les informations codées dans l'image en "palette des couleurs" 8 bits : 3 canaux pour les composantes Rouge, Verte et Bleue, ainsi qu'un canal Alpha, qui transforme en canal ALPHA (qui gère la transparence) une couleur déterminée par ALPHA_R, ALPHA_G, ALPHA_B).

3.5 *map*

3.5.A DONNEES

3.5.A.a Constantes

MAP_HEIGHT[.h] Nombre de cases en hauteur.

MAP_WIDTH[.h] Nombre de cases en largeur.
Ces deux premières constantes doivent correspondre à ce qui se trouve effectivement dans les fichiers textes contenant le dessin des cartes.

CELL_SIZE[.h] La hauteur/largeur des cases (carrées) de la carte.

WINDOW_WIDTH[.h] Calcule et stocke la largeur totale de la fenêtre.

WINDOW_HEIGHT[.h] Idem pour la hauteur.

NUMBER_LEVELS Le nombre de niveaux (cartes différentes) du jeu.

3.5.A.b Variables globales

Nous définissons ici deux variables globales :

char current_map[MAP_HEIGHT][MAP_WIDTH] un tableau à deux dimensions (hauteur et largeur) qui recevra les données de la carte à afficher, récupérées à partir d'un fichier texte.

GLuint texWall un "entier openGL" (GLuint) qui recevra l'id de la texture à appliquer dans la carte en cours.

static int level Stockera le niveau courant

static char* levels_tab[] Tableau qui contient les chemins vers les fichiers (textes) servant à dessiner les cartes pour chaque niveau (une ligne par niveau).

static char* textures_tab[] Tableau qui contient les chemins vers les textures des cases pour chaque niveau (une ligne par niveau).

3.5.B FONCTIONS PRINCIPALES

3.5.B.a Interface (.h)

int get_level(); Renvoie simplement le niveau courant.

int next_level(); Passe au niveau suivant (+ chargement carte et texture).

void reset_level(); Remet le jeu au premier niveau (+ chargement carte et texture).

void draw_map() Cette fonction utilise les données contenue dans le tableau à deux dimensions (current_map) pour dessiner la carte sur l'écran. Elle sera appelée à chaque tour de boucle du jeu.

bool isWall(int x, int y) Renvoie après vérification dans la matrice de la carte si une position (x,y) correspond ou non à un mur.

bool isPlatform(int x, int y); Idem que `isWall()` mais renvoie vrai si la coordonnée correspond à une plateforme.

bool isSolid(int x, int y); Idem que les deux fonctions ci-dessus, mais renvoie vrai si la coordonnée correspond à un mur extérieur OU une plateforme.

3.5.B.b Implementation (.c)

void read_map() Nous retrouverons ici tout ce qui est nécessaire à l'importation des "cartes" ou "niveaux" du jeu, dessinées dans de simples fichiers texte, dans un tableau à deux dimensions.

void draw_wall() Dessine une case de mur, reçoit en argument les coordonnées du coin haut-gauche de la case

4 Stratégie des ennemis

Nous n'avons malheureusement pas pu aller très loin dans l'implémentation de l'intelligence des ennemis. Dans la version facile du jeu, les ennemis ne changent de direction que lorsqu'ils rencontrent un mur, et ils sautent de façon totalement aléatoire.

Nous avons ensuite implémenté une version difficile avec deux comportement "intelligents" :

- les ennemis changent de direction lorsqu'ils se trouvent au même niveau que le dragon, pour se diriger vers lui
- ils sautent lorsque le dragon est au-dessus d'eux et qu'ils se situent en desssous d'une plateforme.

Une des difficultés rencontrée consistait à garder un comportement "individuel" aux ennemis en même temps que leur programmer une intelligence. Etant donné qu'ils démarrent tous plus ou moins de la même position, ils se retrouvent en effet très vite en file indienne lorsque leur déplacement s'oriente vers le dragon, ce qui rend le jeu peu attractif. Il aurait peut-être été nécessaire d'individualiser chaque ennemi en lui conférant un comportement propre, pour éviter cet écueil.

Pour amener une autre possibilité de complexifier le jeu, nous avons décidé d'implémenter la possibilité de choisir le nombre d'ennemis présents durant la partie.

5 Conclusion

Nous avons opté pour une seule liste chaînée reprenant les différentes entités, aussi bien les ennemis que les dragons. Nous n'étions pas tout à fait certains que cette option était la bonne. Nous étions poussés par le fait que de nombreuses fonctions sont identiques pour les deux types de personnages. Cela permettait dès lors de simplifier grandement le code en utilisant le même pour les différents types d'entité. Il nous a semblé que ces avantages contrebalançaient les quelques désavantages liés à l'utilisation d'une seule et même structure, à savoir l'utilisation de données inutiles (comme le nombre de vies, ou le pointeur vers la liste chaînée des bulles pour les ennemis) et l'obligation de tester à chaque fois un booléen pour déterminer l'opération adéquate quand elle n'est pas la même pour les ennemis et les dragons.

A posteriori, cette option n'est qu'à moitié satisfaisant. Il aurait fallu remodeler une partie du module *entity* pour implémenter la possibilité d'un deuxième joueur.

Pour la carte, nous travaillons sur de simples fichiers textes, dans lesquels les murs sont indiqués par des '#', et les plateformes par des '='. Cela permet d'avoir directement lors de l'édition un aperçu clair de leur rendu, même s'il est quelque peu hors de proportion, puisque les caractères sont plus hauts que larges, alors que nos cases sont carrées. Il s'est avéré que cette méthode rendait très aisée la création de nouveaux niveaux pour le jeu. Nous en avons créé 9, copiés sur la version originale du jeu [5].

Ces cartes sont ensuites stockées dans le programme au sein d'un tableau à deux dimensions. Vu la dimension réduite de la carte, il nous semblait inutile d'utiliser une structure plus complexe. Cette façon de faire s'est avérée relativement facile à implémenter, la principale difficulté résidant dans la gestion des collisions des entités entre elles et avec les élements de la carte.

References

[1] FormationVidéo. Langage C #19 - listes. https://www.youtube.com/watch?v=FmaNOdbngLc.

[2] Jackson Allan. Using Linked Lists to Represent Game Objects - General and Gameplay Programming - GameDev.net. https://www.gamedev.net/articles/programming/general-and-gameplay-programming/using-linked-lists-to-represent-game-objects-r2041/.

[3] Drawing a circle using glut... https://cboard.cprogramming.com/c-programming/57934-drawing-circle-using-glut.html.

[4] David Henry. Charger des images TGA. http://tfc.duke.free.fr/coding/tga.html.

[5] Adamdawes.com - Bubble Bobble level guide. http://www.adamdawes.com/retrogaming/bbguide/index.html.