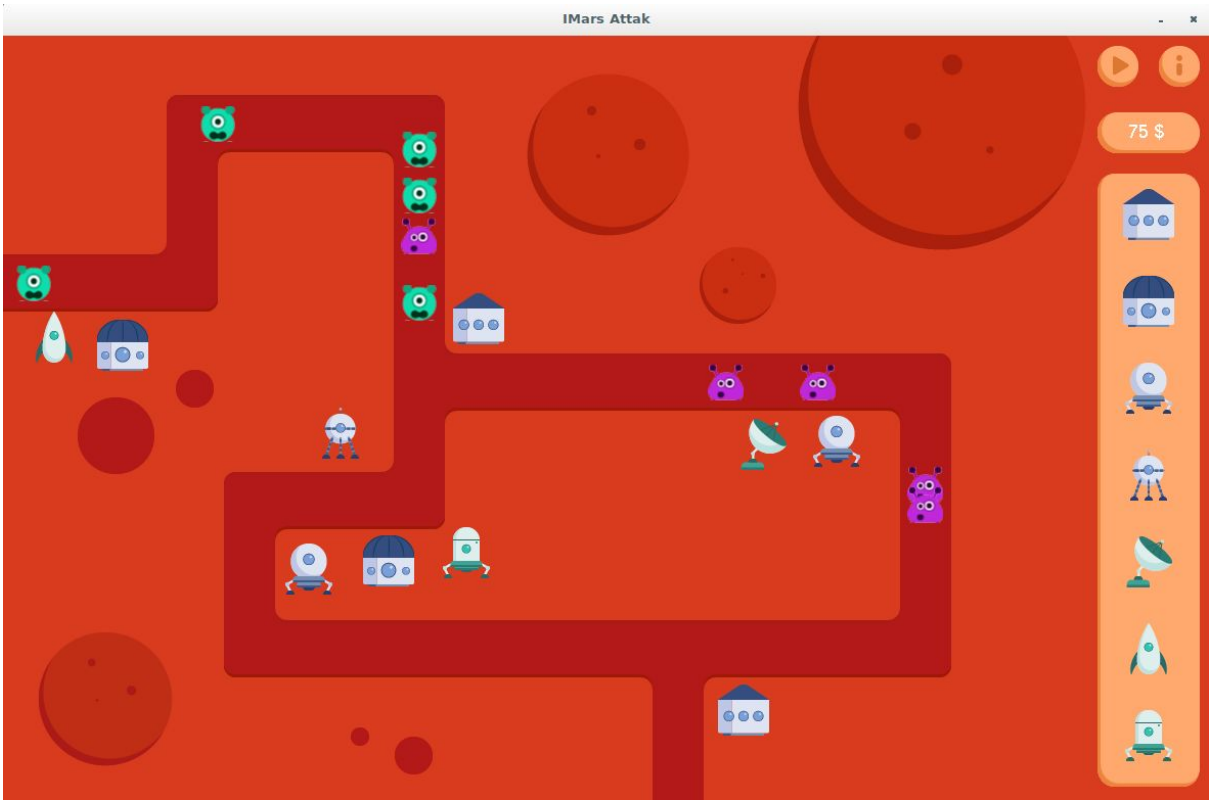
IMAC TOWER DEFENSE

PROJET SYNTHÈSE D'IMAGES / PROGRAMMATION ALGORITHMIQUE



Amandine Kohlmuller
Louisa Chikar

06.06.2019

SOMMAIRE

INTRODUCTION

Dans le cadre des cours de synthèse d'images et de programmation algorithmique du second semestre de la formation IMAC, nous avons réalisé une application SDL, un tower defense, avec le langage C++.

L'objectif était de pouvoir visualiser une carte composée de chemins, des monstres se déplaçant dessus et une interface permettant à l'utilisateur de poser des tours pour détruire les monstres.

Voici le lien du dépôt GIT contenant l'ensemble de notre projet :
https://github.com/Lchikar/Tower_Defense

Le jeu a été réalisé en binôme. Louisa Chikar s'est davantage concentrée sur le développement de la partie algorithmique du projet et Amandine Kohlmuller sur la partie infographie et affichage. En effet, nos prérequis complémentaires nous ont permis de trouver une méthode de travail efficace et adaptée. Cependant, aucune de nous n'avait déjà codé en C++. Ainsi, ce projet nous a permis de mettre en pratique nos connaissances en programmation orientée objet et donc d'apprendre un nouveau langage. Néanmoins, nous verrons que cela nous a également freiné dans l'avancement de certains aspects du projet.

Ainsi, notre jeu : IMARS ATTACK, est actuellement jouable mais nécessite encore des améliorations, notamment dans la gestion du gameplay.

1. DESCRIPTION DE L'APPLICATION

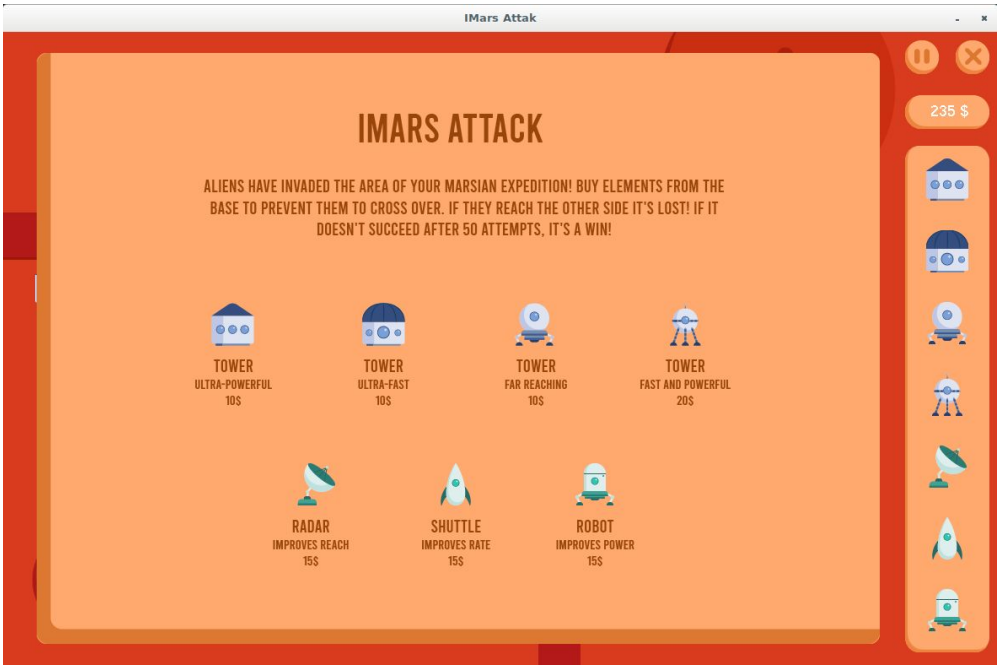
1.1 Fonctionnalités

Au lancement, l'application IMARS ATTACK charge une unique carte et les aliens commencent à arriver par vagues depuis la gauche de la fenêtre. Nous retrouvons deux types d'aliens : les *fatty* qui sont plus lents mais plus résistants et les *nervous* qui sont plus rapides mais moins résistants.

L'utilisateur possède 50$ au début du jeu. Il a la possibilité d'acheter des tours et des bâtiments en cliquant une première fois sur l'installation de son choix dans l'interface à droite de la fenêtre. Il peut poser ses installations avec un second clic à l'emplacement de son choix sur la carte.

Les tours vont tirer sur les aliens, diminuer leurs points de vies jusqu'à les tuer et ainsi rapporter de l'argent au joueur. Les bâtiments améliorent certains paramètres des tours (portée, cadence ou puissance) qui se trouvent dans leur périmètre d'action. Les différences de prix et de pouvoirs des tours et des bâtiment sont consultables en cliquant sur le bouton "i" qui affiche le guide utilisateur. Le bouton pause permet de stopper le jeu, c'est à dire arrêter des aliens dans leur course et empêcher l'achat de nouvelles installations, jusqu'à ce que l'utilisateur clique sur play.

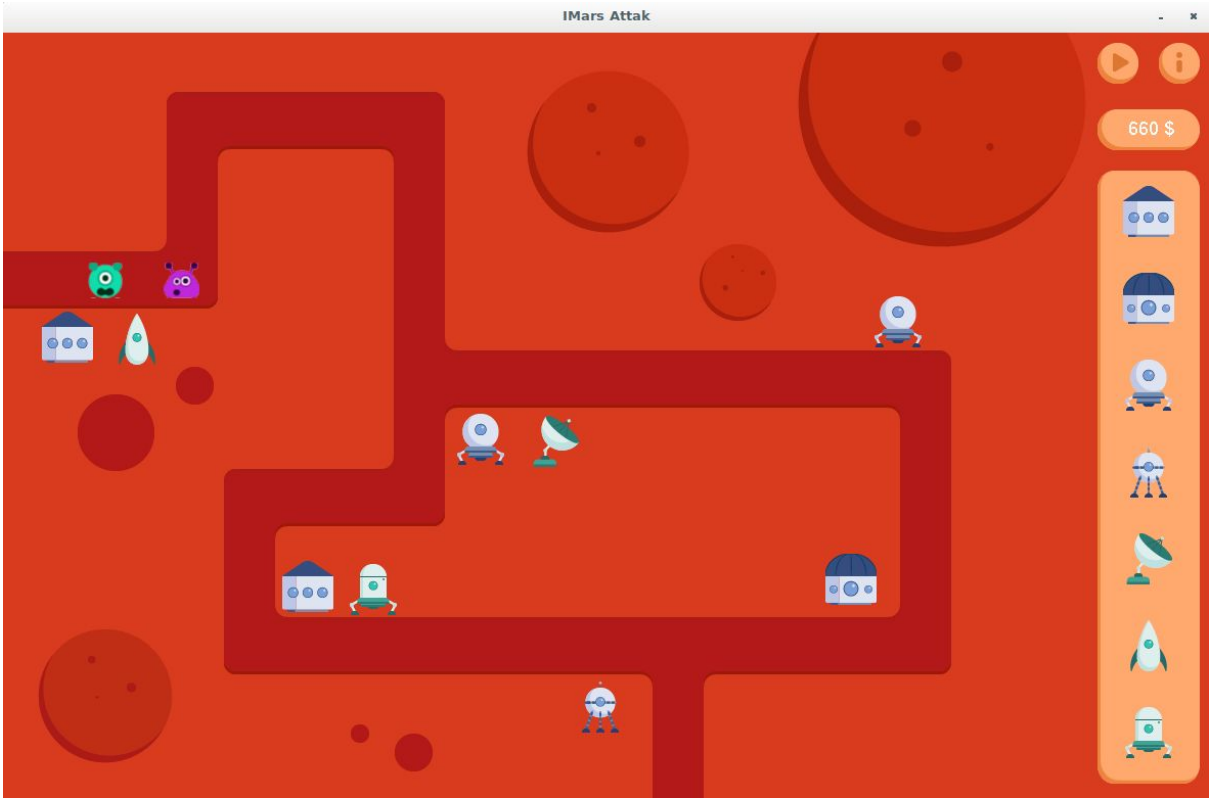
Si un alien arrive dans la zone de sortie, le bout du chemin au bas de la carte, l'utilisateur a perdu et un encart "*game over*" s'affiche avant que l'application se perd. Si au bout de 20 vagues d'aliens, aucun n'a atteint cette zone, l'utilisateur a gagné et, de la même manière, un encart "*winner*" s'affiche.



1.2 Univers

L'avancée technologique et le new space ont permis à l'homme d'installer des bases sur la planète rouge. Mais lorsque ces programmes d'explorations ont démarré, les agences spatiales ignoraient que la vie existait encore sur Mars. Les premières expéditions martiennes se sont déroulées sans encombre. Jusqu'au jour où une équipe chinoise fora un peu trop profondément dans le cratère Korolev et libéra toute une population d'extra-terrestres en colère. La NASA et le CNES ont réagi rapidement en envoyant des missions diplomatiques et scientifiques. Malheureusement aucune ne revint. Vous faites maintenant partie de l'une de ces expéditions. Vous allez devoir vous défendre si vous voulez un jour revoir la Terre.

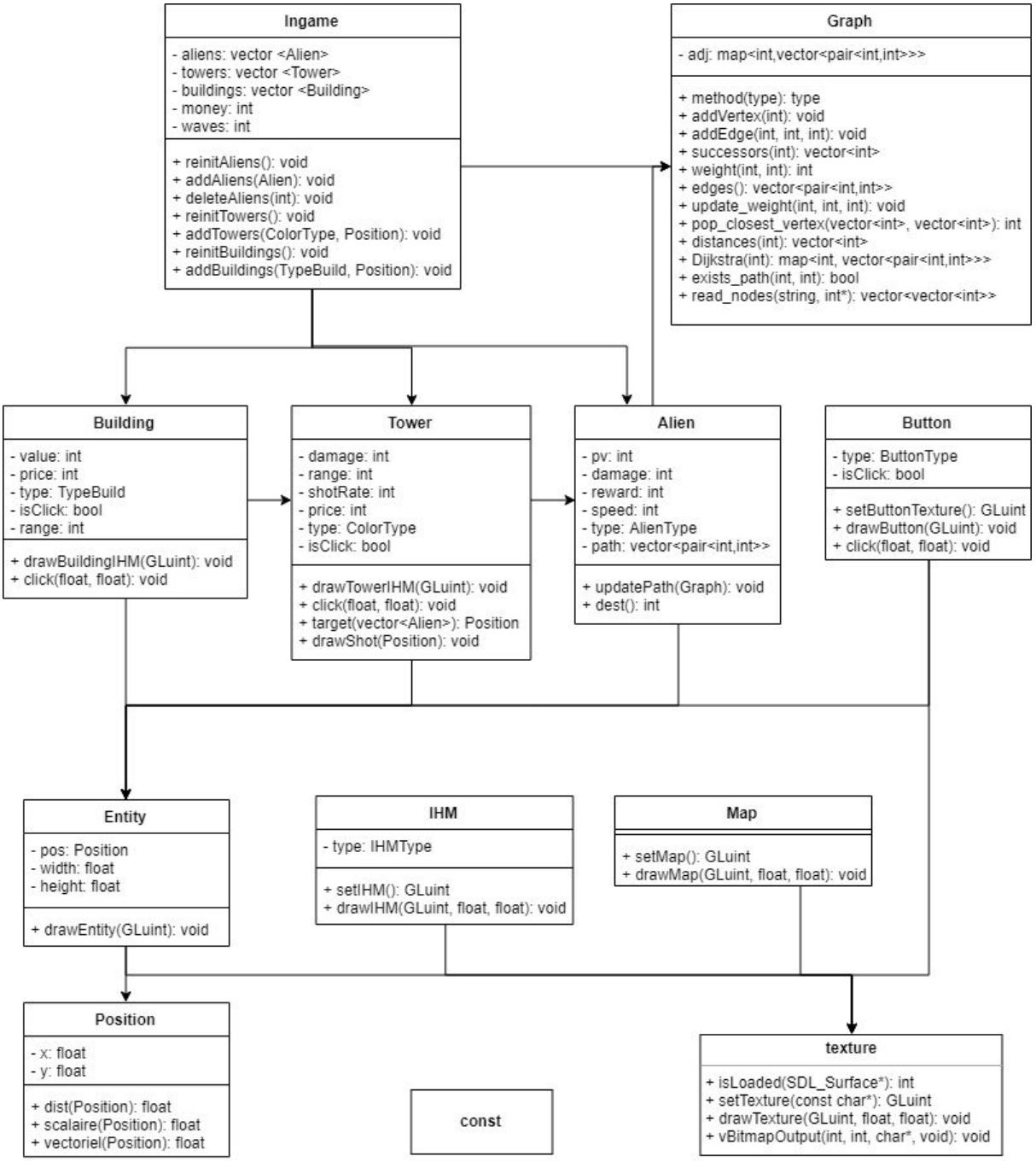
Nous avons réalisé l'entièreté des graphismes du jeu en nous inspirant de style flat design.



2. PRODUCTION

Ce projet était plus propice à l'utilisation de la programmation objet car nous devions implémenter des entités qui intéragissent entre elles. En partant de là, nous avons posé les bases de l'architecture de notre projet en utilisant des schémas UML qui nous ont permis de répartir aisément le travail et surtout de connaître les étapes de développement.

2.1 Architecture logicielle



Nous avons découpé le projet en 5 grandes parties:

- Éléments jouables, regroupés dans la classe *InGame* qui regroupe les éléments nécessaires au jeu, c'est-à-dire
    - la liste des aliens en jeu,
    - celle des tours,
    - celle des bâtiments,
    - l'argent du joueur
    - le nombre de vagues
- Interface graphique
    - IHM
    - Boutons cliquables
    - Textures
- Calculs, avec la classe *Graph* dans laquelle on retrouve les algorithmes pour calculer les chemins des aliens
- Factorisation de code, avec *Entity* qui gère les éléments qui sont positionnables et affichables
- Utilitaires, avec *Position* qui facilite l'implémentation des positions des éléments du jeu.

LES PROBLÈMES

Nous nous sommes rendu compte de plusieurs problèmes au cours du projet.

Par exemple, les tours et les bâtiments ont beaucoup de méthodes en commun. Ils doivent être affichés sur l'interface, nous devons vérifier l'état du clic et ils doivent être posés sur une zone constructible. Le problème avec ces méthodes, c'est qu'elles ne concernent que les tours et les bâtiments, à l'exception du clic qui se fait aussi sur les boutons, elles ne peuvent donc pas être placées dans *Entity*. Ainsi, lorsque nous nous sommes retrouvées face à ce problème, nous avons dû faire un choix entre revoir une partie de l'architecture et donc perdre du temps pour implémenter d'autres fonctionnalités, ou s'adapter. N'ayant vraiment plus le temps de tout revoir, nous avons dupliqué certaines méthodes et factorisé d'autres dans des fichiers à part.

Un autre problème concerne la class *Map*. Elle devait dans un premier temps accueillir les méthodes se trouvant dans *Graph*, mais finalement nous avons décidé de séparer la partie algorithmique de la partie affichage de la carte. Ainsi, *Map* ne possède pas de méthodes différentes de la class *IHM* et cette dernière aurait donc pu se charger de l'affichage de la carte.

Par ailleurs, vous remarquerez aisément le nombre trop important de fonctions *draw* se trouvant dans les différentes classes. Ici aussi nous avons rencontré des problèmes, que nous évoquerons par la suite dans la partie infographie et affichage de ce rapport. Par conséquent, il aurait peut-être été préférable de faire une classe *Draw* contenant toutes ces méthodes.

2.2 Structures de données

DÉROULEMENT DU JEU

```
Test des arguments de l'exécutable
Initialisation du graphe
Initialisation du jeu

Fenêtre SDL
Initialisation des textures

Boucle principale:
    Test fin de vague
    Test victoire

    Recalcul des risques par arcs
    Recalcul des chemins des aliens

    Affichage:
        - map
        - interfaces utilisateur
        - éléments jouables

    Boucle d'événements:
        Test sortie du jeu
        Test clic pour :
            - afficher les informations
            - mettre en pause le jeu
            - achat d'une tour ou d'un bâtiment

    Déplacement des aliens

    Attaque des tours:
        Tir sur la cible la plus proche
        Test mort alien

    Amélioration des tours par les bâtiments

Libértion des données GPU
Libération des données associées à la SDL
```

2.2.1 Algorithmique

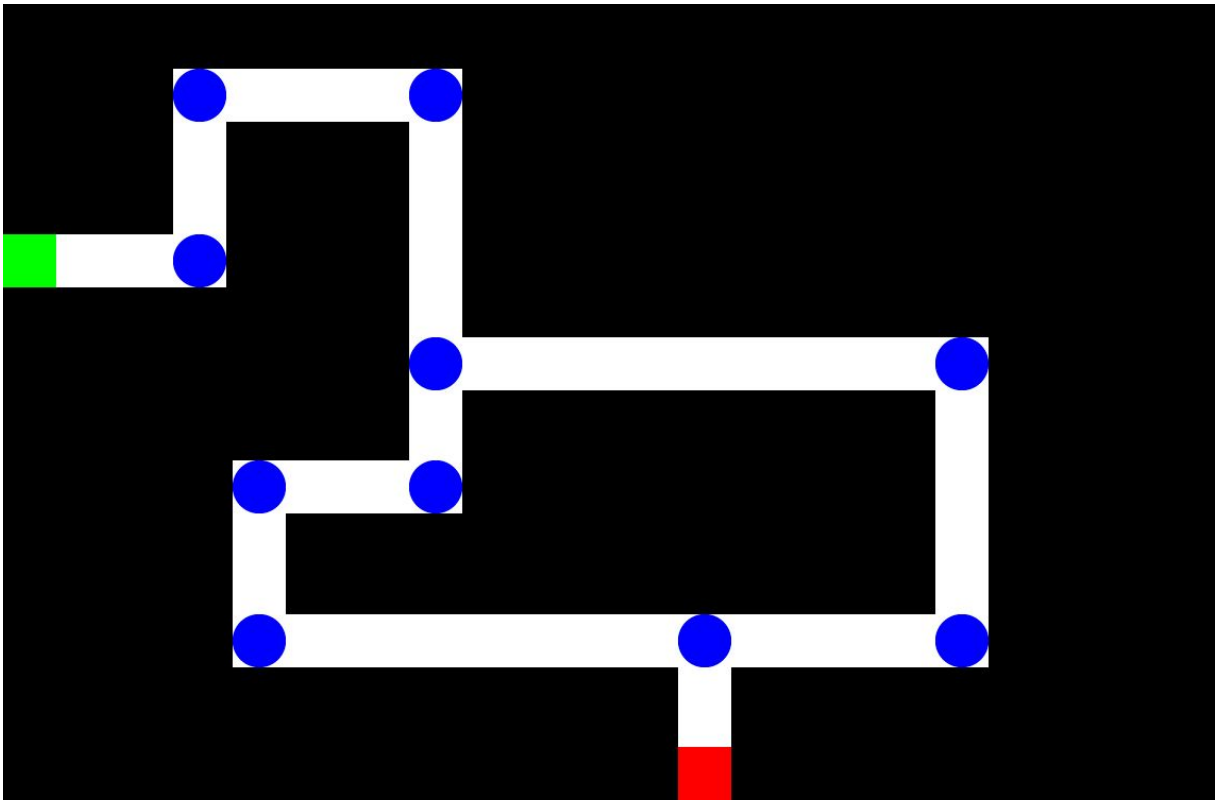
STRUCTURE

Nous avons finalement décidé de ne pas utiliser le fichier *PPM* pour la partie algorithmique. En effet, par manque de temps, nous avons dû faire des choix. Nous sommes donc partie du principe que comme la carte est déjà dans le programme et pas donnée par l'utilisateur, nous n'avions pas besoin du fichier *PPM* ni de faire les vérifications sur les codes couleurs de la carte. Toutes les informations du jeu sont dans le fichier *ITD* et sont récupérées pour être injectées dans la classe *Graph*.

Inconvénient : le calcul des zones constructibles aurait été plus facile avec le fichier *PPM.* À la place, nous vérifions si la position de la tour ou du bâtiment à ajouter est sur un arc du graphe. Pour ça, nous calculons comme ceci:

$$\vec{ut} \wedge \vec{tv} = 0$$
$$\& \; 0 < \vec{uv} \cdot \vec{ue} < \vec{uv}^2$$
$$alors \; t \; est \; sur \; l'arc \; (u, v)$$

Nous avions quand même créé l'image, qui nous a surtout permis d'avoir les coordonnées des noeuds pour le fichier .itd et de créer une texture martienne calquée sur ces chemins.



Pour la représentation du graph, nous avons utilisé un dictionnaire d'adjacence avec comme clés les noeuds du graphe et comme valeur un tableau de couple de voisin et du poids de leur arête. De plus, nous avons choisi de considérer le graphe comme étant orienté.

Nous avons choisi d'utiliser des tableaux de type *vector* pour leur facilité d'utilisation dynamique.

ALGORITHMES

Voici les algorithmes principaux implémentés en pseudo-code:

LIRE / INTERPRÉTER FICHIER ITD

```
READ_NODES:
in: chemin du fichier itd
out: tableau informations, nombre de noeuds
    Pour chaque ligne du fichier:
        Si ligne identifiée est incorrecte:
            Affichage erreur
            Renvoi tableau vide

        Si ligne contient informations d'un noeud:
            Ajout des informations du noeud dans le tableau:
                - Type de noeud
                - Abscisse
                - Ordonnée
                - Successeurs

    Renvoi du tableau rempli
```

CRÉATION DES CHEMINS DU GRAPHE

```
SOMMET LE PLUS PROCHE:
in: liste des sommets, tableau de distances
out: indice du sommet le plus proche

    sommetResultat <- -1
    distMin <- MAX

    Pour chaque candidat parmi les sommets:
        Si distance[candidat] < distMin:
            distMin <- distance[candidat]
            sommetResultat <- candidat

    Si -1 != sommetResultat:
        extraction de sommetResultat dans sommets

    Renvoi de sommetResultat
```

```
DIJKSTRA:
in: Graphe, source
out: dictionnaire des chemins entre source et clés

    aTraiter <- file d'attente contenant tous les noeuds du graph
    dist <- tableau de distance entre indice et source initialisé au max
    chemins <- dictionnaire des chemins vide

    dist[src] <- 0

    Tant que aTraiter:
        noeud u <- sommet le plus proche de src

        Si -1 = u:
            Renvoi de chemins

        Pour tout v successeur de u:
            distActuelle <- dist[u] + coutArc(u,v)

            Si distActuelle < dist [v]:
                dist[v] <- distActuelle
                chemin[v].push((u,v))

    Renvoi de chemins
```

MISE À JOUR DES CHEMINS DES ALIENS

```
UPDATE CHEMINS ALIENS:
    Pour chaque alien en jeu:
        Si destination atteinte:
            Si destination = fin du chemin:
                pop alien
                jeu perdu
            Sinon :
                pop alien.chemin[0]

```

DÉPLACEMENT DES ALIENS

```
DEPLACEMENT DES ALIENS:
    Pour chaque alien en jeu:
        Si position relative à la destination est:
            - Nord -> alien va au Sud
            - Sud -> alien va au Nord
            - Est -> alien va à l'Ouest
            - Ouest -> alien va à l'Est

```

TIRS DES TOURS

```
TIRS DES TOURS:
    Pour chaque tour en jeu:
        cible = alien le plus proche dans le périmètre d'action de la tour
        cible.vie -= tour.degat

        Si cible meurt:
            argent += cible.recompense
            pop aliens[cible]
```

AMÉLIORATIONS DES BÂTIMENTS

```
UPGRADE DES BATIMENTS:
    Pour chaque bâtiment en jeu:
        Pour chaque tour dans périmètre du bâtiment:
            amélioration tour selon type bâtiment
```

2.2.2 Infographie / Affichage

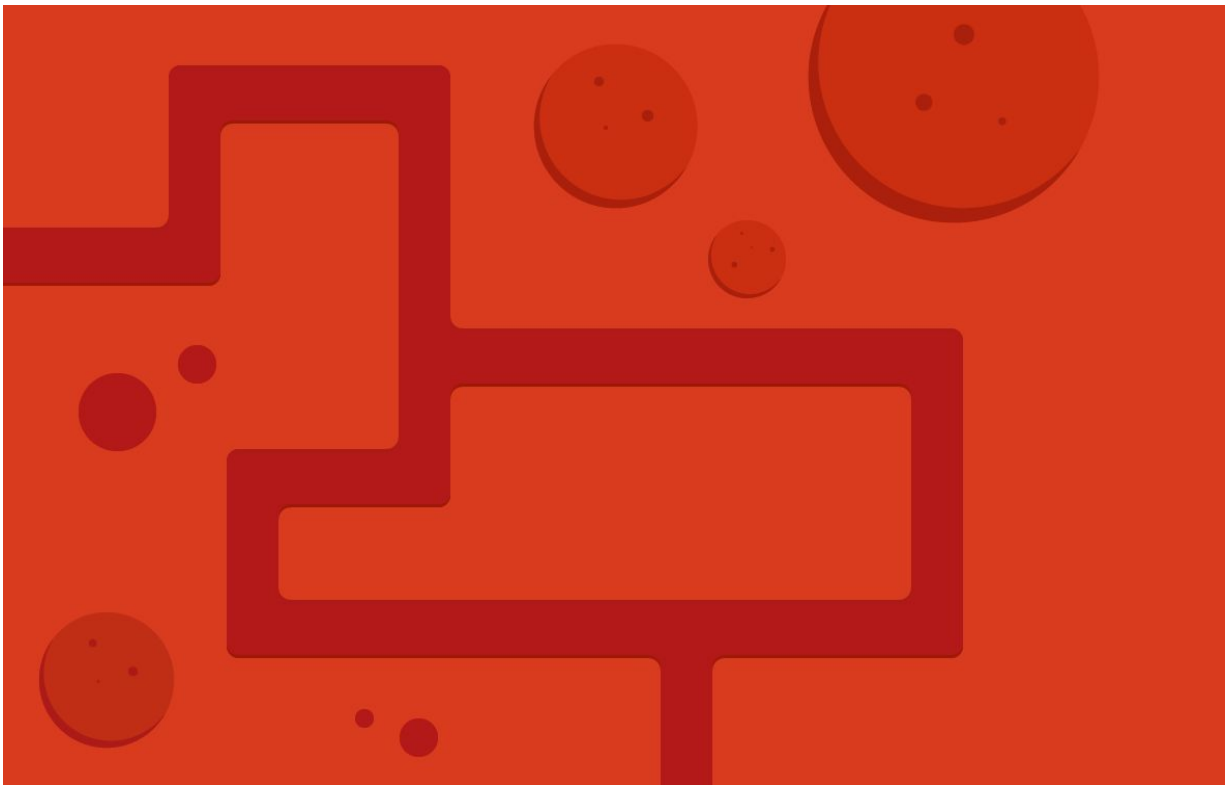
Concernant les sprites de monstres, tours et bâtiments, nous savions que nous n'aurions pas le temps de gérer des animations. Nous avons donc privilégié les autres éléments du projet et donc décidé de fixer toutes les tailles des textures et de faire un *asset* graphique pour chaque élément du jeux.

Nous avons rencontré beaucoup de difficultés pour l'initialisation et le dessin des textures. Dans un premier temps, nous pensions ajouter un attribut GLuint à la classe *Entity*, sachant que chaque entité est représentée par une texture. Puis nous avons compris que nous ne pouvions pas initialiser la texture dans le constructeur car cela signifiait initialiser une texture à chaque instanciation. Or, une texture peut être initialisée qu'une seule fois au début du *main*. Nous avons donc pensé à faire des variables globales pour contenir l'*id* de chaque texture. Cependant, cela ne fonctionnait pas car GLuint a besoin que l'environnement SDL soit créé pour fonctionner.

Ainsi, la solution que nous avons trouvé, pour nous faire perdre le moins de temps possible, est d'initialiser les textures des aliens, bâtiments et tours au début du *main*. En effet, ce sont les objets qui sont présents dans les tableaux dynamiques de la classe *InGame* et qui vont donc être créés et supprimés au cours du jeu. Ces éléments sont donc affichés par la méthode *drawEntity(GLuint)*. Le grand inconvénient est que nous devons indiquer l'*id* de la texture à l'objet. Pour les éléments de l'IHM et la carte, nous avons gardé le code que nous avions déjà mis en place, c'est à dire initialiser et dessiner avec des méthodes de leur classe, pour ne pas perdre de temps et pour alléger le *main* en évitant une plus grande liste de textures.

Avec le recul sur le projet, nous pensons que nous aurions pu essayer une autre architecture avec, par exemple, une classe entièrement consacrée au dessin, au lieu d'appeler des fonctions factorisées dans le fichier *texture.cpp* et de les adapter à chaque type d'objets.

MAP - 1180 x 750 pixels
La texture de la carte fait la taille de la fenêtre et est basée sur le *PPM* qui nous a permis de déterminer les coordonnées des noeuds dans le fichier *ITD*. Les chemins font 60 pixels de largeur.



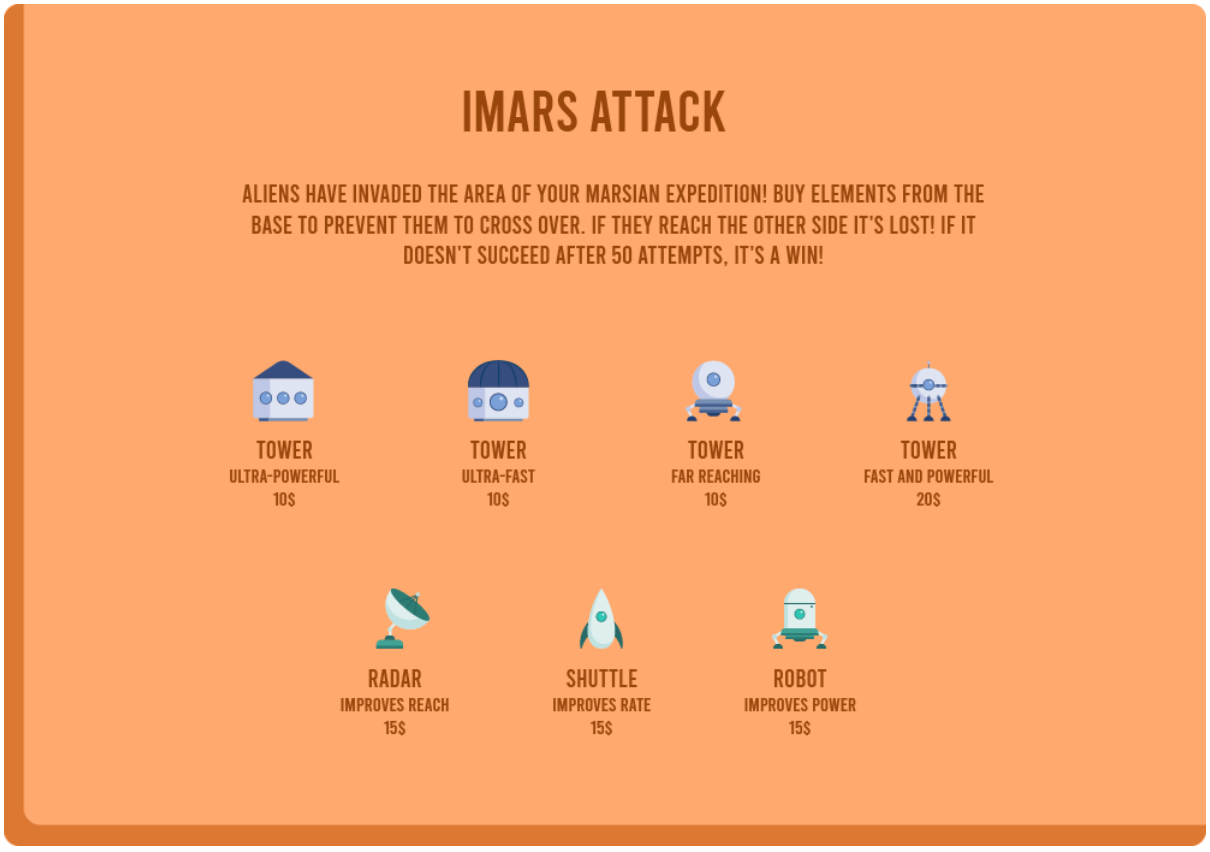
ENTITÉS
Les entités sont des carrés pour simplifier le calcul de la distance en prenant en compte un rayon faisant la moitié de la dimension d'un côté.

Aliens - 35 x 35 pixels
Plus petits que les chemins.

Tours et bâtiments - 50 x 50 pixels
Plus petits que les chemins mais plus grands que les aliens. Graphiquement, nous distinguons les deux types d'installations par la couleur. Les tours sont bleues et les bâtiments verts.

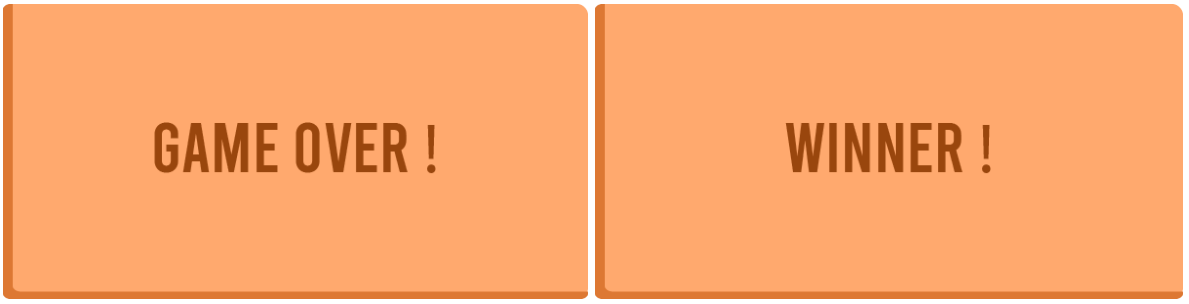
Boutons - 40 x 40 pixels
Les boutons, même s'ils constituent l'IHM, sont des entités. En effet, ils sont cliquables et nous avons donc besoin de connaître leur position. Cependant, nous ne créons qu'un seul objet de chaque type de boutons. Nous n'avons donc pas besoin d'initialiser la texture et de les dessiner comme les autres entités dans le main. Ainsi, ils ont leur propre fonction *draw* qui appelle *drawEntity*…

IHM
Guide utilisateur - 1000 x 700 pixels



Encart argent et encart tours
Support pour afficher l'argent restant au joueur.
Support pour afficher les tours.

Encarts win et loose - 600 x 300 pixels



3. BILAN

3.1 Synthèse récapitulative

LIVRABLES	FAIT	PAS FAIT	FONCTIONNE	FONCTIONNE PAS
Structure ordonnée / système de compilation	X		X	
Création d'une carte (.itd, .ppm et texture)	X		X	
Vérification du .itd	X		X	
Lecture du .itd	X		X	
Lecture du .ppm		X		
Affichage de la carte	X		X	
Création du graphe des chemins	X		X	
Création de 2 types de monstres	X		X	
Création de 4 types de tours	X		X	
Création de 3 types de bâtiments	X		X	
Sprites et animations		X		
Affichage de bâtiments, tours, monstres	X		X	
Acheter et placer des installations	X		X	
Vérifier les zones constructibles	X			X
Tir des tours (algorithmique)	X		X	
Tir des tours (affichage)	X		X	
Actions des bâtiments	X		X	
Déplacement des monstres	X		X	
Calcul du risque du chemin	X			X
Gestion des vagues de monstres	X		X	
Affichage du nombre de vague		X		
Gestion de l'argent	X		X	
Affichage de l'argent	X		X	
Affichage d'un guide utilisateur	X		X	

Bouton pause	X		X	
Affichage feedback win et loose	X		X	
Gestion du temps	X		X	

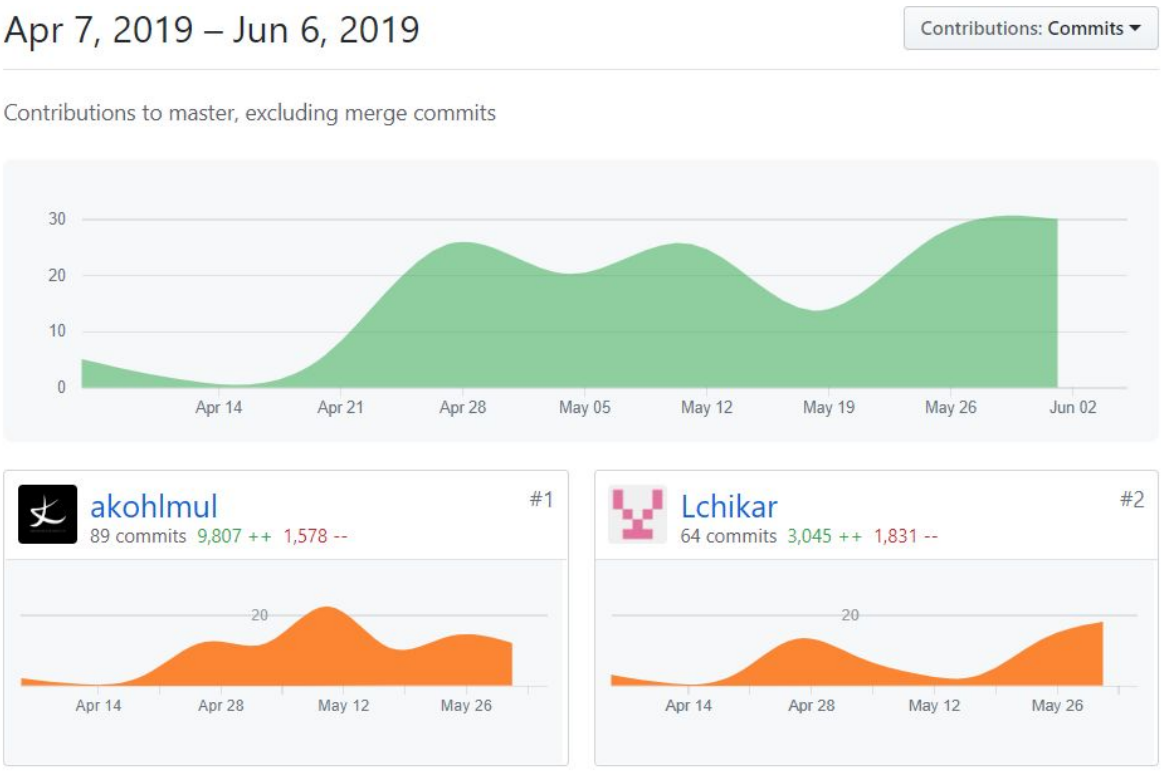
3.2 Difficultés

La plus grosse difficulté à laquelle nous avons été confronté pour ce projet concernait le temps. Dès le début, nous savions que nous allions devoir faire des choix dans la structure du projet quitte à sacrifier quelques aspects du jeu.

Nous avons également passé beaucoup de temps à faire des recherches sur la documentation du C++, d'OpenGL et de la SDL. Par exemple, les calculs des positions entre les différents repères ont été revus plusieurs fois.

En revanche, ces difficultés nous ont permis d'acquérir une importante quantité de nouvelles connaissances. Nous avons beaucoup appris et nous avons aimé travailler sur ce projet en cherchant toujours des solutions pour contrer les problèmes. De plus, nous avons pu prendre du recul sur la structure de notre application et repérer les éléments qui peuvent encore être améliorés.

Ci-dessous une capture d'écran de la synthèse des commits sur le master depuis le début de la phase de production du jeu.



CONCLUSION

Ainsi, nous vous présentons la première version du jeu IMARS ATTACK. Il est actuellement jouable, il respecte la plupart des contraintes du cahier des charges et demande encore à être amélioré. Ce projet nous a beaucoup apporté tant dans l'aspect compétences techniques que dans l'aspect gestion de projet (qualité/délai).