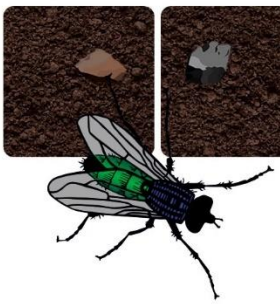
INSECTOWER

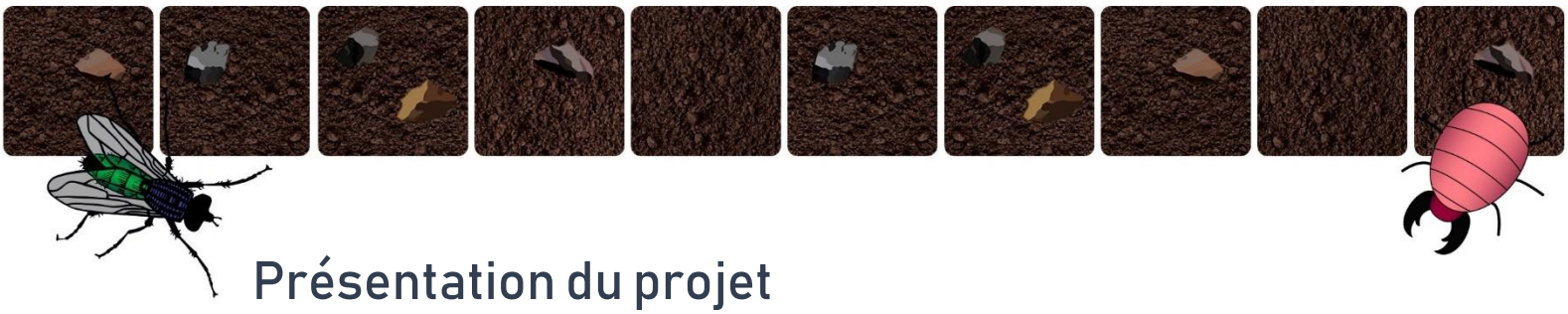
Nicolas Lienart

Laura Dietsch

Antoine Libert




Sommaire



Présentation du projet

Insect Tower est un tower défense autour de l'univers des insectes, où le but est de défendre la reine des fourmis contre des vagues d'envahisseurs. Pour cela, comme dans la réalité, des fourmis feront appel à d'autres espèces d'insectes, qu'elles nourriront en l'échange de leur protection. C'est sur ce thème que nous avons entamé notre projet, en amorçant l'idée d'un univers que nous tenions à développer, afin d'en tirer les différentes fonctionnalités que nous voulions implémenter. Une fois cette base posée, nous nous sommes répartis sur les différents aspects à réaliser pour le mener à bien : Laura s'est orientée sur la gestion des l'affichage des textures, du design et de l'expérience utilisateur, Nicolas s'est d'abord concentré sur la GUI autour du système des cases et des tours, et Antoine s'est davantage plongé dans le gameplay, autour de la gestion des monstres, du plus court chemin et des vagues. Nous avons pu ainsi amorcer le projet sans se retrouver sur les mêmes fonctionnalités. Nous avons tenu à constamment communiquer sur nos avancés afin de nous assurer qu'elles restaient bien concordantes.

Lien vers le repository git : https://framagit.org/nicolaslienart/imac-tower-defense

Architecture du projet

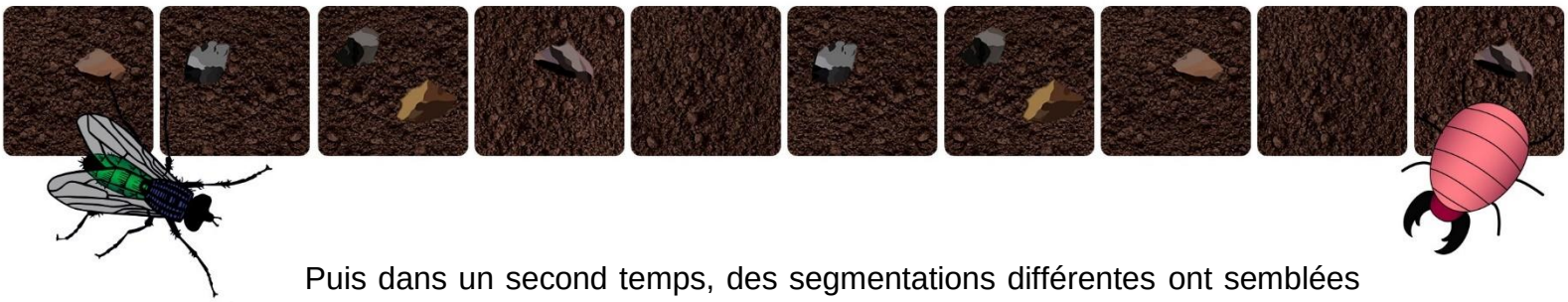
- *Entre Linux et Windows, recours au CmakeList*

Durant le développement du projet, nous n'avons pas travaillé sur les mêmes OS. Nous avons donc exploité les possibilités de Cmake, afin d'avoir un processus de compilation qui s'adapte en fonction de ces deux OS. C'est aussi pourquoi vous trouverez dans le projet des librairies destinées à la compilation sur Windows, ainsi que l'utilisation de #ifdef _WIN32 dans le code, afin de ne pas appeler les mêmes headers en fonction de l'OS (notamment glew au lieu de gl sur Windows).

- *Structure de fichiers*

Le projet comporte tout d'abord différents médias : des sons et des textures, qui ont été placés dans des dossiers distincts (image et sound). Concernant la découpe du programme en différents fichiers sources, nous avons d'abord scindé le code en fonction des différentes composantes nécessaires à l'application :

  - L'affichage (display),
  - La gestion de l'interaction utilisateur (GUI),
  - Le chargement des données (level load),
  - La gestion du gameplay en lui-même (game).



Puis dans un second temps, des segmentations différentes ont semblées nécessaires dans chacune des composantes :

- Level load : en fonction du type de fichier traité, itd ou ppm
- GUI : en fonction du périphérique, souris et clavier
- Affichage : séparation du chargement des sons, de celui des textures, et de leur affichage.
- Game : séparation en fonction des entités manipulées dans le jeu, les monstres, les tours, les projectiles, les cases...

- *Regroupement des structures dans structures.h*

La plupart des structures du programme sont imbriquées dans d'autres structures. Nous avons fait face à des conflits au sein des headers du programme concernant les dépendances des structures : nous voulions déclarer une structure dans un header A, en appelant le header B contenant une structure dont la première dépend. Seulement le header B contenait également une structure dépendant d'une de A. Ces conflits empirant, nous avons choisi de déclarer l'ensemble des structures dans un header à part, auquel nous nous rapportions plus facilement lorsque nous avions besoin des données.

Description des structures de données

- *Choix entre tableau et liste chaînée*

Lors de la création de l'application, nous avions régulièrement à manipuler des ensembles de données du même type : les tours, les monstres, les projectiles, les vagues, des cases… Et nous devions déterminer quelle structure serait la plus adaptée pour les manipuler. Lorsque nous avons dû manipuler un nombre indéterminé de données en les parcourant toutes, nous avons privilégié les listes chaînées. Ce fut le cas pour les monstres et les projectiles, sur lesquels les actions concernaient le plus souvent toutes les entités : les afficher, changer leur position à chaque frame. Cela correspondait d'autant plus à ces données qu'elles étaient volatiles : nous étions amenés régulièrement à ajouter et supprimer des entités.
Mais lorsque nous avions un nombre fini d'éléments sur lesquels les actions ne se faisaient qu'élément par élément, nous avons privilégié le tableau, permettant un accès plus direct. Ce fut le système adopté pour la gestion des cases dans les niveaux, mais également pour le stockage des données de jeu, issue de l'idt et ppm (à l'exception des nodes, car généralement tous parcourus).

Toutefois, il resta quelques données que nous avions tantôt besoin de consulter à la chaîne, tantôt de façon ciblée. C'était particulièrement le cas pour les tours, qui s'inscrivaient dans le système du gameplay, majoritairement dominé par des listes parcourues à chaque frame, et le système de GUI qui, lui, repose essentiellement sur les actions du joueur cliquant sur un endroit précis, pour faire une action sur une tour précise par exemple. Dans ce cas, nous avons opté pour un entre-deux : une liste chaînée de tours, ainsi qu'un tableau de pointeur vers ces mêmes tours. Cela permet la flexibilité dont nous avions besoin.

- *Le recours aux variables globales*

Assez rapidement au cours du développement, nous nous sommes confrontés à une problématique : nous avions certaines données du jeu, comme les sons, les cases, la liste de monstres, dont nous avions besoin dans de très nombreuses fonctions du jeu. Et plus l'encapsulation se complexifiait avec le développement, plus nous avions besoin de passer en cascade ces données en argument, pour les faire parvenir à une énième fonction. Cela alourdissait beaucoup le programme, avec des fonctions ne faisant que transmettre certains arguments à d'autres fonctions...

C'est pourquoi nous avons finalement passé certaines données en variables globales, au sein de structures, afin de limiter les conflits potentiels entre variables locales et globales. Nous sommes curieux de savoir si une autre méthode aurait été préférable dans ce cas de figure. En orienté objet, le passage de ces données aurait sans doute été facilité avec l'encapsulation des méthodes, mais en C, nous n'avons pas trouvé de Design Pattern adéquat pour ces données.

- *Le jeu, le niveau et la partie*

Le système de données du jeu a été pensé à trois niveaux :

- Celui du jeu, c'est à dire du programme, regroupant les données utilisées du début à la fin de l'exécution et qui ne sont pas liées à un niveau de jeu particulier : les textures, les sons, les données liées à la GUI...
- Celui du niveau, c'est à dire les données fixes relatives à un environnement de jeu : les couleurs identifiant les cases, les informations des nodes, des chemins, etc.
- Celui de la partie, c'est à dire les données évoluant au cours d'un niveau jusqu'à ce qu'il se termine : les listes de monstres, tours, projectiles, la vague actuelle de monstres, etc.

Cette dissociation nous a permis de plus facilement gérer la libération dynamique de données, notamment avec la distinction entre niveau et partie : lorsque le joueur gagne ou perd un niveau, il a la possibilité de rejouer ce niveau. Dans ce cas, nous voulions éviter d'avoir à recharger les données de l'itd et ppm, de revérifier la validité des chemins, etc. C'est pourquoi en fin de niveau, nous supprimons les données de partie, mais préservons les données du niveau tant que le joueur ne choisit pas de faire un autre niveau.

- *Recours à de nombreux enum*

Dans ce projet nous avions à manipuler un grand nombre de métadonnées à l'aide de tableaux. Pour mieux s'y retrouver dans l'utilisation et ses données et leur sécurisation, les enum nous ont été d'une grande aide. Permettant à la fois d'identifier avec un nom clair les éléments et en empêchant l'ajout de mauvaises données, ils nous ont facilités l'appel des sons et des textures que nous utilisions notamment, mai aussi de mieux percevoir quel type de tour ou de monstre nous manipulions à certains endroits du code.

Description détaillée des fonctionnalités

- *Règles du jeu*

Notre tower défense dispose de 3 niveaux, leur nom en hommage à 3 grands maitres du tower défense. Ils sont disponibles depuis le menu principal du jeu. Mais il est possible d'en créer ou d'en modifier dans data, il s'ajouterons automatiquement dans le menu principal. Dans chaque niveau, le but est de défendre la reine des fourmis qui subit des vagues d'attaques d'insectes ennemis.

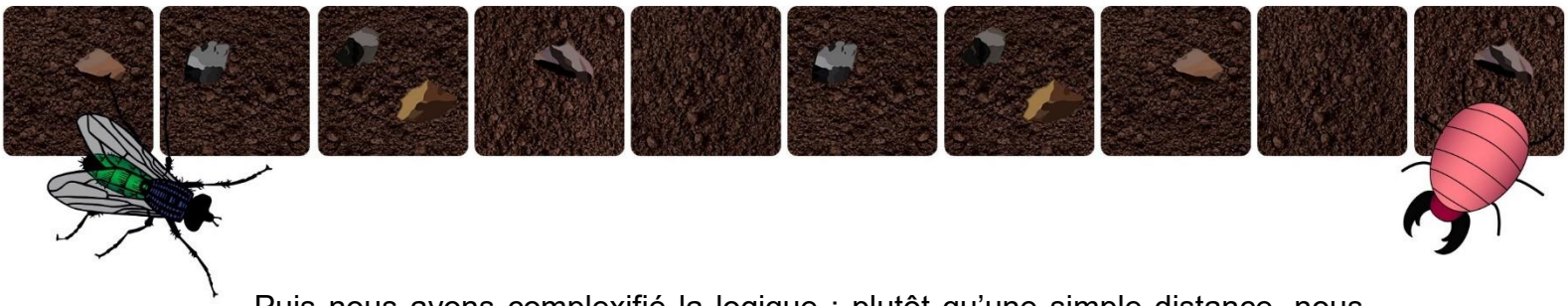
Pour cela, le joueur peut disposer, en dehors des chemins et certaines cases non constructibles, des insectes alliés qui défendront la reine. Il existe deux types d'alliés aux caractéristiques différentes, qu'il est possible de consulter via l'interface du jeu en sélectionnant les tours. Il est aussi possible de placer des artéfacts, qui viendront renforcer la portée, la force, la rapidité ou simplement nourrir les alliés pour qu'ils puissent attaquer. Pour ajouter un allié ou un artefact, il suffit de cliquer sur son icône dans le menu puis sur l'endroit de la carte où l'on veut le placer.

Le bouton en forme de croix permet de passer en mode suppression et de retirer les alliés/artefacts que l'on veut en cliquant dessus. Le point d'interrogation permet également au clic d'afficher les informations d'un ennemi, allié ou artéfact dans le terminal. Le joueur dispose d'une monnaie pour ajouter des alliés/artefacts. Il récupère de la monnaie lorsque des ennemis sont tués ou en retirant des alliés/artefacts.

Les ennemis attaquent par vague, de plus en plus conséquentes, ils peuvent arriver de tous les chemins au bord de la carte. Si un ennemi atteint la reine, le jeu est perdu et le joueur peut rejouer le niveau ou revenir au menu principal. Le niveau est gagné lorsque la dernière vague d'ennemi est terminée et qu'il n'y a plus d'ennemi vivant.

- *Gestion du plus court chemin*

Nous avons d'abord implémenté une version simple de l'algorithme de Dijkstra pour gérer l'identification des plus courts chemins. Celui-ci ne se lançait qu'au chargement du niveau, et ne prenait en compte que les distances séparant chaque node. Comme notre jeu était pensé avec une seule arrivée et plusieurs entrées pour les monstres, l'algorithme calcule l'itinéraire en partant de l'arrivée, ce qui permet en une fois d'avoir le plus court chemin à partir de chaque entrée.



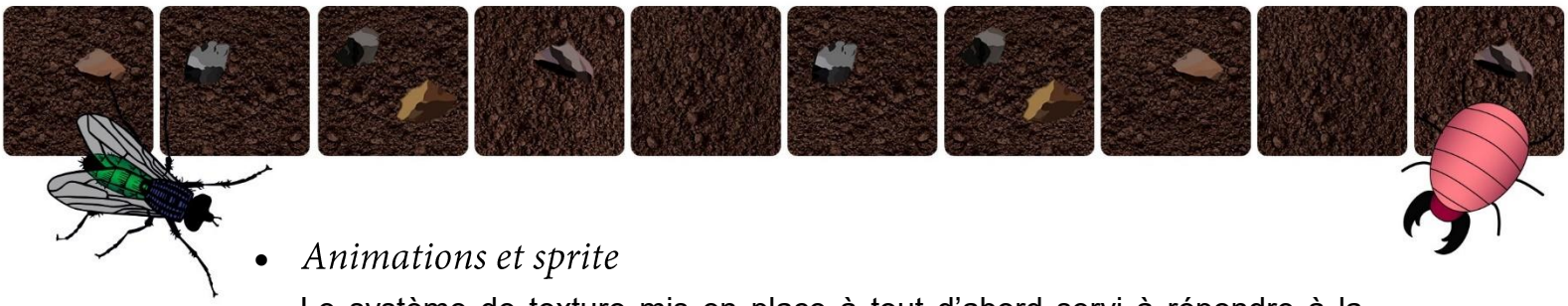
Puis nous avons complexifié la logique : plutôt qu'une simple distance, nous avons considéré la valeur d'un chemin comme une addition de facteur : sa longueur et le nombre de monstre récemment mort sur celui-ci. Lors de l'apparition d'un monstre, celui-ci est attribué aléatoirement à une entrée et il demande son itinéraire à l'algorithme, qui met d'abord à jour les données avant de fournir la liste des nodes que devra choisir le monstre. Nous actualisons ainsi les plus courts chemins à chaque nouveau monstre. Nous aurions pu faire en sorte que les monstres actualisent leur itinéraire à chaque arrivée sur un node, mais nous ne voulions pas qu'ils aient une telle réactivité dans le jeu.

- *Lecture des données itd et ppm*

Lors du lancement d'un niveau, les données de celui-ci sont récupérées en parcourant l'idt : lorsque certains mots clés sont identifiés, le programme sait quel type d'information il doit récupérer. Suite à cela, le programme charge les données du ppm en faisant certaines vérifications : que les couleurs des pixels du ppm correspondent à celles spécifiées dans l'idt, ainsi que les nodes dont l'idt spécifie les connexions, sont bien séparés par des chemins sur le ppm. Les chemins du jeu étant bidirectionnels, l'algorithme vérifie qu'on ne considère jamais deux fois le même chemin lors de cette vérification et par la suite pour le niveau.

- *La gestion de l'interface utilisateur*

Lorsque nous avons fait des croquis nous avons rapidement vu des contraintes qui rendraient le design de la GUI moins efficace, pas difficile à coder mais plutôt très limitant dans l'évolution. C'est pourquoi nous l'avons pensé comme une page HTML avec l'idée d'emboîtement de structures, ou chaque section a un parent, des enfants et des sœurs. On peut la voir comme des poupées russes. Il y a aussi une réflexion basée sur le CSS avec des dimensions classiques mais un positionnement toujours relatif à la section parent. Il en va de même pour les boutons qui à la seule différence n'ont pas d'enfants. Pour l'affichage on part de n'importe quelle section on va pouvoir afficher tout l'arbre visuel, les positions absolues sont rendues abstraites par le programme. Nous avons également des référentiels différents lorsqu'il s'agit d'afficher la GUI ou le plateau de jeu spécifiquement, le dernier est précédé d'une réadaptation du contexte OpenGL et nous offre un raisonnement par case et plus en pixels. Cette façon de concevoir la GUI nous a demandé beaucoup d'efforts à mettre en place mais ils ont été grandement rentabilisés à son utilisation.



- *Animations et sprite*

Le système de texture mis en place à tout d'abord servi à répondre à la problématique d'affichage des monstres. Nous nous sommes permis de les animer et cela repose encore une fois sur un système facile à utiliser et flexible. On va pouvoir donner à chaque monstre, même de type égal, une autonomie d'animation, on obtient ainsi un jeu plus naturel. Lors du chargement d'un sprite pour un monstre, on donne une indication sur le nombre d'éléments en X et Y (c'est flexible) et une durée de double pour le parcours de l'animation. On peut ainsi imaginer un monstre à une texture comme un monstre à 100x23 très aisément. Le plateau est d'ailleurs généré par placement aléatoire de texture et s'adapte à toutes tailles de cartes.

- *Gestion des monstres*

Il est possible d'avoir 4 types de monstres différents par niveau, chaque monstre possède des caractéristiques de points de vie, de résistance, de vitesse et une valeur marchande lors de leur mort. Ces données sont spécifiées dans l'idt et sont donc personnalisables par niveau. Pour chaque monstre ayant un itinéraire fixe lors de sa création, il est possible d'afficher le chemin du monstre en passant la souris dessus.

Les monstres disposent de trois états : vivant, mourant, et mort. Dans l'état mourant le monstre prend une texture diminuant progressivement d'opacité. Pendant cet état transitif le monstre augmente la valeur du chemin sur lequel il se trouve, poussant temporairement les futurs monstres à ne pas emprunter le chemin sur lequel il est mort. Les monstres disposent également d'une jauge de vie, qui ne s'affiche que lorsque de premiers dégâts ont été reçus.

- *Gestion des vagues*

Les vagues sont personnalisables également via l'itd. Elles sont caractérisées par une fréquence de monstres, une variation aléatoire de cette fréquence, un délai relatif avant son lancement et un nombre de monstres pour chaque type. Lorsqu'une vague est en cours, elle crée un nouveau monstre à fréquence régulière, plus ou moins une variation comprise dans un intervalle, cela permet de rendre les vagues moins machinales. À chaque nouveau monstre, la vague pioche aléatoirement son type dans un tableau qui se vide au fur et à mesure. La présence de chaque type de monstre dépend des indications de l'itd. Une fois ce tableau vide, la vague suivante est lancée. Elle attendra la fin de son délai pour créer à son tour des monstres.

- *Les bâtiments*

Les bâtiments à rôle d'amélioration de tours peuvent être placés par le joueur, nous avons fait en sorte de les rendre peu restrictifs sur leurs capacités, c'est à dire qu'un bâtiment peut apporter aux tours une série d'améliorations en même temps. Ils permettent notamment d'allonger leur portée, de les alimenter, d'augmenter leur cadence voire de les rendre plus létales.

Autre point à noter c'est qu'ils ont eux-mêmes une portée mais qu'elle est représentée par une zone carrée. Cette décision se base sur le fait que ce sont sont des cases qui séparent tours et bâtiments, et qu'ils sont immobiles, ce comportement diffère de la relation tour/monstre. Leurs effets s'appliquent à leur placement et incrémente les tours à leur proximité, le contraire survient quand on les supprime.

- *Le système d'attaque via projectile*

Les attaques se font indirectement entre les tours et les monstres. Lorsqu'une tour est chargée en munition, elle émet un projectile dont la structure connaît le monstre ciblé. Le projectile va ensuite se déplacer sur la map en se rapprochant de sa cible, dans un système similaire à celui des monstres avec les nodes. C'est lorsque que le projectile entre dans une zone de collision à proximité de sa cible, qu'il est détruit et que les dégâts sont infligés aux monstres. Si une cible meurt, les projectiles associés sont automatiquement détruits.

Les projectiles suivent dynamiquement les monstres, ils ne cherchent pas à se rapprocher d'un point X et Y de la map, mais questionnent sans cesse le monstre pour s'en rapprocher.

- *Les centrales à énergie*

Pour fonctionner, une tour a besoin d'être alimentée, nous nous sommes donc naturellement orientés vers un système où chaque type de tour nécessite un certain nombre de centrales à sa proximité. Mais une idée plus intéressante a surgi à la fin du projet. En fait comme nous sommes dits que vu que nos tours sont des insectes, on aurait plutôt tendance à les "activer" avec de la nourriture. Dorénavant nos centrales sont des tas de nourritures en apports limités et les tours y puisent une quantité qui leur dépend à l'intérieur, et ce à chaque tir sur un monstre. Les tours choisissent le tas de nourriture le plus proche. Nous avons tenu à ce que le joueur puisse constater la quantité restante dans les tas en réduisant visuellement leur contenance, il peut donc prévoir une pénurie.

- *La gestion de l'évolution du jeu*

Après la création des niveaux, nous avons eu à considérer les menus et l'agencement des niveaux avec eux. L'idée a donc été de définir d'une part une variable d'état qui permet notamment la bascule graphique entre les menus et niveaux, et actionne le chargement d'un niveau ou sa suppression (libération de la mémoire, etc). D'autre part, nous avons dû gérer l'écoulement du temps de jeu, notamment pour gérer la succession des vagues. Nous sommes d'abord parties sur une considération du temps absolue, le début des vagues étant déterminé par une durée les séparant de ce temps de départ. Puis finalement, nous avons trouvé plus judicieux d'aborder le temps de façon plus relative, la vague suivante étant lancé en fonction de la fin de la précédente. C'est ce même système relatif que nous avons mis en place pour la gestion du rechargement des tours, et la décomposition des monstres.

Il est aussi possible de mettre en pause le déroulement du niveau, ce qui suspens l'avancement des monstres comme des vagues.

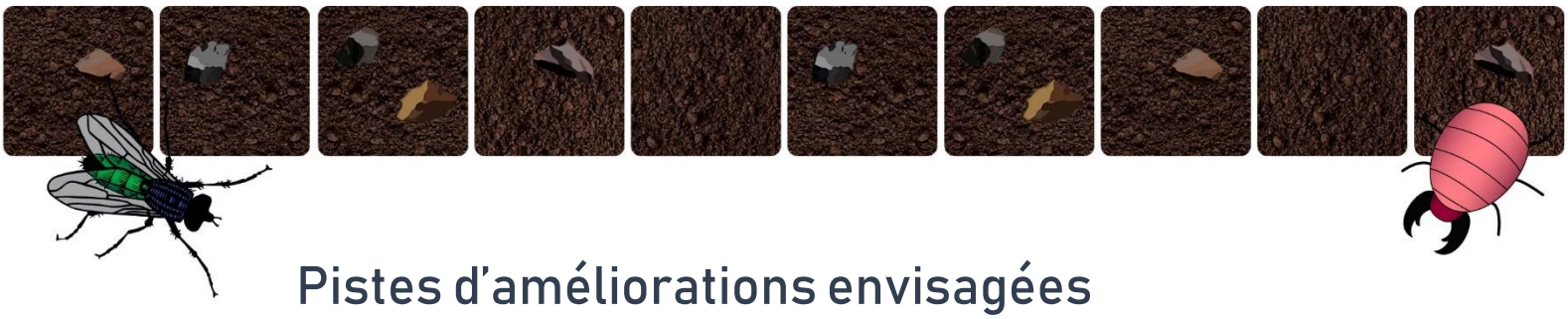
- *Le joueur*

Le jeu, comme demandé, dispose d'un système d'achat, c'est une fonctionnalité qui s'est avéré extrêmement simple à gérer et on permet au joueur de revendre ses tours également pour qu'il puisse réadapter sa stratégie. L'autre aspect qui touche au joueur est son "état", c'est à dire dans quel mode d'interaction il est. On va être parmi l'ajout, l'obtention d'informations ou la suppression de tours, et quel type de tour est actuellement sélectionné.

- *Autres aspects de gameplay*

Nous offrons au joueur la possibilité de révéler l'itinéraire d'un monstre vers son objectif simplement lors de son survol. On a aussi un affichage d'information dynamique sur les tours/bâtiments/monstres qui va s'adapter à l'éléments (ses données et sa nature) au survol et à la sélection par clic d'une tour. En effet l'information est très variable puisque les tours sur le plateau et celles des boutons n'ont pas les mêmes données (notions de connexions aux bâtiments), on a aussi les bâtiments et les différents monstres.

On tient également le joueur informé sur de nombreuses choses comme les interdictions, les positions, des retours sur ses actions. Tout cela par l'intermédiaire de sound design et feedback visuels spécifiques.



Pistes d'améliorations envisagées

Durant la création de ce projet, nous avons collectivement amorcé la réflexion sur un certain nombre de fonctionnalités, que nous n'avons finalement pas eu le temps d'implémenter. Nous vous en partageons ici certaines

- *Un PPM plus intelligent*

Notre jeu ayant pour principe que les chemins au bord de l'écran sont des entrées, il nous aurait été assez simple de déduire automatiquement des entrées parmi les cases chemins. En consultant notre tableau de cases, nous aurions identifié celles présentes sur les colonnes et lignes aux extrémités. De même par l'analyse des cases, nous aurions pu retrouver les nodes et les connexions bidirectionnelles qui les lient. Cet ajout aurait par exemple permis de tracer l'entièreté des chemins d'une seule couleur, à l'exception de la sortie, et ainsi plus facilement proposer à l'utilisateur de concevoir ses propres niveaux.

- *Mieux annoncer les prochaines vagues*

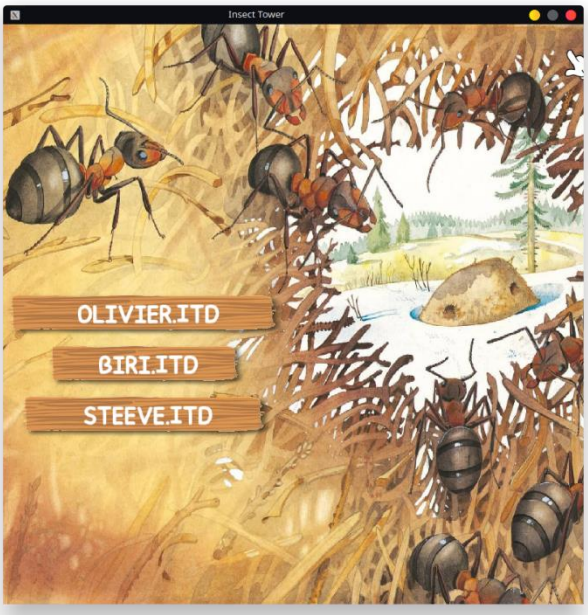
Nous comptions également implémenter un compteur qui à la fin d'une vague annoncerait le temps en seconde avant la prochaine vague. Il aurait suffi pour cela de consulter la structure de la vague courante et de regarder la valeur de son délai. Cela aurait pu s'accompagner d'une mention temporaire à l'écran, annonçant le début d'une nouvelle vague. Le système de section en GUI aurait permis cela facilement.

Nous pensions également afficher une jauge indiquant l'état d'avancement de la vague en cours, avec un ratio entre monstre déjà apparut et monstre restant, et de façon similaire un indicateur du nombre de vagues restantes. Ses infos sont déjà facilement récupérables dans notre structure de données, il ne manquait plus qu'à les afficher.

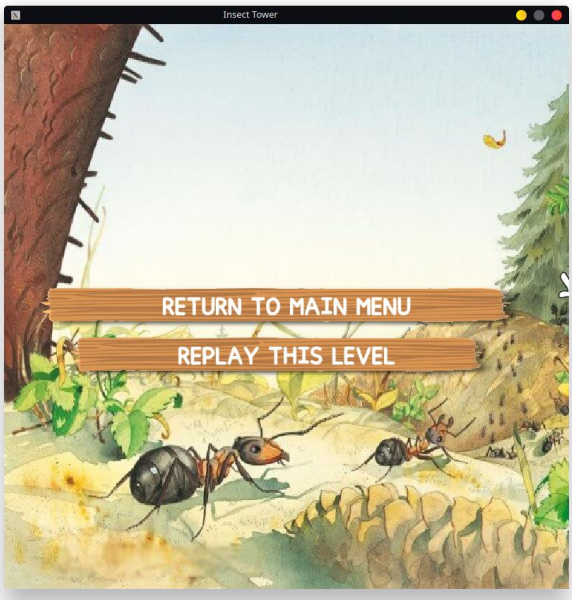
- *L'ajout d'un tutoriel*

Nous aurions aussi aimé ajouter un tutoriel sur la base d'un vidéo accessible depuis le menu principal. Nous aurions enregistré un moment de jeu illustrant son fonctionnement, que nous aurions lancé à l'aide de SDL_video.

Résultats du jeu

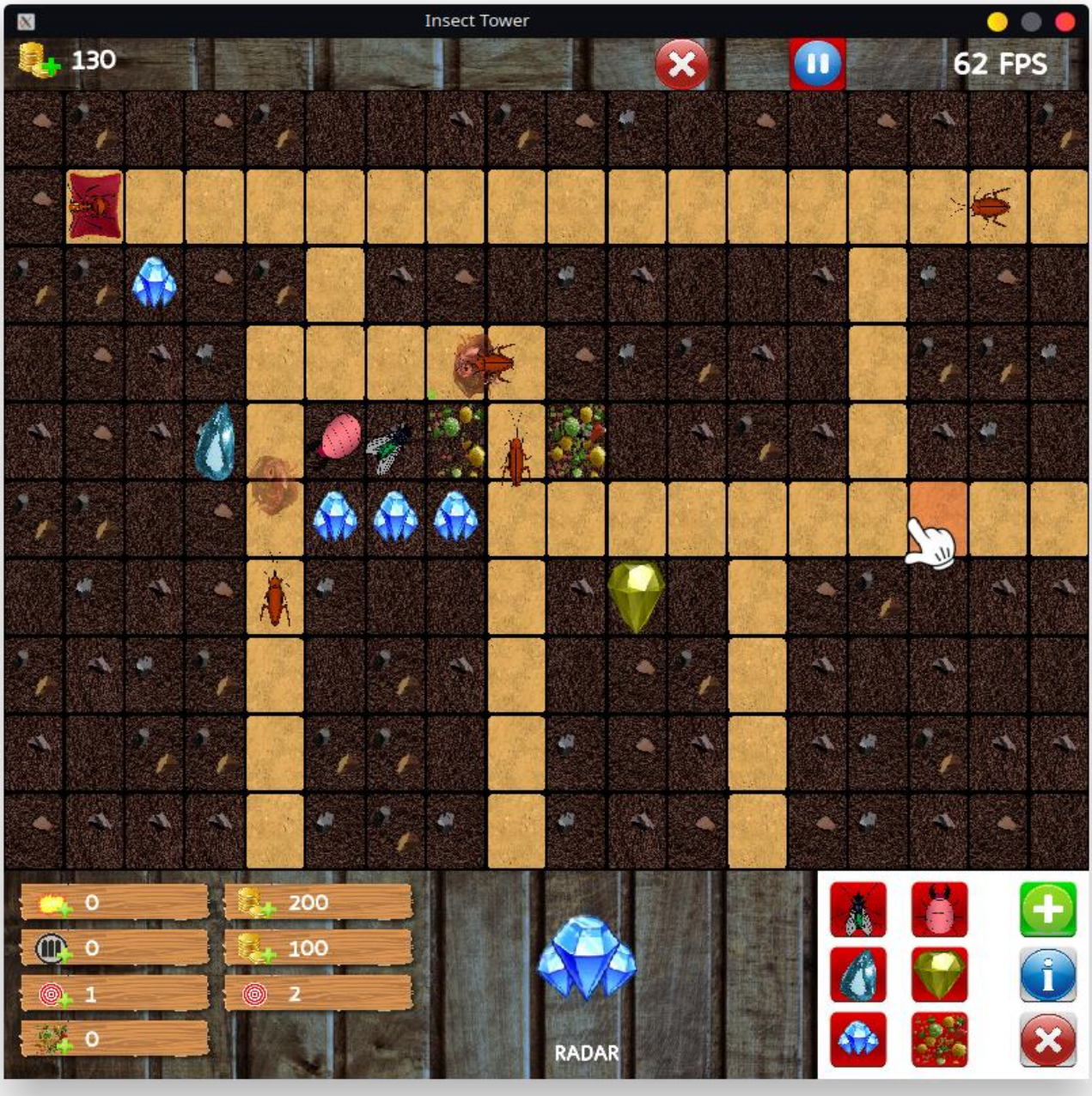


Menu Principal

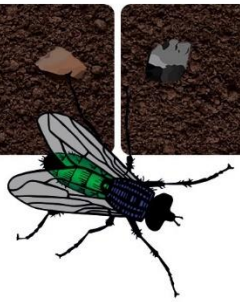


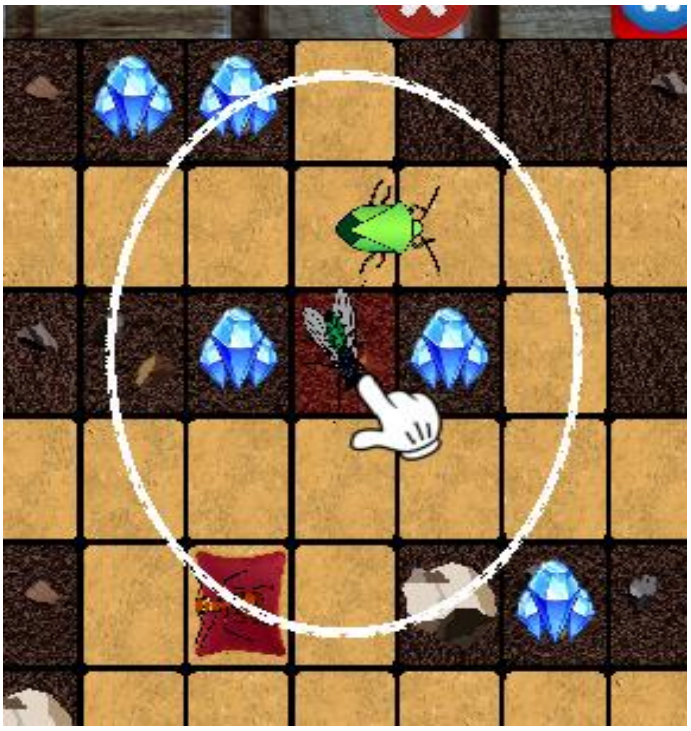
Menu Gagné

Menu Perdu

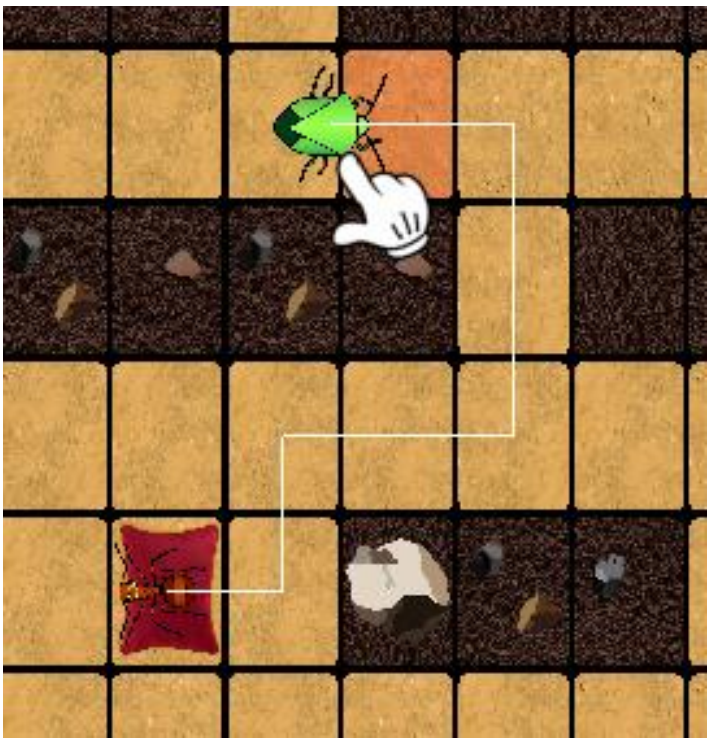


Interface d’un niveau

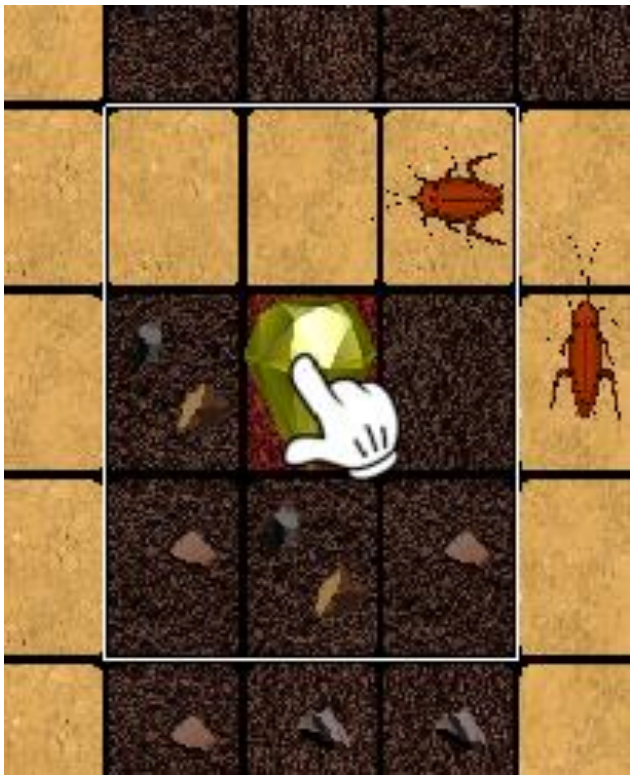




Zone de portée tour

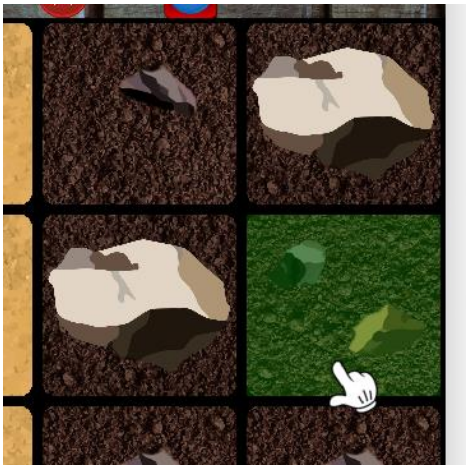


Itinéraire monstre



Zone de portée centrale

Zone non-constructible



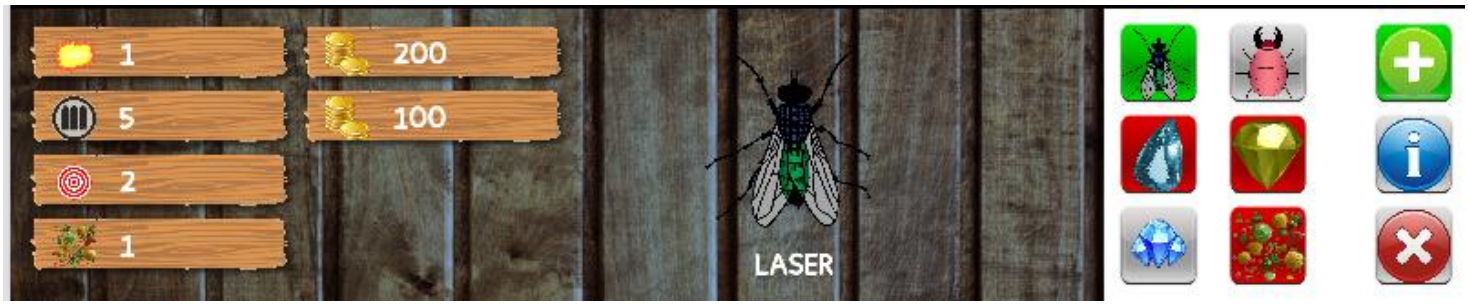
Zone constructible

Différentes textures

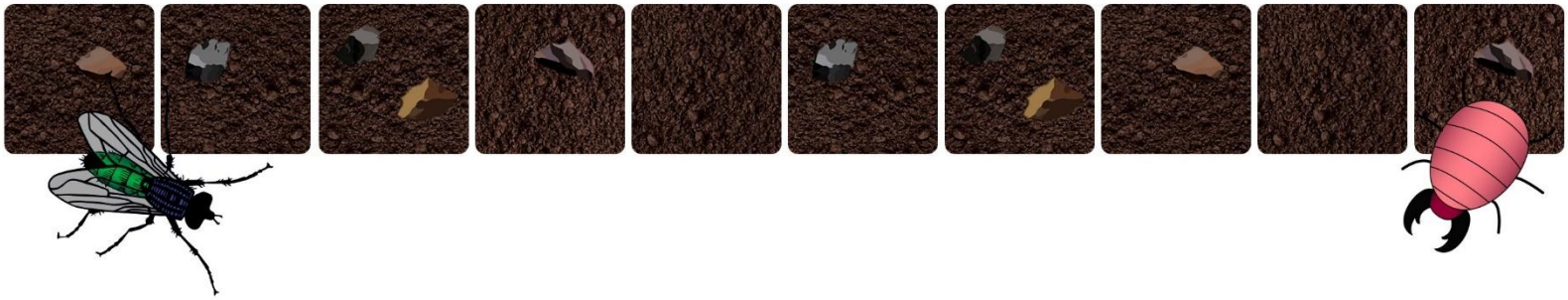
de bâtiments

Stocks de nourritures décroissants

Menu Top



Menu Bottom



Conclusion

Ce projet nous a énormément appris sur la manière d'aborder la création d'un jeu vidéo et plus largement celle d'un programme graphique. Avant ce semestre, nous n'avions pas idée de la manière dont se gère l'affichage des éléments à l'écran, l'interaction avec l'utilisateur, la gestion d'entités au sein du jeu. Algorithmiquement, nous avons découvert les moyens de faire face à de nombreuses problématiques, qu'il s'agisse de faire en sorte qu'un projectile se rapproche de sa cible, qu'un utilisateur passe d'un menu à un niveau, que des monstres empruntent un chemin plutôt qu'un autre. Face aux fonctionnalités à mettre en place, nous aurions pu, à de nombreuses occasions, choisir la première venue, mais nous avons vraiment tenu à nous interroger à chaque étape sur les manières de faire, à explorer les raisons qui peuvent nous pousser à procédé d'une façon plutôt qu'une autre. Nous pensons que c'est ce qui nous a permis d'acquérir des bases solides sur les aspects fondamentaux de la programmation. Nous aurions aimé implémenter encore beaucoup d'autres fonctionnalités dans ce jeu, chaque algorithme découvert nous donnant de nouvelles idées d'implémentation. Mais même si le temps nous a limité dans nos ambitions, nous sommes très fières du travail produit et des connaissances acquises. Ce fut aussi l'occasion pour nous de concevoir une expérience de jeu telle que nous aimons l'imaginer, et que nous espérons que vous apprécierez.