RAPPORT

Il était une fois la vie

Imac Tower Defense

Clara Daigmorte - Laurine Lafontaine - IMAC 1

Algorithmie - OpenGl

SOMMAIRE

INTRODUCTION

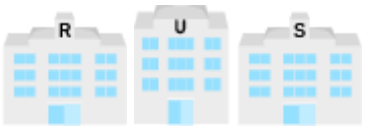
Notre objectif était de réaliser un prototype de tower defense au sein d'une très grande entreprise de jeu vidéo : IMACGAMING.
Le but d'un jeu tower defense est de protéger un terrain en empêchant des vagues d'ennemis de le traverser, à l'aide de tours.

L'environnement de notre jeu se base sur le monde de la santé avec comme éléments principaux du jeu : des microbes et des anticorps. Il existe deux types de monstres : les virus et les bactéries. Les tours représentent les anticorps, les médicaments et les pansements. Un système monétaire a été implémenté, l'argent (représenté par des médicaments de type gélule) permet au joueur de payer ses tours et bâtiments.
Tout cela est très largement inspiré du dessin animé "Il était une fois la vie".
C'est pourquoi, nous avons proposé à la division marketing de renommer ce jeu "Il était une fois la vie" qui a rapidement accepté (les droits d'auteurs seront gérées par le service communication et droit de l'entreprise).

Ce programme comporte une partie algorithmique en codée C ainsi qu'une partie infographie en OpenGL. Afin que le projet fonctionne correctement, vous aurez besoin d'installer le package GLUT, OpenGL, SDL, SDL_Image et SDL_Mixer (tout est très bien expliqué dans le README du repo github !).

PRÉSENTATION PROJET

Protégez vous des bactéries et des virus en vous aidant des anticorps afin de vous protégez de la maladie !
Supprimez 50 vagues de microbes afin de vous guérir de toute maladie potentielle.



Exécution de l’application

Placez vous dans le dossier principal, où se situe le makefile. Ouvrez le terminal et tapez la commande “make”.
Attention, il faut penser à bien avoir un dossier obj s’il n’est pas créé.
Tapez ensuite la commande ./bin/itd ./data/map01.ppm OU ./bin/itd/map02.ppm afin de pouvoir commencer à jouer. Map01.ppm correspond à la map de niveau 1 et map02 à celle de niveau 2.

Si l’envie vous prend, vous pouvez ajouter une carte dans ./images/maps (de préférence une map de 600x600) ainsi que son itd associé dans ./data. Vous pourrez ainsi probablement tester notre jeu avec votre propre map !

ARCHITECTURE DU PROJET

Architecture

bin/
Exécutable

data/
Les fichiers itd qui contiennent les informations de la carte

doc/
Sujet et rapport

images/
info/
Images pour afficher l'information sur les tours et les installations
installations/
Sprite des installations
maps/
.ppm des cartes
monsters/
Sprite des monstres
towers/
Sprite des tours
ui/
Images de toutes les autres sources qui permettent l'affichage de l'interface

include/
Fichiers .h

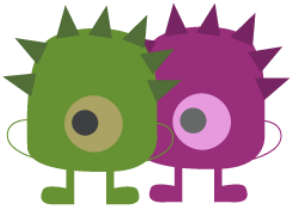
lib/
Contient le dossier des librairies utilisées pour la précompilation

obj/
Fichiers objets pour la compilation

son/
Sons utilisés dans le jeu

src/
Fichiers sources du projet

makefile
readme



Structure logicielle du projet

main.c	Le fichier main.c rassemble les fonctions des structures ci-dessous en un programme fonctionnel.
tower.h tower.c	Ces fichiers contiennent la structure de la tour et toutes ces fonctionnalités.
monster.h monster.c	Ces fichiers contiennent la structure des monstres et toutes leurs fonctionnalités.
node.h node.c	Ces fichiers contiennent la structure des noeuds de la map itd, ils sont reliés sous la forme d'une liste chaînée.
map.h map.c	Ces fichiers gèrent la structure map et toute son initialisation.
installation.h installation.c	Ces fichiers contiennent la structure des installations et toutes leurs fonctionnalités.
sprite.h sprite.c	Ces fichiers ces fichiers gèrent le chargement et affichage des sprites.
colors.h colors.c	Ces fichiers contiennent la structure couleur.
common.h common.c	Ces fichiers contiennent les fonctionnalités communes à de nombreux autres fichiers.
display.h display.c	Ces fichiers contiennent toutes les fonctions qui touchent à afficher des visuels dans le jeu (carte, déplacement de monstre, …).
game.h game.c	Ces fichiers contiennent une structure game ainsi que les fonctionnalités touchant au jeu et à l'argent.
image.h image.c	Ces fichiers contiennent la structure image et tout ce qui touche à la lecture et l'enregistrement de ces données.

Bibliothèques utilisées

Afin de vous guider tout du long, il faut du texte, et quoi de mieux que la bibliothèque **GLUT** pour s'en occuper ? C'est celle -ci qui permet d'afficher l'argent en temps réel, et le temps de jeu passé.

Pour une meilleure immersion dans le jeu, nous avons décidé de rajouter du son (bruitage de tirs et mort des monstres), et en particulier de la musique de générique d'Il était une fois la vie. Pour cela il nous a fallu rajouter la bibliothèque **SDL/Mixer**.

Afin de nous rendre la vie plus facile et de pallier les problèmes sur les ordinateurs de l'école, nous avons ajouté SDL_MIXER à la précompilation. Même si c'est une mauvaise pratique pas vraiment recommandé, il en était nécessaire pour ce projet.

ORGANISATION DU TRAVAIL

Tout projet mérite une organisation méticuleuse !
Nous nous sommes d'abord créé un document **Word Drive** afin de nous organiser et prioriser les tâches à réaliser.
Tout au long du projet, nous nous sommes régulièrement concertée par rapport à l'avancée du projet. En particulier, grâce à l'utilisation de **Github** qui nous a grandement aidé pour ce suivi.

Voici le lien de notre **GitHub** : https://github.com/LafLaurine/imac-tower-defense.git

De plus, nous nous sommes organisées de nombreux moments pour se retrouver afin de coder ensemble. Ainsi, nous avons pu réfléchir à deux aux problèmes rencontrés : cela nous a permis d'avoir plusieurs visions sur un même algorithme. Ces deux points de vue nous ont souvent aidé à nous corriger l'une l'autre.
Cet arrangement nous a permis de travailler de manière efficace, en sachant ce que chacune devait faire.

Nous nous sommes partagés le travail ainsi :

- Clara s'est occupée de l'algorithmique, de la validation de la map, des tours et des installations
- Laurine s'est occupée de l'algorithmique, de l'affichage et design du jeu (sonore et visuel) et des monstres

Le reste des tâches a surtout été réalisé en commun. Comme par exemple, la structure du projet, les structures le composant et leurs fonctions (tours, installation, nodes, …).

Lors de ce projet, nous avions un cahier des charges à respecter. Voici ci-dessous, le récapitulatif de ces fonctionnalités réalisées (ou non).

Récapitulatif des fonctionnalités obligatoires

TÂCHE	RÉALISÉE ?
Réalisation d'une structure ordonnée de projet avec système de compilation intégrée	Oui
Chargement d'une carte	Oui
Création du graphe des chemins	Oui

Vérification de la jouabilité de la carte	Oui
Sprites de bâtiments, de tours, de monstres	Oui
Création des structures de bâtiments, de tours, de monstres	Oui
Ajout, suppression et édition de bâtiments et de tours	Oui sauf édition
Déplacement des monstres	Oui mais pas selon le chemin le plus court (Dijkstra codé mais non fonctionnel)
Gestion des actions des tours et bâtiments	Oui
Gestion du temps	Oui

En plus de ces fonctionnalités obligatoires, nous avons pu ajouter quelques fonctionnalités bonus.

Récapitulatif des fonctionnalités bonus

BONUS	RÉALISÉ ?
Accélération des monstres	Oui
Affichage des informations des bâtiments (tours et installations) au hover	Oui
Affichage du rayon de la tour	Oui
Affichage barre de vie du monstre	Oui
Affichage tirs des tours	Oui
Génération de sons	Oui

FONCTIONNALITÉS DE L'APPLICATION

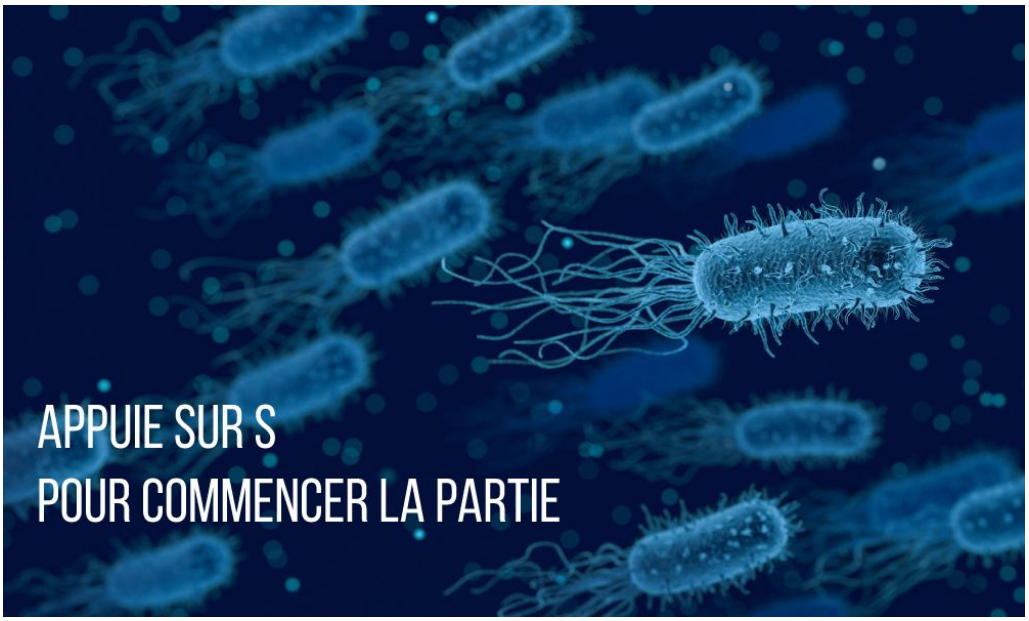
Règles du jeu

Le but du jeu est de tuer tous les monstres qui arrivent par vague de 10. Si l'un des monstres parviens à la sortie, alors la partie est perdue. L'interface affiche le nombre de vagues de monstres envoyées. L'argent du joueur est aussi affiché sur cette même interface. Cet argent évolue en fonction de la partie, en fonction des actions effectuées : mort d'un monstre (augmentation), construction de tours et construction d'installations (diminution).

Déroulement d'une partie

Voici pas à pas comment se déroule une partie de notre jeu "Il était une fois", avec vous comme joueur.

Vous arrivez sur une page incroyable de présentation, avec la sublissime musique. Dessus, il est indiqué d'appuyer sur "S" pour commencer la partie. Pourquoi ne pas appuyer sur cette touche pour voir ?

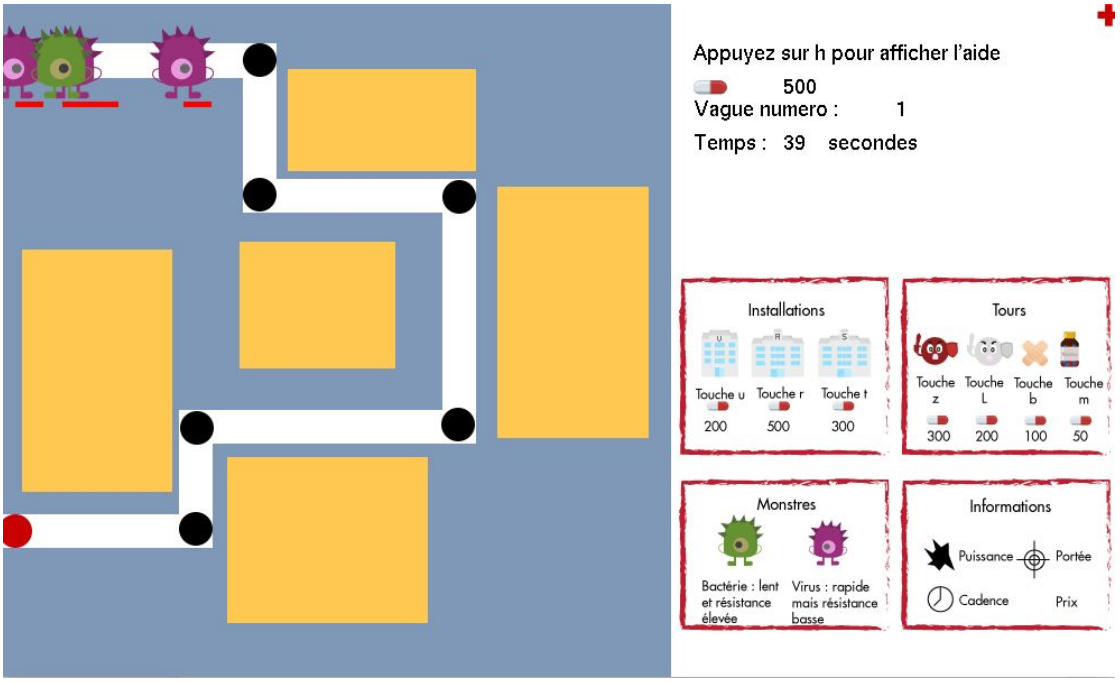


Écran titre du jeu

Vous admirez cette page quelques instants, puis vous appuyez sur la touche "S". Une carte s'affiche alors, celle-ci dépend en fait de la carte passée en argument. Deux niveaux de jeu sont disponibles : map01.itd correspond au niveau 1 tandis que map_02.itd correspond au niveau 2.

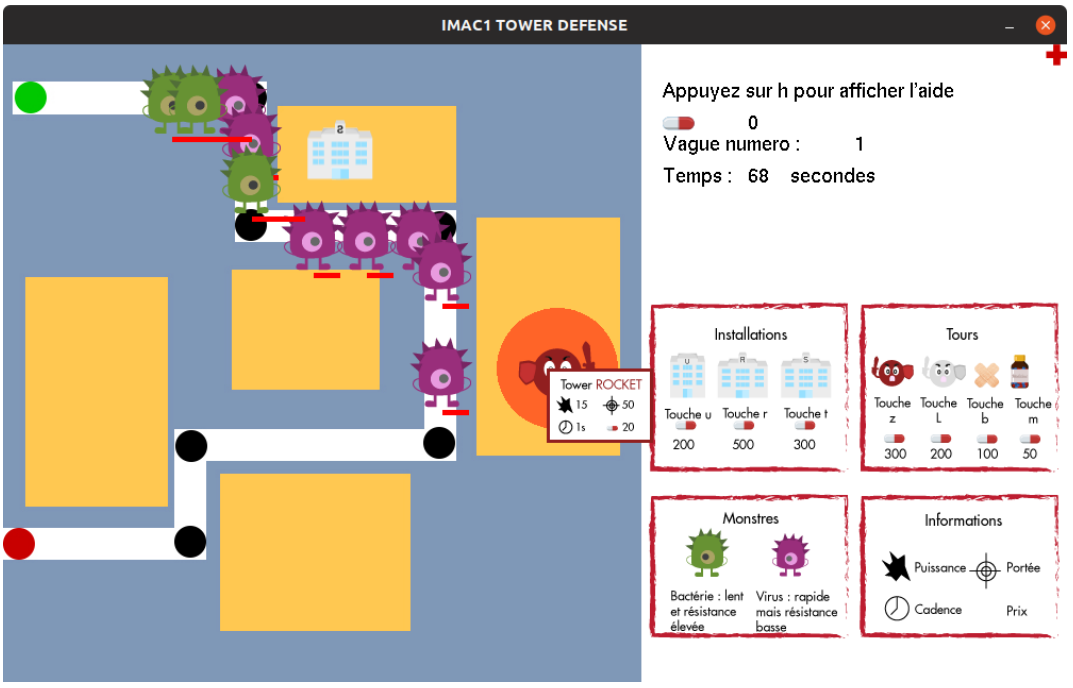
Le jeu démarre ensuite directement.
En appuyant sur la croix rouge, sur q ou sur esc, vous pouvez quitter le jeu à tout moment (ne partez pas tout de suite !).

La première vague de monstre va de son point de départ, au point final en passant par le chemin constitué des autres noeuds. Les monstres arrivent par vague de 10. Afin de les contrer, vous pouvez appuyer sur “Z”, “L”, “B” ou “M” afin de créer une tour. Chacune de ces touches correspondent à un type de tour, respectivement : globule rouge, globule blanc, bandage et médicament. Vous pouvez construire les tours sur les zones constructibles affichées en jaune. Sur votre droite, vous pouvez observer l’interface qui affiche les touches pour les installations et les tours ainsi que leur prix. Il y a aussi des informations sur les deux types de monstres ainsi que sur la signification de certains symboles que nous verrons plus tard.



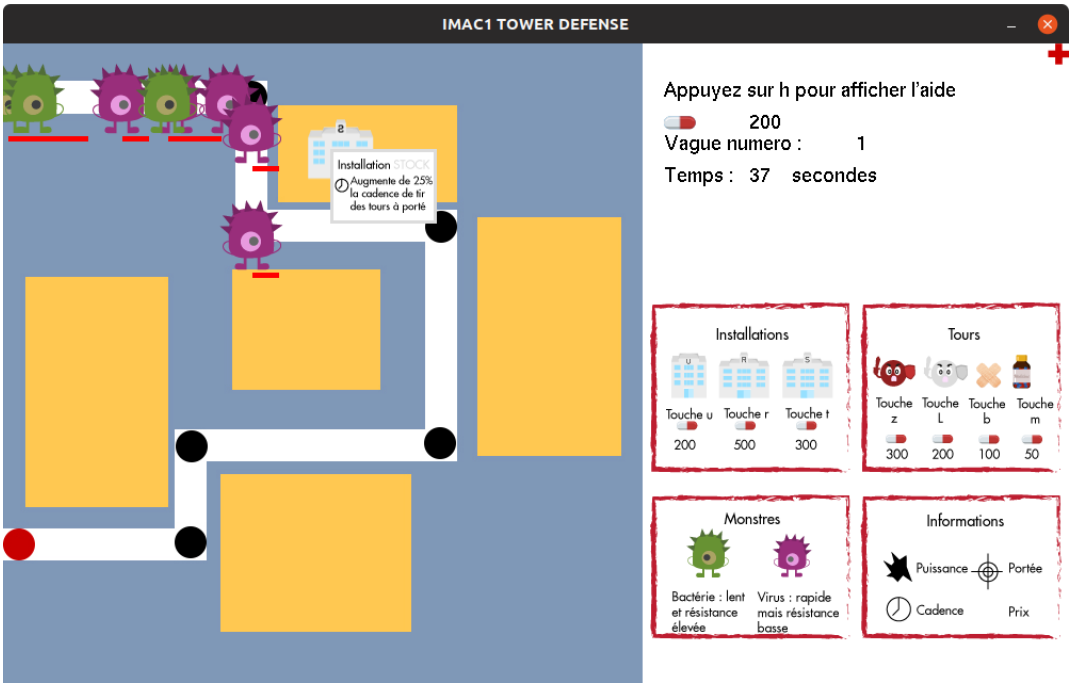
Phase 1 du jeu

Vous pouvez accéder aux propriétés d’une tour créée (cadence, puissance, portée et prix) en passant la souris dessus.



Informations sur la tour rocket

Pour renforcer ces tours, vous pouvez créer des installations en appuyant sur “U”, “R” ou encore “T”. Les installations ajoutent des bonus aux tours : le radar (“R”) permet d’augmenter de 25% la portée de la tour. L’usine d’armement (“U”) permet d’augmenter de 25% la puissance des tours. Enfin, le stock (“T”) permet d’augmenter de 25% la cadence de tir des tours. Comme pour les tours, au survol des installations est affiché la propriété de celle-ci.
Évidemment, pour acheter tous ces éléments, vous devez posséder assez d’argent.



Informations sur l’installation stock

Un clic droit sur une tour ou une installation permet de supprimer/vendre un de celles-ci. Cette suppression permet de récupérer l'argent dépensé à l'achat.

Enfin, si vous souhaitez un jeu plus rapide, il y a possibilité d'accélérer le mouvement des monstres grâce à "A".

Chaque monstre tué rapport 5 unités de médicaments (argent) au début du jeu. A chaque nouvelle vague, les monstres deviennent plus résistants aux tours et ont plus de points de vies. De ce fait, leur mort permet de gagner deux fois plus d'argent (l'argent des monstres calculé est nombre de vagues/2 * argent monstre).

D'ailleurs, il est aussi possible de mettre le jeu sur pause en appuyant sur "P".

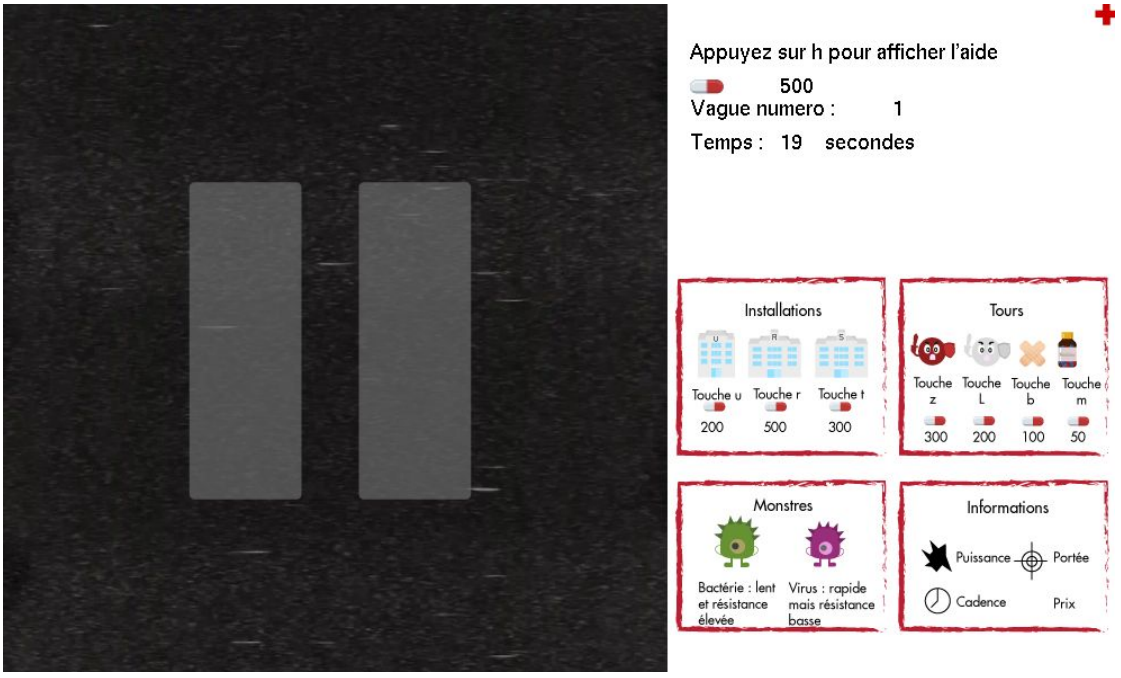


Illustration du jeu sur pause

Après avoir admiré les monstres se déplacer d'un bout à l'autre de la carte avec agilité, vous vous rendez compte que vous avez oublié de placer des tours et donc de tuer cette vague de monstre. Ils arrivent au bout de la carte et un nouvel écran s'affiche : PERDU.

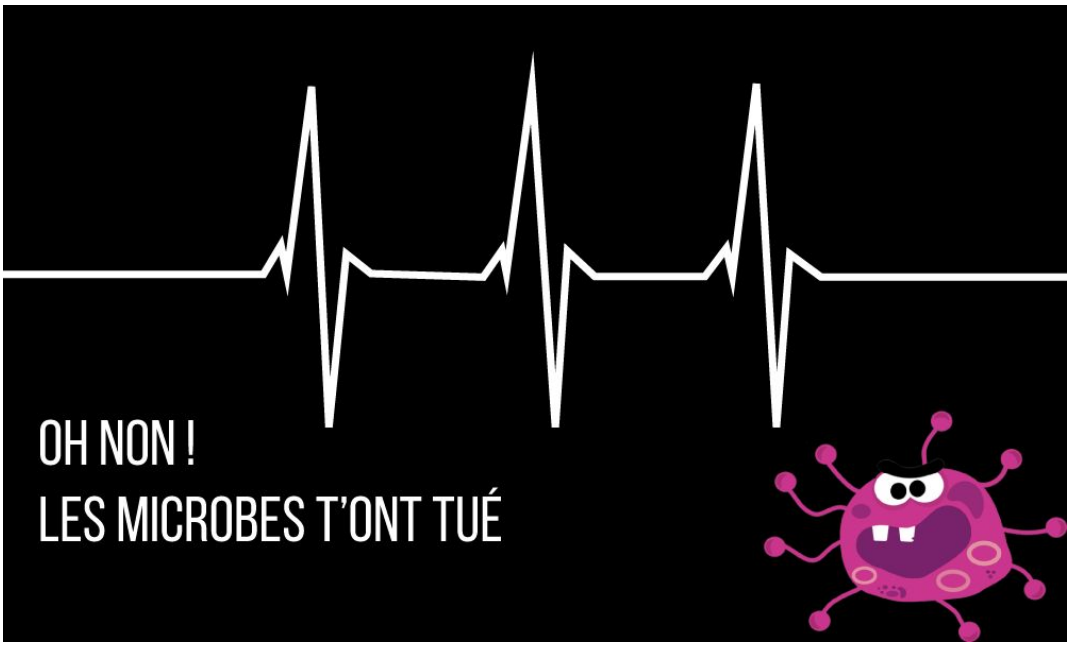


Illustration fin du jeu

Mais revenez jouer ! Peut-être après un peu d'entrainement pourrez vous voir l'illustration de fin de jeu gagnante après 50 vagues de monstres (attention spoiler : sinon elle est directement dans le dossier ./images/ui).

CPU

```
real    2m34,630s
user    1m8,436s
sys     0m2,215s
```

itd	laurine	21	23733	89,4 MiB	57,0 MiB	4,0 KiB

Grâce à la fonction time en bash, nous pouvons voir ce que le jeu consomme après avoir un peu joué.

Dans le gestionnaire des applications, nous voyons que l'itd consomme 21% du CPU.

ÉLÉMENTS DU PROGRAMME

Nous avons créé un fichier pour chaque élément du projet. C'est à dire qu'à chaque élément important du jeu est associé un .c et un .h

Noeud (node.c et node.h)

Les **noeuds** correspondent aux points qui tracent les chemins de la carte. Il peuvent être de 4 types : *entrée, sortie, coude* et *intersection*. Nous définissons la structure des **noeuds** (Node) avec les attributs suivants : *coordonnée en x, coordonnée en y, type de noeud, tableau d'entier contenant les successeurs de ce noeud, pointeur sur noeud suivant* et *pointeur sur noeud précédent.*

On déclare aussi une structure pour les **listes de noeuds**, afin de pouvoir récupérer tous les noeuds (ce qui nous sera utile plus tard, afin de construire la carte, nos structures de tours , de monstres,...). En effet, nous basons la création de graphe sur les **listes chaînées** et non sur les matrices.
Dans la structure **List_Node**, nous retrouvons comme attribut *la taille de la liste, une pointeur vers le premier élément de la liste (de type Node)* et *un pointeur vers le dernier élément de la liste (de type Node aussi).*
Voici les fonctions liés à la structure Node

- **Initialiser la liste des noeuds *:*** *List_node* new_List_Node()*
- **Créer un noeud :** *Node* create_node(Node_Type type, float x, float y, int* successors, List_Node* l_node, int index)*
- **Ajouter un noeud à la liste :** *void add_node_list(Node *n, List_Node* list_node)*
- **Supprimer un noeud de la liste :** *List_Node* remove_node(List_Node* current_node, Node* current)*
- **Supprimer tous les noeuds de la liste :** *void free_all_node(List_Node* list_node)*

Lecture du PPM (image.c et image.h)

La carte utilisé pour ce projet est une image de type **PPM 3**. Une des premières étapes de ce projet est donc de lire cette carte afin d'obtenir toutes les informations nécessaires sur le terrain de jeu. Il faut vérifier que le PPM est bien de type 3 et que l'en-tête est bien composée de commentaires, de la hauteur et largeur de l'image ainsi que du nombre magique 255.

La structure **Image** est constituée d'un entier pour la hauteur, d'un entier pour la largeur, d'un tableau de caractère pour stocker le chemin de cette image et d'un tableau de pixels.

Voici les fonctions liées à l'image :

- **Lire l'en-tête d'un fichier PPM :** *Image* readPPMHeader(FILE* fp, int *w, int *h)*
- **Lire le PPM :** *Image* read_image(char *filename)*
- **Libérer la mémoire :** *void free_image(Image *img)*

Carte (map.c et map.h)

La **carte** est décrite sous le format **itd**. Il faut parcourir ce fichier afin de vérifier l'intégrité de la carte. Ce fichier contient le nom de la carte, la couleur du chemin, la couleur des noeuds, la couleur des zones constructibles, la couleur des zones d'entrées et de sorties. S'en suit de la description des noeuds avec le nombre total puis une description de chacun : leur type, leur position en x, leur position en y et leur successeur(s).

La carte contient différents noeuds (voir la partie précédente) qui correspondent à des entrées, sorties, coudes et intersection. Afin que le jeu fonctionne correctement, ses noeuds doivent être vérifiés, pour être sûr que les chemins soient des tracés de couleur blanc par exemple.

La structure map est constituée d'un pointeur sur la structure Image, de la couleur du chemin de type Color3f (structure définie dans color.h) et de même pour le noeud, les zones de constructions, le noeud d'entré et le noeud de sortie. La structure contient aussi un entier non signé qui corresponds au nombre de noeud ainsi qu'un pointeur sur la structure List_Node.

On doit vérifier ligne par ligne que le fichier .itd correspond au fichier .ppm de l'image chargée. Pour cela, nous utilisons notamment les fonctions de gestion de fichiers.
La première ligne à vérifier est celle de "@ITD".
On vérifie ensuite la présence de commentaire. Un commentaire commence par un #, c'est donc grâce à cela que l'on peut l'identifier. Nous récupèrons le chemin de l'image et nous construisons cette image. Nous utilisons la fonction readImage déclarée dans le image.c.
On affecte à l'attribut img de la map la nouvelle image créée.
Le but de la suite va être de vérifier que chaque composante de la map : noeud, zone de constructions, noeud d'entrée, noeud de sortie et chemin possède la bonne couleur. Qu'est-ce que la bonne couleur ? C'est la couleur qui est inscrite dans l'itd. Nous vérifions que les composantes R,G,B sont bien entre 0 et 255. Si jamais les couleurs ne satisfassent pas ces conditions, alors nous utilisons la fonction change_color qui est de type Color3f et qui permet d'affecter à un type Color3f de nouvelles composantes r,g,b. Ce qui nous permet donc de corriger l'image si jamais elle est fausse.

Voici les fonctions liées à la carte :

- **Vérification de la map :** *int check_map(Map* map, char* map_itd);*
- **Vérifier que les chemins soient continus :**
  int check_segment_X(int x1, int y1, int x2, int y2, Map map);*
  int check_segment_Y(int x1, int y1, int x2, int y2, Map map);*

Ces deux fonctions utilisent l'algorithme de Bresenham. Nous nous devons de l'expliquer un peu. Ces fonctions vérifient entre deux noeuds chaque couleur de pixel avec celle du chemin ou d'un noeud. Si ces couleurs correspondent, le chemin est correct, sinon il y a un "trou".
check_segment_X vérifie pour un chemin en diagonale X. *check_segment_Y* vérifie pour un chemin en diagonale Y. L'algorithme détermine donc quels sont les points d'un plan discret qui doivent être tracés afin de former une approximation de segment de droite entre deux points donnés. Bresenham nous permet de vérifier qu'entre deux noeuds, il y a bien un chemin !

- **Vérification couleur du pixel :** *int check_pixel(int x, int y, Map* map, Color3f color);*
- **Initialisation de la carte :** Map* init_map (char* path);
- **Libérer la mémoire associée à la carte :** void free_map(Map* map)

Jeu (game.c et game.h)

Le jeu est défini par une structure game que contient un entier pour l'argent, un entier pour gérer le début du jeu, un entier pour la victoire, un entier pour mettre le jeu en pause et un entier pour la défaite. A part pour l'argent, les entiers jouent le rôle de booléen car ils ne prennent que la valeur 0 ou 1 (exemple : jeu débuté ou non, jeu en pause ou non,...).

En réalité, c'est grâce à cette implémentation que notre projet prend un aspect jouable.

Voici les fonctions implémentées dans cette structure :

- **Initialisation d'une nouvelle partie :** *Game *new_game()*
- **Mettre à jour l'argent du joueur (l'augmenter) :** *int player_money_up_update (Game* game, int cost)*
- **Mettre à jour l'argent du joueur (le diminuer) :** *int player_money_down_update (Game* game, int cost)*
- **Finir la partie :** *void game_end (Game* game)*

Monstre (monster.c et monster.h)

Monster est une structure qui contient les coordonnées du monstre (x, y), ses points de vies et points de vie maximum, sa vitesse de déplacement, move pour gérer son déplacement, son MonsterType, le noeud suivant ce monstre, le monstre précédent, le monstre suivant et l'argent qu'il rapporte une fois tué, son type et sa résistance. Un enum gère les deux types de monstres : BACTERY et VIRUS.
Nous avons aussi créé une liste de monstre pour tous les répertorier et pouvoir les contrôler. Cette liste est une structure qui contient un pointeur vers le premier monstre de la liste, un pointeur vers le dernier monstre de la liste, un entier qui gère le nombre de monstres total et un entier qui gère le nombre de monstres envoyés.
Enfin, nous retrouvons aussi une structure pour la vague, qui comporte un pointeur sur List_Monster, d'une taille maximale de 10 ainsi que du nombre de listes (pour le nombre de vague).

- **Création d'une liste monstre :** *List_Monster* new_monster_list();*
- **Création d'un monstre :** *Monster* create_monster(Monster_Type type, float x, float y, Node *node_next, List_Monster* l_monster);*
- **Ajouter un monstre à la liste :** *void add_monster_list(Monster* m, List_Monster* list_monster);*
- **Tuer un monstre de la liste :** *Monster_Type kill_monster(List_Monster* list_monster, Monster* current);*
- **Enlever un monstre de la liste :** *List_Monster* remove_monster(List_Monster* current_m, Monster* current);*
- **Enlever tous les monstres de la liste :** *void free_all_monster(List_Monster* l_monster);*

Tour (tower.h et tower.c)

Tour est une structure. Elle contient les coordonnées d'une tour (x, y), son type, sa cadence, sa puissance de tir, sa portée, son coût, ainsi que son noeud suivant et précédant, et la tour qui la suit / précède.
Il existe aussi une liste de tours qui contient toutes les tours créées. Elle fonctionne sous le même principe que pour les monstres

Voici les fonctions qui ont été créées pour gérer ces tours.

- **Initialisation de la liste de tours :** *List_Tower* new_tower_list();*
- **Création d'une tour :** *Tower* create_tower(TowerType type, float x, float y, Node* head, List_Tower* l_tower);*
- **Ajout d'une tour à la liste de tour :** *void add_tower_list(Tower* t, List_Tower* list_tower);*
- **Suppression d'une tour de la liste de tour :** *List_Tower* delete_from_position(List_Tower* list_tower, Tower* current);*
- **Repère s'il y a eu clic sur une tour :** *Tower* click_tower(List_Tower* p_ltower, float x, float y);*
- **Destruction d'une tour :** *void destroy_tower(List_Tower* list_tower);*
- **Vérifie si le centre de la tour est sur la zone de construction** : *int tower_on_construct(Map* map, int posX, int posY);*

Installation (installation.h et installation.c)

Installation est une structure très ressemblante à tour. Elle contient ses coordonnées (x, y), son type, son coût, ainsi que l'installation qui la suit / précède.
Il existe aussi une liste d'installations qui contient toutes les installations créées. De même encore une fois, cela est semblable à Tour.

Voici les fonctions qui ont été créées pour gérer ces installations.

- **Création d'une liste d'installation :** *List_Installation* new_installation_list();*
- **Création d'une installation :** *Installation* create_installation(InstallationType type, float x, float y, List_Installation* list_inst);*
- **Repère s'il y a eu un clic sur une installation *:*** *Installation* click_installation(List_Installation* l_install, float x, float y);*
- **Supprime installation de la liste :** *List_Installation* delete_install_from_position(List_Installation* l_inst, Installation* current);*
- **Ajout d'une installation à la liste :** *void add_installation_list(Installation* t, List_Installation* list_installation);*
- **Libère une installation :** *void destroy_installation(List_Installation* list_installation);*
- **Repère si clic pour créer installation est sur une zone construct :** *int installation_on_construct(Map* map, int x, int y);*

Textures (sprite.h et sprite.c)

Dans un jeu où est lié algorithme et infographie, les textures ont une place très importante. L'identité graphique d'un jeu est ce qui en fait l'unicité. C'est grâce à ces fonctions implementées (surtout load_sprite qui est tellement utilisé) que notre jeu est beau !
Voici les fonctions qui gèrent les sprites.

- **Repère si l'image est chargée :** *int is_loaded(SDL_Surface *image);*
- **Charge la texture de la carte :** *SDL_Surface* load_map_texture(Map* map, GLuint *texture);*
- **Charge un sprite :** *SDL_Surface* load_sprite(char* file_name, GLuint *texture);*
- **Libère un sprite :** *void free_sprite(GLuint texture, SDL_Surface* img);*

Affichage des éléments (display.h et display.c)

L'affichage des éléments c'est fait grâce à display.h et display.c. Ces fichiers contiennent de nombreuses fonctions qui permettent l'affichage d'éléments sur l'interface de jeu. Comme par exemple :

- **Affichage de la carte :** *int display_map(GLuint* texture);*

Cette fonction existe aussi pour l'affichage des listes de monstres / tours / installations, de l'argent et un peu pour beaucoup de choses en fait. Si les textures étaient importantes, eh bien l'affichage l'est encore plus. Évidemment, charger des textures dans le vide n'a aucun

intérêt. Nous avons même réussi à afficher un petit laser lors d'un tir de la tour, nous sommes plutôt satisfaites !

Commun (common.h et common.c)

Mention spéciale à ces fichiers, qui implémentent les fonctions communes, régulièrement utilisées :

- **Repérer une intersection :** *int is_intersect(float x1, float y1, float x2, float y2, float r1, float r2);*
- **Écrire et afficher le texte :** *void vBitmapOutput(int x, int y, char *string, void *font);*

En plus de ces fonctions, common contient aussi de nombreuses fonctions qui calculent l'intersection entre éléments grâce à la fonction is_intersect notamment :

- **Repérer autour d'une tour s'il y a une installation :** *int check_around_tower(Tower* t, List_Installation* list_inst);*

Cette fonction permet de vérifier si autour d'une tour (nouvellement créée) il existe une installation qui est dans sa portée, afin de pouvoir (ou non) obtenir son bonus.

Conclusion

Difficultés rencontrées

La vérification de la carte était une tâche fastidieuse étant donné qu'il fallait lire ligne par ligne le fichier itd, tout en appliquant dessus une batterie de tests. Effectivement, il fallait vérifier de nombreux éléments afin d'être sûres que la carte rentrée en argument est correcte et utilisable pour le jeu.

Viens alors, l'algorithme de Bresenham qui justement fait partie des tests à rédiger pour la vérification de la carte. Il permet de vérifier qu'un chemin est continu entre deux noeuds. La difficulté était surtout de comprendre comment fonctionnait cet algorithme, heureusement Steeve, notre chargé de TD a pu nous aider à ce propos. Finalement nous avons pu réussir cette partie-là.

En restant encore sur la vérification de la carte : l'algorithme de Dijkstra. Ce dernier permet de trouver le chemin le plus court de la carte en vérifiant les successeurs de chaque noeuds. Il nous a été très compliqué de coder cette partie, surtout à cause du fait que nous nous y sommes prises plus tard que prévu. Nous avons attendu d'avoir des vagues de monstres qui se déplaçaient avant de le coder.

Un problème autre que pour la vérification de la carte : supprimer la tour courante, c'est à dire, la tour sur laquelle nous venons de cliquer. Comment faire ? Récupérer les coordonnées sur lesquelles nous venons de cliquer, vérifier si une tour s'y trouve, si oui : la supprimer de la liste et la libérer. Il nous a été compliqué de comprendre comment faire pour couper ces différentes étapes en code fonctionnel.

Il était aussi difficile de voir les projets des autres sans se comparer et désespérer de leur avancée, mais nous avons réussi à surmonter cela et avons pu finalement rendre un projet fonctionnel !

Améliorations à ajouter

Par manque de temps, nous n'avons pas pu concevoir comme nous le souhaitions la carte dans le thème "Il était une fois la vie". Nous avons préféré nous concentrer sur les fonctionnalités obligatoires à coder, plutôt que sur les graphismes. Nous avons pensé que l'équipe graphique de la grande entreprise pourrait nous aider à nous constituer une charte graphique avant de la faire par nous-même.

Nous aimerions que les tours ne dépassent plus sur les routes, tout simplement en utilisant la fonction d'intersection entre une tour et le chemin, mais avec la couleur comme argument.

L'algorithme de Dijkstra est quasiment terminé, mais non fonctionnel et non utilisé dans notre programme de vérification de carte.

On pourrait aussi imaginer une sauvegarde grâce à un fichier texte externe dans un dossier, mais nous n'avons malheureusement pas pu exploiter cette idée !

Ce qui est aussi dommage est le fait que nous n'avons pu régler les mixs des sons, notre jolie générique de "Il était une fois la vie" disparaît avec les bruits des lasers !

En fait, il y aurait tant d'améliorations à faire. Et coder ce jeu nous a vraiment fait plaisir. La dernière amélioration intéressante aurait été de bien gérer différents niveaux et de rendre le jeu plus difficile, car il est très facile à gagner.

Aussi, nous souhaiterions ne pas avoir de segfault en quittant le jeu ! Nous n'avons malheureusement pas encore pu repérer l'erreur dans le code, mais nous pensons que la libération de la mémoire ne se fait pas de manière très appropriée en fin de jeu.

Enseignements de ce projet

Ce projet, qui au début nous semblait insurmontable, nous a beaucoup appris.
Nous sommes maintenant plus à l'aise avec le langage C, et l'OpenGl. Notre compréhension de la source des erreurs s'est agrandie. Et surtout, nous avons compris comment bien répondre à des problèmes posés.

Au début du projet, nous avons directement (trop rapidement) découpé le projet en structure sans trop savoir quoi mettre dedans car nous n'avions pas de vision globale du jeu et de son fonctionnement, étant donné que c'était la première fois que nous nous essayions à coder un jeu pareil. Cette expérience nous a permis de comprendre comment fonctionne réellement un jeu, et nous permettra de ne pas reproduire ces mêmes erreurs une prochaine fois.
Nous sommes fières d'avoir pu réaliser un jeu fonctionnel. Et nous avons hâte de réaliser d'autres projets.

Merci pour votre lecture.
Des remerciements à l'IMAC, nos professeurs et à nos camarades, sans qui nous n'avancerions jamais.