Rapport SpaceCraft

Thomas Delépine, Alexis Delplace, Jacobo Ruiz-Ocampo

2021/2022

1 Introduction

L'objectif de ce projet est de réaliser un jeu de stratégie en temps réelle. Le jeu nommé *SpaceCraft* est un jeu de gestion spatiale. Plus précisément, c'est un city-builder de micro-gestion d'une base spatiale.

Le jeu est fortement inspiré des jeux de stratégies développés durant les années 90 tel que *Dune 2000* (1998) ou *SimCity* (1989).

Le jeu consiste à établir une colonie puis à la développer afin qu'elle survive malgré des conditions telles que des chutes de météorites, des tempêtes, etc. Le joueur devra prévoir les carences de stocks en produisant suffisamment d'oxygène, de nourriture pour la survie des colons et de pièces détachées pour la maintenance.

FIGURE 1 – Capture d'écran du jeu *Dune 2000*

Table des matières

2 Analyse Globale

- Création et initialisation du terrain de jeu
  - Création du terrain de jeu
  - Génération des montagnes de manière aléatoire
  - Apparition aléatoires de ressources sur la carte (roches contenant différents minerais)
  - Atterrissage du vaisseau mère avec les premiers astronautes
- Création de la fenêtre dans laquelle est affichée la représentation du terrain ainsi que le panneau de contrôle
- Affichage du terrain de jeu
  - Création du *JPanel* représentant l'affichage du terrain de jeu
  - Affichage statique (affichage d'un état du jeu à un instant t)
  - Affichage dynamique (affichage prenant en compte la mise à jour du modèle)
  - Zoom de la caméra sur l'affichage du terrain via la mollette de la souris
  - Déplacement de la caméra à travers le terrain via la souris (clic and drag)
- Affichage du panneau de contrôle
  - Création du *JPanel* représentant l'affichage panneau de contrôle
  - Mécanisme permettant à l'utilisateur de sélectionné une entité en cliquant dessus
  - Affichage de la miniature correspondant à l'entité sélectionné
  - Affichage des boutons permettant aux entités de faire des actions en fonction des actions qu'ils peuvent effectuer
- Personnages contrôlés par le joueur : les Space Marines
  - Apparition des Space Marines sur la carte
  - Déplacement de leur position vers un point sélectionné sur la carte via le panneau de contrôle
  - Récolte des ressources à proximité du Space Marine (1 case) via le panneau de contrôle
  - Attaque des aliens à proximité du Space Marine (1 case) via le panneau de contrôle
- Personnage non-joueur : les aliens
  - Déplacement aléatoire sur la carte en continu
  - Attaque les ressources à proximité
- Les bâtiments
  - Construction d'un bâtiment via le panneau de contrôle en cliquant sur une case vide
  - Destruction d'un bâtiment via le panneau de contrôle en cliquant dessus
  - Génération de ressources, certains bâtiments tel les mines génère de l'or de manière continu et automatique
- Le vaisseau
  - Apparition aléatoire de vaisseau spatial sur la carte à un endroit ayant suffisamment de place pour l'accueillir
- Les ressources : les météorites
  - Génération (initiale) aléatoire sur la carte des météorites
  - Récolte des météorites par les Space Marines
  - Destruction par les aliens
  - Apparition aléatoire sur la carte au fur et à mesure que la partie avance
- Un score pour classer les performances du joueur
  - Affichage d'un score des roches du joueur
  - Affichage d'un timer
  - Création et affichage à la fin du jeu d'un leaderboard
- Le launcher
  - Création de la fenêtre
  - Création du panel permettant de modifier les paramètres du jeu
  - liaison entre les données entrer dans le launcher et les données du jeu
  - Création du bouton *Play*

3 Plan de développement

Tout au long du projet, on implémente et améliore différentes options et différentes mécaniques pour le jeu. Cela se fait avec des tests, des périodes de débogage et des périodes de réflexions. Finalement, on a constamment travaillé sur tous les aspects du jeu. Pour l'organisation, on a beaucoup communiqué en vrai et sur discord. On a d'ailleurs créé un serveur pour le suivi de ce projet, sur lequel on note les bugs, les choses à coder, et sur lequel on organise l'évolution du projet. Finalement, les étapes de notre travail ressemblent à ça sur toute la durée du projet :

	Semaine 1	Semaine 2	Semaine 3	Semaine 4	Semaine 5	Semaine 6	Semaine 7	Semaine 8	Semaine 9	Semaine 10
Génie logiciel	Github	Restrucutration	Factorisation		Rapport & Version stable			Paramétrisation du jeu	Rapport & Tests	Rapport & Restrucutration
Modèle	V1	Rework	V2		Météorite					
Board Panel	V1 statique	V2 dynamique				V3 images		Montagne images		
Fenêtre	V1							Launcher LeaderBoard V0		LeaderBoard V1
Terrain	Generation V1	Seed		Spawn vaisseau et space marines		Spawn Météorites				
Caméra	V1 Zoom	Déplacement		Clic Zoom		V2				
Control Panel	V0		Modulaire + Bouttons				V1	Lore		
Algorithme			hitbox V1 + A* V1	Opti A* + Mouvements			Alien destruction ressources			
Gameplay				Peur Alien			Minage + Score V1	Timer + Score V2 + Combat		

Figure 2 – Organisation du travail

4 Conception générale

On a adapté le modèle *Modèle-Vue-Contrôleur* pour qu'il aille avec nos besoins. La structure est détaillée dans la partie 8 **Documentation Développeur**.

Pour comprendre la conception générale, il peut être intéressant de regarder le flux des données de notre jeu :

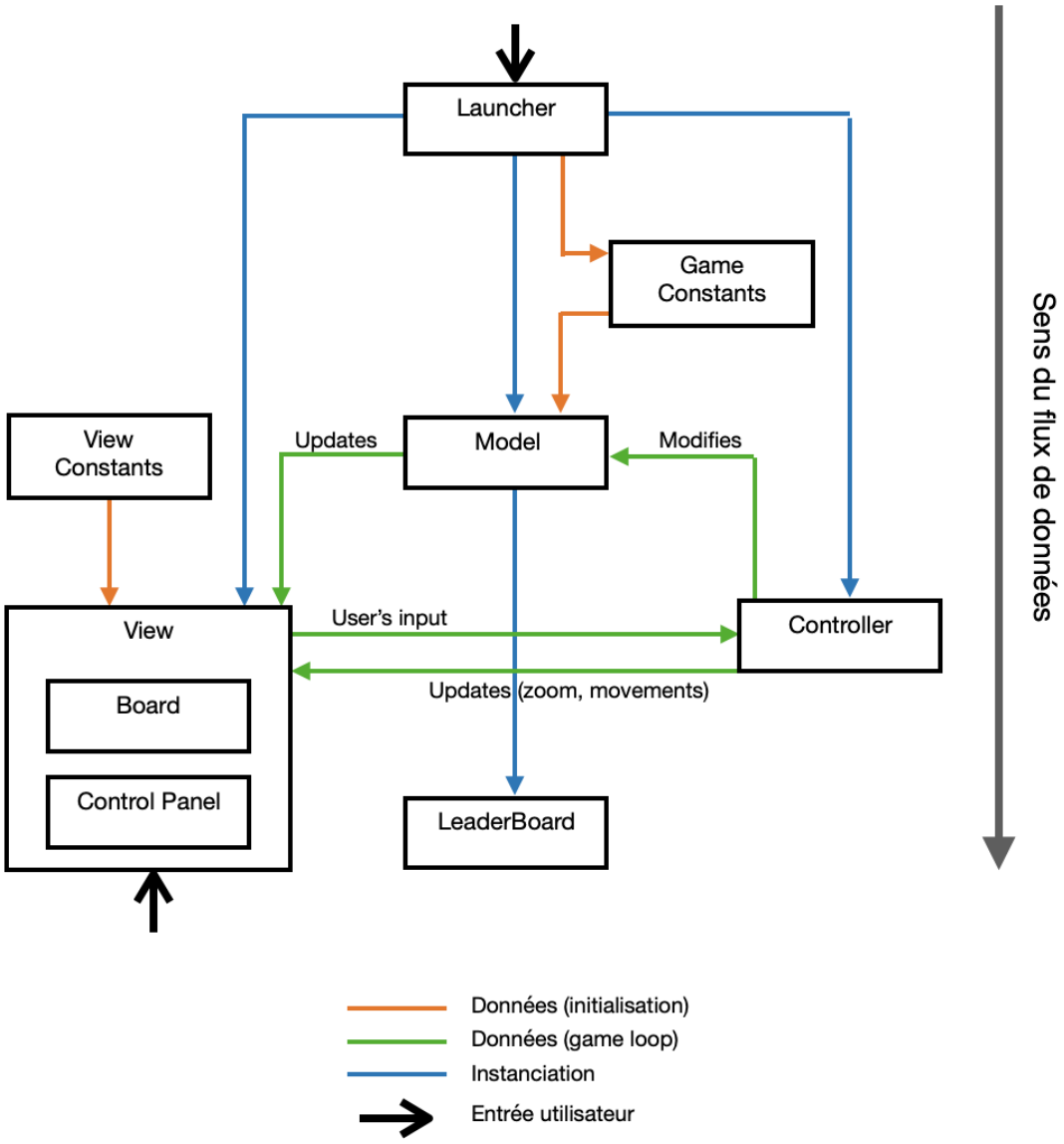


FIGURE 3 – Schéma représentant le flux de données

On y voit que l'utilisateur paramétrise le jeu depuis le launcher. Le launcher modifie GameConstants qui est l'une des classes centrales du projet, et GameConstants influe sur la création du modèle. Puis le jeu se déroule, avec un modèle, un contrôleur et une vue. Ainsi, la structure de notre projet suit le modèle *MVC*, à la subtilité près que la vue modifie le modèle au début du jeu.

5 Conception detaillée

5.1 Création et initialisation du terrain de jeu

5.1.1 Création du terrain de jeu

Le terrain du jeu est représenté par la classe `GameBoard`. Plus précisément, le terrain est représenté par 3 `ArrayList` présentent dans la classe `GameBoard`. Ces trois listes sont `mountains`, `structures`, `entities`. La liste `mountains` est initialisée via la classe `RandomLandGeneration` présentée dans les parties suivantes. Les listes, `structures` et `entities` sont quant à elles initialisées par la fonction `initLand()` présentée dans la partie *5.1.3*.

Le terrain de jeu est créé et initialisé assez logiquement en instanciant un nouvel objet de type `GameBoard`.

5.1.2 Génération aléatoire des montagnes

On souhaite générer aléatoirement un terrain. Celui-ci doit avoir plusieurs propriétés :
— les dimensions sont fixées
— il y a un nombre fixé de chaînes de montagnes
— leur forme est aléatoire
— les montagnes n'isolent pas de parties du plateau.

Pour ce faire, on génère dans un premier temps les chaînes de montagnes. Puis, on peut voir le plateau de jeu comme un graphe. On souhaitera donc modifier les montagnes générées pour rendre le graphe connexe. On se dote de constantes nbMontagnes qui désignent le nombre de chaînes de montagnes qu'on souhaite générer et pourcent, qui désigne la proportion du plateau qu'on souhaite que les montagnes recouvrent. On ajoute des attributs borneMax et borneMin servant aux tirages aléatoires.

Pour la génération de montagnes, on a créé un type énuméré Couleur, contenant les couleurs "montagne", "vue", et "pas-vue". L'idée étant de modéliser les montagnes dans un tableau de couleur, plus agréable à manipuler, puis de recopier ce tableau de couleur vers un tableau d'objets Montagne, le modèle du terrain. On modélise donc dans un premier temps le plateau par un tableau à deux dimensions de couleurs : board. On commence alors à remplir board de montagnes. L'algorithme de génération de montagnes est le suivant :

```
1 Board est initialisé à un tableau de couleur pas-vue ;
2 Le contour de Board est coloré par la couleur montagne ;
3 for i ← 0 to nbMontagnes do
4 |   x, y ← valeurs aléatoire pointant une case pas-vue;
5 |   generateMountain(x, y, Board);
6 end
```

et le code de generateMountain(int, int, tableau[couleur][couleur]) :

```
1 Board[i][j] ← montagne;
2 Tirage ← nombre aléatoire entre 0 et borneMax;
3 if Tirage ≤ borneMin then
4 |   x, y ← coordonnée d'un des voisins de la case courante;
5 |   generateMountain(x, y, Board);
6 end
```

De cette façon, on ne contrôle pas le nombre de cases colorées en montagne. la génération est totalement aléatoire. On peut noter que pour les tirages, tant qu'on a un succès, on continue et sinon on s'arrête. Cela se modélise par une loi géométrique. En notant α le pourcentage de terrain qu'on veut recouvrir, β le nombre de chaînes de montagnes qu'on souhaite générer, et M, N les dimensions du plateau, on peut facilement calculer le paramètre p de la loi géométrique

pour avoir en moyenne le bon recouvrement. En effet, l'espérance d'une loi géométrique est simplement :

$$E[X] = \frac{1}{p}$$

À noter qu'ici, on a une répétition de succès, puis un échec alors que dans la loi géométrique, on a une répétition d'échecs puis un succès. On va alors vouloir calculer non pas p, mais 1-p, 1-p est alors la probabilité de succès de notre loi géométrique. De cela, il vient :

$$\beta \times \frac{1}{1-p} = \frac{\alpha}{100} \times M \times N$$

$$\Leftrightarrow p = \frac{\alpha \times M \times N - 100\beta}{\alpha \times M \times N}$$

Le résultat sous forme de fraction est parfait, car il ne reste plus qu'à associer le dénominateur à borneMax, et le numérateur à borneMin. Cela permet donc en théorie une génération suivant les paramètres désirés. À noter que le cas où on souhaite générer une montagne autour d'une case, mais que toutes les voisines de la case sont déjà des montagnes est possible. On pourrait simplement arrêter la génération à ce moment-là, mais cela biaiserait la génération. On serait en moyenne bien en dessous de l'espérance calculée plus tôt. Ce cas est donc traité : on cherche une case non colorée proche, et on relance la génération depuis ce point. La recherche de cette case suit un parcours pseudo-aléatoire pour limiter au plus les biais dans la génération du terrain.

On veut ensuite rendre le graphe formé par le plateau connexe. Pour cela, on suit l'algorithme suivant :

```
1  flag ← true;
2  Sélectionner puis Colorer une case non-vue en vue;
3  Colorer récursivement les voisins de cette case par un DFS;
4  while flag do
5  |  flag ← false;
6  |  for c ∈ Board do
7  |  |  if c = non-vue then
8  |  |  |  flag ← true;
9  |  |  |  Colorer c en vue;
10 |  |  |  Trouver c', une case colorée en vue proche de c;
11 |  |  |  Colorer en non-vue les montagnes entre c et c';
12 |  |  |  effectuer une nouvelle coloration par DFS;
13 |  |  end
14 |  end
15 end
```

À noter qu'en pratique, pour éviter un dépassement de pile (la profondeur étant en $O(n^2)$ au pire, où n représente les dimensions du terrain), le DFS s'arrête sur une case déjà colorée et qui n'est pas la case de départ. Il est appelé pour chaque case colorée du tableau, et on utilise une heuristique de coloration naïve pour colorer une bonne partie du tableau avant les DFS afin que la profondeur des appels récursifs de ceux-ci soit la plus basse possible. les heuristiques étant linéaires en la taille du plateau, le temps actuel de la génération est finalement à peu près équivalent à celui où on colorerait simplement avec les DFS, et où on aurait une pile d'appel infinie.

Une autre optimisation est de générer des montagnes à une échelle plus grande que les simples cases. Pour cela, on utilise la constante MOUNTAIN_SIZE de GameConstants. On génère les montagnes dans un tableau de dimension Board_Size/Mountain_Size. Cela permet de diminuer grandement la taille du tableau utilisé pour la génération et le rendu est beaucoup plus réaliste. En effet, si on génère aux dimensions 1/1, les montagnes peuvent générer des labyrinthes et former des méandres. Aux dimensions 1/MOUNTAIN_Size, les montagnes forment des plus gros paquets.

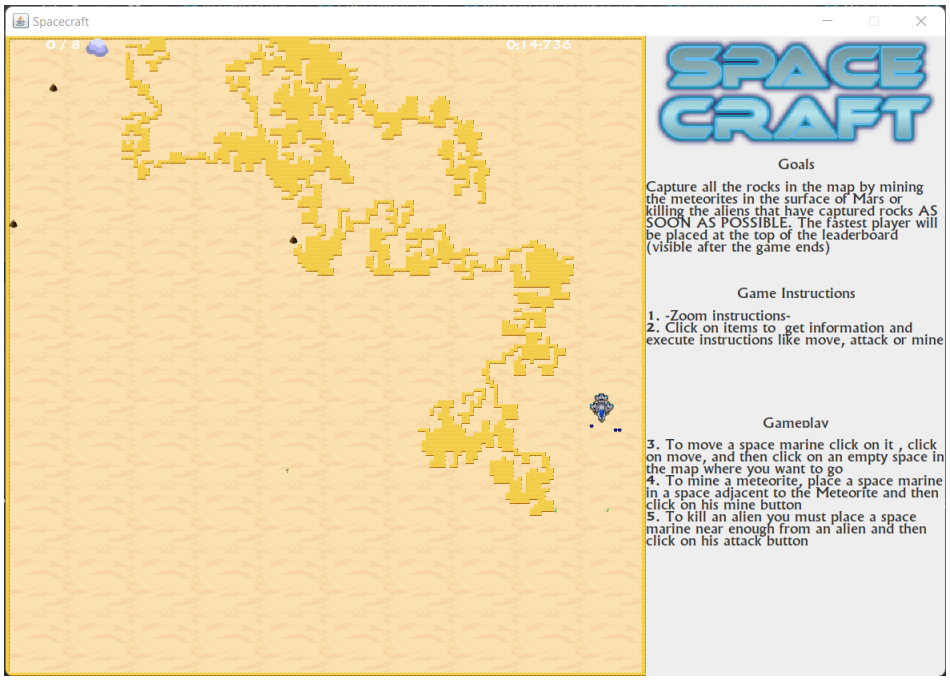


FIGURE 4 – génération sans Mountain_Size

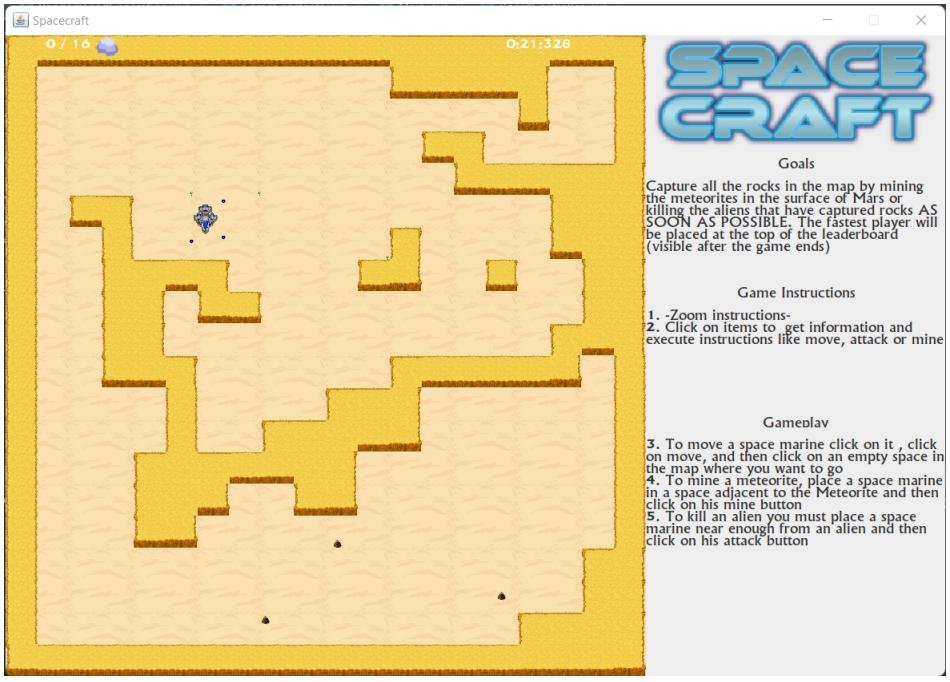


FIGURE 5 – génération avec Mountain_Size

5.1.3 Apparitions des entités

Comme indiqué précédemment, le vaisseau, les space marines et les aliens sont ajoutés au terrain de jeu via la fonction `initLand()`

La première chose que fait cette fonction est de chercher un espace sur le terrain assez grand pour accueillir le vaisseau ainsi que les spaces marines. la taille de cet espace est définie par `gameConstant` *SPACESHIP_LANDING_ZONE*.

Une fois la zone trouvée, on y ajoute le vaisseau au centre, les space marines quant à eux sont ajoutés à une position aléatoire à l'intérieur de la zone trouvé. Voici l'algorithme utilisé pour trouver leurs coordonnés :

```
1 posX ←
    zoneX + tailleZone/2 ± la place restante entre la bordure de la zone et le vaisseau;
2 posY ←
    zoneY + tailleZone/2 ± la place restante entre la bordure de la zone et le vaisseau;
3 while case(posX, posY) n'est pas dans le terrain OR case(posX, posY) est occupée do
4 |   posX, posY ← recalcul;
5 end
```

La dernière chose à faire est donc de rajouter les aliens, pour cela on trouve de manière aléatoire une case vide du terrain et on ajoute un alien à ces coordonnées.

5.2 Création de la fenêtre

La vue de notre jeu est géré par la classe `GameView`. Cette classe est une sorte de `main` pour la vue, c'est via cette classe que l'on crée la fenêtre d'affichage et que l'on instancie tous les composants graphiques de notre jeu.

L'affichage du jeu est subdivisé en deux parties principales, l'affichage du terrain ainsi que l'affichage du panneau de contrôle. De manière concrète, la classe `GameView` étends la classe `JFrame` de l'API `Swing`. `GameView` est ainsi organisé via un `BorderLayout`. Cela nous permet d'ajouter le `BoardPanel`, responsable de l'affichage du côté gauche de la fenêtre (via `BorderLayout.West`) et le `ControlPanel`, responsable de l'affichage du panneau de contrôle du côté droite de la fenêtre (via `BorderLayout.Est`).

C'est aussi via cette classe qu'est rafraîchi l'affichage des composants graphiques. Pour cela, `GameView` implémente l'interface `Runnable` qui nous permet de définir une fonction `run`, celle-ci appelle `repaint` sur tous les composants graphiques de la fenêtre et ce 60 fois par seconde. Afin que cette fonction soit exécutée, on crée un `Thread` nommé `displayUpdateThread` qui exécutera cette fonction `run` tant que le jeu tourne.

5.3 Affichage du terrain de jeu

L'affichage du terrain de jeu ainsi que du score et du chronomètre est géré par la classe `BoardPanel`, celle-ci étends `JPanel`.

5.3.1 Affichage statique

Pour afficher le terrain du jeu, nous avons décidé d'utiliser un système qui a déjà fait ses preuves dans le monde du développement de jeu 2D, la `tilemap`.

Notre terrain est donc décomposé en cases de 16x16 pixels, une case définie l'unité de base pour la taille des éléments affichés. Chaque élément du jeu à une dimension (hauteur et largeur). Cette dimension représente la taille occupée à l'écran en nombre de cases par cet élément. Ainsi, par exemple, les aliens ont une dimension de 1x1, ils sont donc représentés par une case (carré de 16x16 pixels) à l'écran. Un bâtiment qui serait de dimension 4x3 serait représenté par un rectangle de 4 cases par 3, soit 64x48 pixels.

En pratique, on override la fonction `paintComponent` de `BoardPanel` qui nous permet de dessiner sur le panel. Dans cette fonction, on parcourt la liste des éléments du jeu (à savoir les

listes, `mountains`, `structures` et `entities` définies les parties précédentes) et on dessine chacun des éléments du jeu à l'écran en fonction de leur type.

La première "couche" de l'affichage du jeu étant l'arrière-plan, c'est le premier élément dessiné, en particulier, ici l'arrière-plan est une image, on le dessine donc via la fonction via la fonction `drawImage()`. Ensuite, on ajoute les éléments du jeu par-dessus cet arrière-plan.

Plus précisément, tous les éléments du jeu sont définis par la classe `Item`, cette classe a un attribut `view` qui représente la vue de cet élément. Par exemple, à chaque objet `SpaceMarine`, un objet `SpaceMarineView` est associé. C'est dans la classe `View` (ici `SpaceMarineView`) que l'on implémente la fonction `draw` qui définit comment l'élément doit être dessiné. Par exemple, la fonction `draw` de `SpaceMarineView` est définie de manière à dessiner une image de space marine en fonction de sa direction.

Cette implémention nous permet de dessiner simplement tous les éléments du jeu de la manière suivante :

```
for (Element e : elementsOfTheGame){
    e.getView().draw();
}
```

L'ordre dans lequel on dessine les différents éléments du jeu (structures puis entités ou l'inverse) n'a pas d'importance, car les "règles" du jeu définies par le modèle ne permettent pas à deux éléments d'avoir les mêmes coordonnées et donc en pratique, ils ne peuvent pas se superposer.

5.3.2 Affichage dynamique

La section précédente détail comment est affiché le terrain à l'écran à un instant t, mais elle n'explique pas comment est mis à jour cet affichage, c'est ce que nous allons voir dans cette section.

Afin de prendre en compte les changements de l'état du jeu (déplacements des entités, destruction des bâtiments, ...), l'affichage comme vu précédemment est rafraîchi 60 fois par seconde. Cela veut dire qu'environ toutes les 16 millisecondes, on efface ce qu'il y a à l'écran et on redessine l'arrière-plan et les éléments du jeu avec le nouvel état du jeu donné par le modèle.

En pratique, la fonction `repaint` de notre `BoardPanel` est appelé par le `Thread` `displayUpdateThread` définie dans la section *5.2*.

5.4 Zoom de la caméra

Notre terrain de jeu étant grand, on souhaite implémenter un zoom, en fonction du milieu de l'écran. On va donc définir la largeur du terrain de base comme étant 640 unités. On va aussi définir un compteur de la profondeur de zoom initialisé à 0, et les coordonnées du milieu de l'image (320, 320), et un décalage initialisé à (0,0). L'idée est de dessiner l'écran à partir des coordonnées du point en haut à gauche et des dimensions de l'image. Ces dimensions sont calculées en fonction du Zoom. On implémente donc une méthode zoomIn qui zoom d'un cran dans l'écran en fonction. Cette méthode consiste en une incrémentassions de la profondeur si celle-ci est inférieure à une profondeurMax (3), de même avec le dezoom implémenté par zoomOut, qui décrémente la profondeur si celle-ci est strictement positive.

5.5 Déplacements de la caméra

Après avoir implémenté le zoom, on s'est vite rendu compte qu'ajouté la possibilité à la caméra de se déplacer sur le terrain était nécessaire.
Pour que la caméra puisse se déplacer il y a 2 mécanismes nécessaires :
Le premier étant de “déplacer” la caméra à savoir de changer ce qui est affiché à l'écran en fonction du déplacement. Le deuxième est de détecter l'action de déplacement de l'utilisateur (flèches directionnelles, mouvement de souris, clique de souris glissé...) et de la transformer en un mouvement de caméra.

En pratique, on a choisi le déplacement clique de souris glissé. Pour ce faire, chaque déplacement glissé de souris incrémente ou décrémente les coordonnées du décalage. Avant l'affichage de l'image, on effectue une translation en fonction du décalage pour afficher la fenêtre décalée. À noter que la modification du décalage se fait en fonction de la profondeur pour éviter que les déplacements de caméras ne soient trop rapides.

5.6 Dezoom de la caméra

Cependant, un zoom suivi d'un déplacement peut créer une sortie de la fenêtre du terrain lors du dezoom. Il Faut donc corriger ce souci : Lors d'un dezoom, on calcule les coordonnés de chacun des côtés de la future image, pour vérifier si on sort de la fenêtre, auquel cas on effectue une modification du décalage en fonction de la sortie de la fenêtre pour que le dezoom revienne parfaitement. Finalement, zoomOut est comme suit :

```
public void zoomOut(int mouseX, int mouseY) {
       if(evol > 0) {
           int newXGauche = (int) (milieuX - decalageX - dimension/Math.pow(2, evol));
           int newYGauche = (int) (milieuY - decalageY - dimension/Math.pow(2, evol));
           int newXDroite = (int) (milieuX - decalageX + dimension/Math.pow(2, evol));
           int newYDroite = (int) (milieuY - decalageY + dimension/Math.pow(2, evol));
           // On regarde si le dezoom sort par la gauche (x < 0)
           if(newXGauche < 0){
               System.out.println("yo1");
               // On regarde si le dezoom sort par le haut (y < 0)
               if(newYGauche < 0){
                   decalageY += newYGauche;
               }
               // On regarde si le dezoom sort par le bas (y >= dimension)
               else if(newYDroite >= dimension){
                   decalageY -= dimension - newYDroite;
               }
               decalageX += newXGauche;
           }
           // On regarde si le dezoom sort par la droite (x >= dimension)
           else if(newXDroite > dimension){
               // On regarde si le dezoom sort par le haut (y < 0)
               if(newYGauche < 0){
                   decalageY += newYGauche;
               }
               // On regarde si le dezoom sort par le bas (y >= dimension)
               else if(newYDroite > dimension){
                   decalageY -= dimension - newYDroite;
               }
               decalageX -= dimension - newXDroite;
           }
           // On regarde si le dezoom sort par le haut (y < 0)
           else if(newYGauche < 0){
               decalageY += newYGauche;
           }
           // On regarde si le dezoom sort par le bas (y >= dimension)
           else if(newYDroite > dimension){
               decalageY -= dimension - newYDroite;
           }
           evol--;
       }
   }
```

5.7 Affichage du panneau de contrôle

5.7.1 Création du JPanel représentant l'affichage du panneau de contrôle

Le panneau de contrôle est une classe qui hérite de JPanel, ce JPanel est placé dans la JFrame GameView à droite du JPanel du tableau de jeu. Pour ce panel, on a utilisé le LayoutManager appelé CardLayout. Ce layout nous permet d'afficher une "Carte" parmi plusieurs dans le panneau, ici l'idée étant d'avoir une carte par item (voir partie) sur lequel on peut cliquer dans le jeu puis quand on clique sur l'item on affiche la carte correspondante.

5.7.2 Fonctionnement général

Quand un objet est sélectionné (quand l'utilisateur clic sur un objet) on veut avoir un pointeur vers l'objet pour afficher ses données et les actions qu'il peut exécuter. Quand l'utilisateur clique sur un Space Marine et ensuite exécute l'action *Move* on va devoir attendre un deuxième clique sur la case vers laquelle l'utilisateur veut déplacer le Space Marine. On a donc besoin de savoir quelle action vienne d'être exécuté pour savoir si on doit attendre un deuxième clique ou pas (Ceci n'est pas toujours le cas, en fait l'action Mine ni l'action Attack ont besoin d'attendre de recevoir des coordonnées). De plus, on va devoir se souvenir de cette deuxième coordonnée pour la donner comme paramètre à la méthode qui effectue le déplacement.

Ainsi, ControlPanel a un attribut selectedItem pour pointer vers l'objet sélectionné, un attribut waitingAction pour connaître la dernière action exécutée et un attribut targetCoord pour la coordonnée du deuxième clique.

5.7.3 Mécanisme permettant à l'utilisateur de sélectionner une entité en cliquant dessus

Pour sélectionner un objet, on a utilisé la méthode MouseClicked de la classe BoardController (qui hérite de MouseListener) qui se trouve dans le package Controller. Quand on clique, on va directement envoyer les coordonnées de la case sur laquelle l'utilisateur a cliqué au ControlPanel en appelant sa méthode SelectItem. Voici l'algorithme suivi par cette méthode pour traiter les coordonnées :

```
1  clickedCoordinates ← Coordonnées cliqués;
2  boardStructures ← Structures du tableau de jeu;
3  BoardEntities ← Entités du tableau de jeu;
4  waitingAction ← La dernière action exécutée;
5  if waitingAction est vide // waitingAction n'est pas en attente de coordonnées then
6  |  for Structure ∈ boardStructures do
7  |  |  if Structure coordinates == clickedCoordinates then
8  |  |  |  sélectionner cette structure et afficher sa carte dans le ControlPanel ;
9  |  |  end
10 |  end
11 |  for Entity ∈ boardEntities do
12 |  |  if Entity coordinates == clickedCoordinates then
13 |  |  |  sélectionner cette entité et afficher sa carte dans le ControlPanel ;
14 |  |  end
15 |  end
16 end
17 if waitingAction attends des coordonnées then
18 |  targetCoordinates ← clickedCoordinates;
19 |  Exécuter l'action waitingAction ;
20 end
```

Nous avons défini un mécanisme pour choisir une entité en cliquant sur elle, son panneau sera donc affiché. Dans la suite, nous allons donc décrire comment ces différents panneaux ont été créés.

5.7.4 Organisation des différents JPanels

Pour chaque item qui peut être affiché (Alien, Space Marine, Spaceship et Météorite) on a créé un JPanel appelé ItemPanel, chaque JPanel est organisé dans trois sections, du haut vers le bas : une image de l'objet, les actions qu'il peut exécuter et ses données (points de vie par exemple).

5.7.5 Affichage des données correspondantes à l'objet sélectionné

Affichage de la miniature
Pour afficher le images de tous les items on a décidé de les "load" d'une fois pour toutes dans une classe ImageManager, les images sont donc disponibles dans des variables globales. En suite pour afficher la miniature on a décidé de créer un JPanel Thumbnail qui va être placé en haut de chaque ItemPanel (AlienPanel, SpaceMarinePanel..). Le Thumbnail JPanel va dessiner l'image correpondante à une coordonnée spécifique avec un @Override de la méthode paintComponent.

Affichage de la description
D'une façon très similaire on va créer un JPanel description qui va écrire un string contenant la description de l'objet sélectionné avec la méthode drawString dans la méthode paintComponent en bas du ItemPanel.

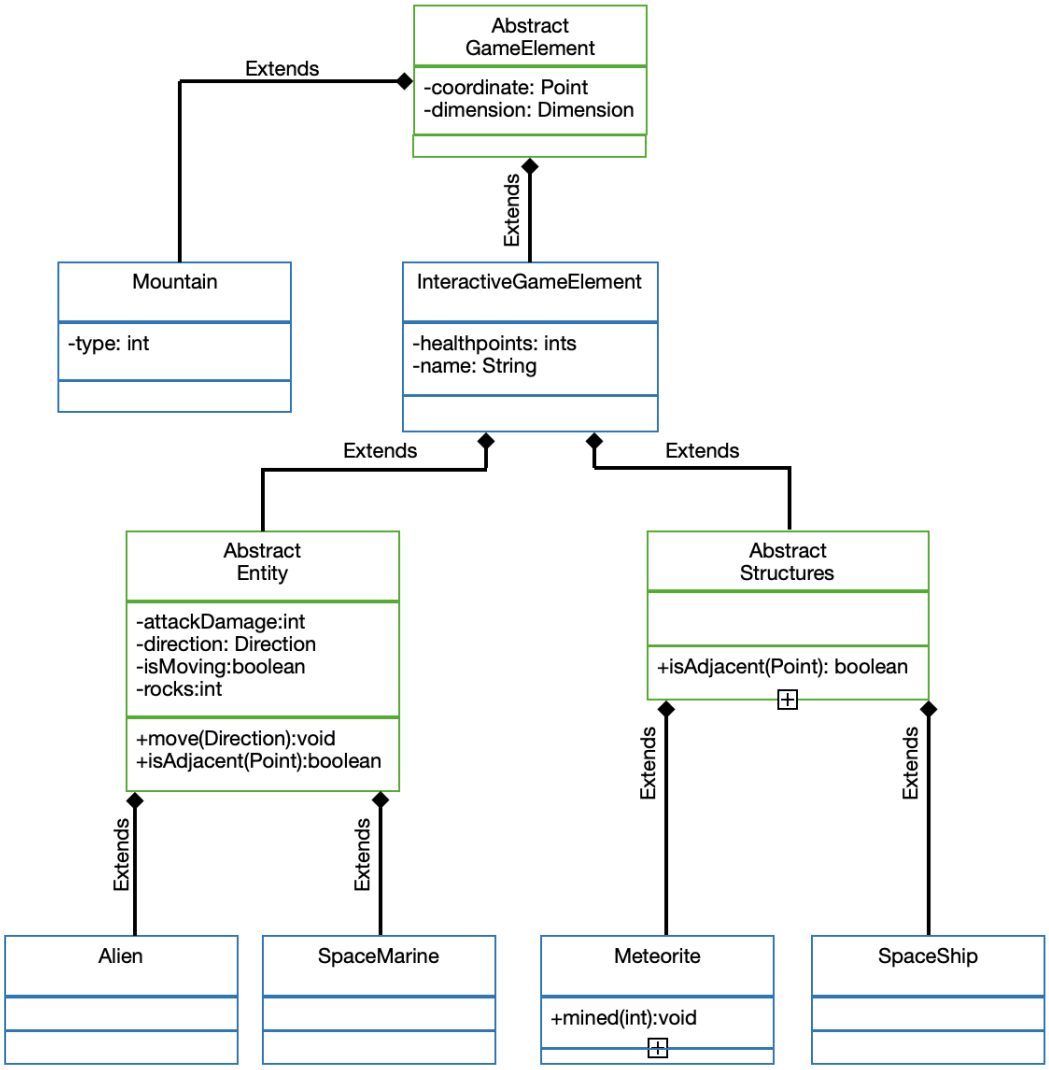
Affichage des données spécifiques
Pour afficher les données spécifiques, on a décidé de créer un StatsPanel qui va être placé au milieu (verticalement) du ItemPanel. Les données principales à afficher sont les points de vie. Pour ce faire on va représenter chaque 50 points de vie avec l'image d'un cœur, ainsi un objet avec 400 points de vie aura dans son ItemPanel 8 cœurs. Ceci est géré directement dans la méthode paintComponent. Le cas des entités avec des roches dans leur inventaire est géré de la même façon, sauf qu'au lieu d'images de cœurs, on utilise des images de roches.

5.7.6 Affichage des boutons d'action

L'affichage des boutons est utilisé que dans le cas du SpaceMarine. Pour cette entité, on a besoin de trois boutons : Move, Attack et Mine. On va créer un JPanel ActionPanel qui va être placé entre le StatsPanel et le Thumbnail du SpaceMarinePanel. On place trois JButtons au centre du panel avec des spaces entre eux. C'est aussi dans cette classe qu'on va mettre les ActionsListeners de chaque bouton. Quand Move est cliqué, on notifie le ControlPanel, sinon on exécute l'action.

5.8 Organisation fatorisée des objets du terrain

Pour faciliter la gestion de nos objets, on a voulu factoriser les classes. Le diagramme de classe suivant montre le résultat :



5.9 Les Space Marines

Les Aliens et les Space Marines sont deux objets qui représentent de personnages qui se déplacent dans la map, ainsi on a créé une classe Entity pour regrouper ces deux classes qui partagent la plupart du code. Le comportement spécifique des Aliens est décrit dans la partie 5.9.

Les Space Marines sont les personnages contrôlés par le joueur, leur génération au début du jeu est faite en même temps que la génération du SpaceShip (voir partie 5.1.3). Les actions des personnages sont dépendantes de l'input de l'utilisateur et sont gérés par le panneau de contrôle.

5.9.1 Déplacement de leur position vers un point sélectionné sur la carte via le panneau de contrôle

On a commencé par ajouter le bouton Move dans le panneau de contrôle du Space Marine. Quand on clique sur le bouton, le Space Marine est sauvegardé comme l'entité à déplacer et le panneau de contrôle se met à l'attente d'un clic sur la map.
Quand l'utilisateur clique sur la map, la méthode mouseClicked() de la classe BoardController qui hérite de MouseListener (Partie controller du modèle Model-View-Controller) va notifier le panneau de contrôle que l'utilisateur a cliquer sur les coordonnées x,y.
Le panneau de contrôle va ensuite vérifier que la case sélectionnée est une case libre : on va chercher dans le tableau des entités et dans le tableau des structures, si aucun objet de ces deux tableaux possède les coordonnées x, y alors on crée un objet de la classe Move (voir partie 5.10) qui sera chargé d'effectuer le déplacement du Space Marine.

5.9.2 Les Space Marines minent des météorites adjacentes

Pour permettre aux Space Marines de miner, on a ajouté le bouton "mine" à son panneau de contrôle. Quand on clique sur ce bouton, on crée un objet décrit dans par la classe Mine qui prend en paramètre le Space Marine et on vérifie s'il est dans une position adjacente à une météorite, si c'est le cas on mine la météorite adjacente trouvée.

5.9.3 Création d'une classe Mine qui gère le thread pour miner

Cette classe hérite de Thread et va créer un thread quand elle sera appelé par son constructeur sur deux objets adjacents : une entité et une météorite. Cet objet va alors lancer un thread qui va diminuer les points de vie des météorites et ajouter des rochers dans les poches de l'entité. Quand la météorite atteint 0 points de vie, le thread la supprime de la liste de structures et libère son emplacement dans les hitboxs.

On introduit une méthode qui enlève des points de vie à la météorite chaque fois qu'elle est appelé. On s'assure que la vie de la météorite ne soit pas négative. De cette manière, on diminue simplement la vie des météorites. À noter qu'il y a une constante damages dans `GameConstants` qui stock les dégâts faits aux météorites à chaque itération, et une constante SpaceMarineDPS qui représente un temps en millisecondes, et qui permet de faire une pause entre chaque itération du minage. Miner un rocher n'est donc pas instantané.

5.9.4 Les Space Marines peuvent attaquer des Aliens dans une case adjacente

Pour permettre aux Space Marines de récupérer les roches qui ont été minés par les aliens, on a introduit la mécanique d'attaque. Similairement à la fonctionnalité Déplacer et Miner, on introduit un bouton Attack au panneau de contrôle qui va permettre à l'utilisateur d'attaquer avec un Space Marine.
Quand on clique sur le bouton *Attack* on crée un objet Fight qui va prendre en paramètre le Space Marine qui va attaquer et le tableau du jeu. Cet objet va se charger de créer un thread quand lors de l'appel à son constructeur qui va diminuer les points de vie de

5.9.5 Création d'une classe Fight dans la classe météorite

Cette classe hérite de Thread et lance un thread qui à chaque tic va diminuer les points de vie de l'Alien, les points de vie enlevés correspondent à une constante du jeu appelé damages dans la classe de variables globales GameConstants. Quand l'alien est tué, on donne tous les minerais qu'il possédait au Space Marine et on l'enlève de la map.

5.10 Aliens

5.10.1 Mécanisme de peur dans les Aliens

Comme mécanique de jeu, on souhaite que les Aliens s'attaquent à des mines. Il faut donc pouvoir repousser les Aliens, afin de protéger les ressources. On ajoute donc deux possibilités : repousser les Aliens et bloquer les Aliens. Le déplacement d'un Alien se fait de façon aléatoire. Il choisit un point dans la map puis s'y rend. Le chemin suivi par l'Alien est alors calculé avec un objet Hitboard. On souhaite que les SpaceMarines ait une sorte d'aura autour d'eux, qui bloquerait les Aliens. Pour ce faire, on a créé un second objet HitBoard dans lequel les SpaceMarines prennent plus de place. Les calculs de chemin des Aliens se font donc avec cet objet Hitboard. De cette manière, si un Alien décide de se rendre à un endroit, il y ira. Puis, si un SpaceMarine se trouve sur son chemin, alors il le contournera, ce qui simule la mécanique de peur. De son côté, un SpaceMarine n'a aucun soucis pour se coller à un Alien, car ses déplacements sont plus rapides. Si c'est le cas, alors l'Alien sera "noyé" dans la hitbox du SpaceMarine, et aucuns chemin ne sera trouvé pour tous les déplacements qu'il souhaite effectuer. Cela est repéré, et il ne bouge simplement pas. Il est donc bloqué ce qui créé la mécanique d'immobilisation. L'immobilisation est importante pour les mécaniques de combat entre Alien et SpaceMarine.

5.10.2 Déplacement aléatoire sur la carte en continu

Pour chaque alien, un thread contrôle le déplacement. Cela se fait dans la classe AlienMovements. L'idée est simple, on choisit aléatoirement une case libre sur la grille puis on s'y rend. Le mouvement effectué l'est de la même manière que pour les Space Marines. Il y a donc de la même manière, un contrôle des collisions, un calcul de plus court chemin par l'algorithme A*, et un thread de mouvement.

5.10.3 Attaque les ressources dont il passe à proximité

Une des bases de gameplay de notre jeu est que les aliens s'attaquent aux rochers à récolter. Pour modéliser cela, on fait en sorte que les aliens puissent repérer les rochers. Il y a donc dans GameConstants, la constante alienRadar qui stock la distance maximum de détection d'une météorite par un alien. À chaque étape du déplacement, les aliens parcours la liste de météorite, et si une se trouve à moins de alienRadar unités en distance euclidienne de l'alien, alors on arrête le thread de mouvement, et on en relance un vers une case adjacente à la météorite trouvée. Une fois collé à la météorite, l'alien en mange un rocher, puis continue sa vie, en relançant u nouveau déplacement.

Cela fonctionne bien, à une subtilité près, lors du nouveau déplacement, l'alien étant adjacent à la météorite, la distance entre lui et cette dernière est inférieure à alienRadar, et il va chercher à la manger. Avec cette implémentation, les rochers se font très vite manger par les aliens, et ce n'est pas ce qu'on souhaite. On va donc créer deux types de déplacements pour les aliens, ceux où ils cherchent les météorites, et ceux où ils se déplacent simplement dans la map. On utilise un booléen lookingForMeteorite dans AlienMovements qui stock quel type de déplacement effectue l'alien. Si lookingForMeteorite est à true, alors on effectue le mouvement comme expliqué précédemment, avec la recherche de météorite proche. Si on trouve une météorite, on met lookingForMeteorite à false. Si lookingForMeteorite est à false, on effectue un simple déplacement aléatoire vers un endroit de la map sélectionné aléatoirement, puis on met lookingForMeteorite à true quand le déplacement est fini. De cette manière, les rochers ne se font pas instantanément mangés par les aliens, et cela correspond beaucoup plus à l'implémentation désirée.

5.11 Mouvement

Une fois un plus court chemin calculé, on peut effectuer un mouvement. Cela se fait dans la classe Movement qui étend Thread. Dans cette classe se trouve la méthode shiftEntity qui selon la classe de l'entité, s'occupe de la déplacer d'un cran dans une direction données dans la hitbox classique, et dans la hitbox du point de vue des aliens. On peut alors s'occuper du mouvement de l'entité dans la méthode run() :

```
1  if la case destination est vide then
2  |  while entité.coordonnées ≠ destination do
3  |  |  parcours ← ShortestPath(entité.coordonnées, destination);
4  |  |  for étape ∈ parcours do
5  |  |  |  if l'étape suivante est vide then
6  |  |  |  |  shiftEntity(entity, direction);
7  |  |  |  |  pause();
8  |  |  |  end
9  |  |  |  else
10 |  |  |  |  break;
11 |  |  |  end
12 |  |  end
13 |  end
14 end
```

De cette manière, tant qu'on n'est pas arrivé, on avance. Si la prochaine étape n'est pas vide, alors on sort de la boucle for et on recalcule un trajet prenant en compte la case non libre.

Deux choses à noter. Premièrement, On ajoute un attribut isMoving à chaque entité. Cet attribut vaut true si l'entité bouge et l'entité bouge tant que cet attribut vaut true. L'attribut est initialisé à true lors des cliques qui lancent un mouvement. Si un SpaceMarine est en mouvement, et que l'utilisateur lui commande un autre mouvement, alors on stoppe son mouvement en mettant isMoving à false. On fait une petite pause pour que son étape de mouvement actuel se termine, puis on relance le nouveau mouvement.

Deuxièmement, si un alien se retrouve bloqué par un SpaceMarine, alors on simule un nouveau mouvement de la même manière que juste au-dessus.

5.12 Plus court chemin

Pour déplacer nos éléments dans le jeu, il nous faut calculer un chemin, et en particulier, un plus court chemin. On va donc pour cela implémenter l'algorithme A*.

Cependant, avant de coder l'algorithme, il faut centraliser les informations de positionnements de tous les éléments et structures sur le terrain, un élément ne pouvant pas marcher sur les montagnes, les bâtiments ni les autres éléments. Pour cela, on a créé la classe HitBoard. Cette classe contient un attribut hitbox qui est une liste 2D de booléens. Une case est vide si sa valeur est vraie. Une case est pleine sinon. Ainsi, cette liste est une modélisation simplifiée et notre terrain, et est mise à jour à chaque déplacement, construction de bâtiments et destruction de bâtiments. Pour calculer le plus court chemin, ce sera dans le graphe formé par cette liste.

Ensuite, on souhaite coder les fonctions pour calculer le plus court chemin. Cela se trouve dans la class ShortestPath. Pour coder, A*, il nous faut dans un premier temps une heuristique pour choisir le chemin optimal. On a choisi pour cela la distance euclidienne. En fait, une case est évaluée en fonction de sa distance euclidienne avec le point d'arrivée, et du nombre d'étapes avec le point de départ. On parlera de G-value pour le nombre d'étapes depuis le point de départ, de H-value pour la distance euclidienne avec le point d'arrivée, et de F-value pour la somme des deux. L'algorithme A* va calculer le chemin avec des nœuds de G-values minimales.

Pour représenter les étapes de notre chemin, on construit la classe Node. Un objet de cette classe contient ses coordonnées dans le modèle, sa G-value, sa H-value, sa F-value, ainsi qu'un pointeur vers son parent, le noeud de l'étape précédente. L'algorithme A* est alors le suivant :

```
1  start.G ← 0;
2  start.H ← heuristique(start, end);
3  start.F ← start.G + Start.H;
4  Open ← [start];
5  Close ← [];
6  while start ≠ [] do
7  |  currentNode ← nœud de plus petite F-value dans Open;
8  |  if currentNode = end then
9  |  |  break
10 |  end
11 |  retirer currentNode de Open;
12 |  ajouter currentNode à Close;
13 |  for child ∈ currentNode.children do
14 |  |  cost ← currentNode.G + 1;
15 |  |  if child ∉ Open then
16 |  |  |  if g(child dans Open) > cost then
17 |  |  |  |  child.G ← cost;
18 |  |  |  |  child.F ← child.G + child.H;
19 |  |  |  |  Open ← Open ∪ [child];
20 |  |  |  end
21 |  |  |  else
22 |  |  |  |  currentNode.g ← g(child dans Open) + 1;
23 |  |  |  |  child.F ← child.G + child.H;
24 |  |  |  |  Open ← Open ∪ [child];
25 |  |  |  end
26 |  |  end
27 |  |  else if child ∉ Open ∧ child ∉ Close then
28 |  |  |  child.h ← heuristique(child, end);
29 |  |  |  child.G ← cost;
30 |  |  |  child.F ← child.G + child.H;
31 |  |  |  Open ← Open ∪ [child];
32 |  |  end
33 |  end
34 |  backtrack(end);
35 end
```

Une fois l'algorithme fini, chaque noeud connaît son parent pour aller de l'arrivée vers le point de départ. Pour retrouver le chemin, il faut donc utiliser le backtracking. Le backtracking consiste en un parcours de la liste chaînée formée par les noeuds, et le pointeur parent de chaque noeud. Pour reconstituer le chemin final, On stock donc ces noeuds chaînés dans un tableau.
Enfin, en faisant la différence entre les coordonnées d'une étape du parcours et celles du point d'après, on obtient un point parmi (1,0), (-1,0), (0,1), (0,-1). Cela nous donne donc la direction à suivre pour un déplacement. On va calculer un parcours puis avancer tant que la prochaine étape du parcours est libre (car le terrain peut être modifié après le calcul du parcours). Si la case suivante n'est pas libre, on recalcule un plus court chemin, etc.

Dans un souci d'optimisation, la structure de données utilisées pour Open et Close n'est pas une simple liste, mais un tas binaire. un tas binaire est un arbre binaire dont la racine contient la plus petite valeur de l'arbre selon une relation d'ordre donnée, et dont chacun des sous-arbres de la racine est un tas binaire. Cela permet un accès à la plus petite valeur d'Open en $O(1)$ au lieu de $O(n)$, le calcul pour savoir si un élément est dans un ensemble se fait en $O(n)$ pour les deux, et la suppression d'un élément se fait en $O(log(n))$ au lieu de $O(n)$. Enfin l'ajout d'un élément dans l'ensemble se fait en $O(log(n))$ au lieu de $O(1)$. Toutes les opérations faisables sur une liste sont donc faisables sur un tas binaire et sont toutes beaucoup plus efficaces, sauf pour l'ajout d'un élément à l'ensemble, mais une opération en $O(log(n))$ est quand même très rapide.
Dans notre implémentation, on utilise aussi deux tableaux de taille Board_Size×Board_Size. Le premier est un tableau de booléen stockant si un nœud est présent dans Close. Le second est un tableau de float stockant si un nœud est présent dans Open, et si c'est le cas sa g-value, sinon -1. De cette manière, le calcul de la présence d'un nœud dans Close se fait en $O(1)$, et de même pour Open. Finalement, la seule opération linéaire est dans le cas où un nœud est bien présent dans Open, et où on le retrouve soit pour le modifier, soit pour modifier currentNode.

5.13 Spawn des météorites au lancement de la partie

Création d'une structure pour les Météorites
On commence par créer la classe météorite qui hérite de la classe Structure, cela implique que les météorites ont des Healthpoints, une dimension, une coordonné et une capacité mise par défaut à 0. Le paramètre principal a tenir en compte est le healthpoints, ceux-ci sont liés aux nombres de roches que les Space Marines peuvent collecter.

5.13.1 Affichage dans le tableau de jeu

En suite il faut créer la classe MeteoriteView qui hérite de ItemView et MeteoriteTile qui hérite de ItemTile. MeteoriteTile est chargé d'appeler la fonction draw dans laquelle on choisi de dessiner les roches d'une couleur marron. La MeteoriteView met son attribut tileView à une instance de MeteoriteTile.

5.13.2 Spawn des météorites au début et pendant le jeu

Le nombre initial de météorites est déterminé par le joueur, la méthode chargé de la génération aléatoire des météorites est basé sur celle du vaiseau spatial (voir partie 5.1.3). Elle est écrit dans la classe GameBoard sous le nom initMéteorites. L'algorithem suivi pour la génération est décrit ici :

À la fin de l'execution de cet algorithme un nombre nbMeteorites de météorites vont avoir été crées et ajoutés dans le Game Board.

5.13.3 Affichage des données du météorite dans le Control Panel

En fin il ne reste que la création du panel MeteoritePanel qui hérite de JPanel. Cette classe

```
nbMeteorites ← user input;
meteoriteSurface ← 2x2;
for nbMeteorites do
    foundLocation ← false;
    while not foundLocation do
        newCoordinates ← random coordinates;
        isThisPlaceBigEnough ← false;
        surfaceBigEnough ← is meteoriteSurface area around newCoordinate available;
        if newCoordinates is empty and surfaceBigEnough then
            isThisPlaceBigEnough ← true;
        end
        if isThisPlaceBigEnough then
            Create a meteorite with a random amount of health between 300 and 400 ;
            Add it to the Game Board at the location newCoordinates;
        end
    end
end
```

contienne une nouvelle méthode qui va afficher les données de l'objet : les Health Points restants, une image et une déscription (voir partie 5.7).

5.14 Affichage du score

Score

Pour faire le score, on a créé une classe Score qui compte le nombre de roches dans les poches des Space Marines et compte initialement le nombre de roches qui existent dans la map. Pour compter le nombre initial de roches, il suffit d'additionner les points de vie de toutes les météorites et ensuite le diviser par la constante de jeu Damages. Ceci nous donne le nombre de roches existantes

Timer

On a décidé de coder cette classe entièrement, pour ce faire on a utilisé System.currentMillis() pour enregistrer l'heure précise quand on a déclanché le timer dans une variable startTime. Puis pour connaître le temps écoulé, on fait simplement System.currentMillis()-startTime. Ce temps écoulé peut être obtenu avec un getter. Ce getter va rendre un long qui représente le temps écoulé en millisecondes, il faut donc le transformer en minutes, secondes et millisecondes lors de l'affichage. Le temps dans le jeu est très important puisqu'on classifie le meilleur joueur comme celui ayant terminé le jeu le plus rapidement possible et donc il faut afficher les millisecondes.

Affichage

Pour afficher le score et le timer on a créé les attributs score et timer dans le tableau du jeu (Dans le modèle). Ensuite, quand on crée le JPanel correspondant à la view du tableau (Dans la vue), on va avoir deux attributs score et timer qui vont correspondre aux valeurs des attributs score du modèle. Pour afficher le score et le timer dans l'écran, on va utiliser la méthode paintCompenent, dans cette méthode, on va appeler les méthodes drawTimer et drawScore qu'on a créé pour afficher ces deux données dans le bord supérieur de l'écran. La police d'écriture utilisée est la même dans le deux cas et la couleur du texte choisie est blanche pour contraster le plus avec les couleurs de la map (orange principalement).

5.15 Le Leaderboard

Le fichier score.txt

Pour sauvegarder le leader board on a créé un fichier de texte score.txt (dans le package resources). Le format des données du fichier est : "nom_temps" le nom étant le nom du joueur rentré dans le launcher et le temps étant le temps qu'il a pris à finir le jeu en millisecondes. En

plus les données du fichier sont séparés par des virgules.

Mécanismes de gestion du LeaderBoard
Les méthodes qui modifient ou lissent le score.txt sont dans la classe LeaderBoard. Pour gérer les couples nom, temps, on a créé une classe qui s'appelle BestScore qui a deux attributs : String username et int score. Le fonctionnement général est le suivant : on lit les scores dans le fichier (on split une première fois pour les virgules et une deuxième fois pour l'underscore) et on sauvegarde la couple de valeurs obtenues dans une ArrayList de BestScore. Ensuite on ajoute le score que l'utilisateur viens d'obtenir et finalement on réécrit la liste dans le fichier.

L'affichage du Leaderboard
Pour faire le leaderboard on a créé une classe LeaderBoardView qui hérite de JPanel dans laquelle on affiche les top 5 joueurs. Il suffit d'obtenir la liste de BestScore en instanciant la classe LeaderBoard puis afficher ses cinq premières valeurs. Ce JPanel va être affiché après la fin de la boucle de jeu dans un nouveau JFrame appelé LeaderBoardWindow.

5.16 Launcher

Le launcher est la première chose que voit l'utilisateur lorsqu'il lance le jeu. C'est la fenêtre d'affichage qui lui permet de changer son nom, les paramètres de la partie et de lancer la partie.

Pour une question de simplicité et de modularité, nous avons décidé de créer une fenêtre d'affichage différente de celle du jeu pour le launcher. Nous avons donc une classe `LauncherWindow` qui étend la classe `JPanel`. Cette fenêtre est composée d'un panel principal nommé `LauncherPanel` qui étend `JPanel`. Ce panel est composé de plusieurs sous élément organisé via un `GridLayout`. On a donc de haut en bas :

— Le logo
— Le panneau de configuration
— Le bouton *start*

Le logo est représenté par un `JLabel` auquel on a ajouté une icône. De la même manière, le bouton *start* est une simple *JButton* auquel on a ajouté une image et enlevé toutes les animations (mouseOver, ...). Le panneau de configuration est un autre panel. Il est organisé via un `GridLayout` ayant 7 lignes et 2 colonnes. Chaque ligne correspond à un paramètre, les éléments dans la colonne 1 sont des `JLabel` représentant le nom du paramètre, la colonne 2 contient la valeur du paramètre, en pratique cette valeur est représenté par un `JTextField` ou un `JSlider`. Voici la liste des 7 paramètres modifiables :

— Le nom du joueur
— Le nombre de space marines
— Le nombre d'alien
— Le nombre de météorites
— Le nombre de chaînes de montagne (sans compter le contour)
— La difficulté : le rayon du cercle dont le centre est le space marine dans lequel les aliens sont bloqués s'ils y rentrent.
— La seed de la carte
— Le bouton reset leaderboard : cliquer pour effacer le fichier score.txt

Lorsque l'on appuie sur le bouton *start*, les valeurs des paramètres sont enregistrées dans des variables `static` de la classe `GameConstants`. Ces valeurs seront ensuite utilisées à l'initialisation du jeu. Une fois cela fait, la fenêtre du launcher est fermé via la fonction `dispose()`. Enfin, le jeu est lancé en instanciant les classes suivantes : `GameEngine`, `GameView`, `BoardController`.

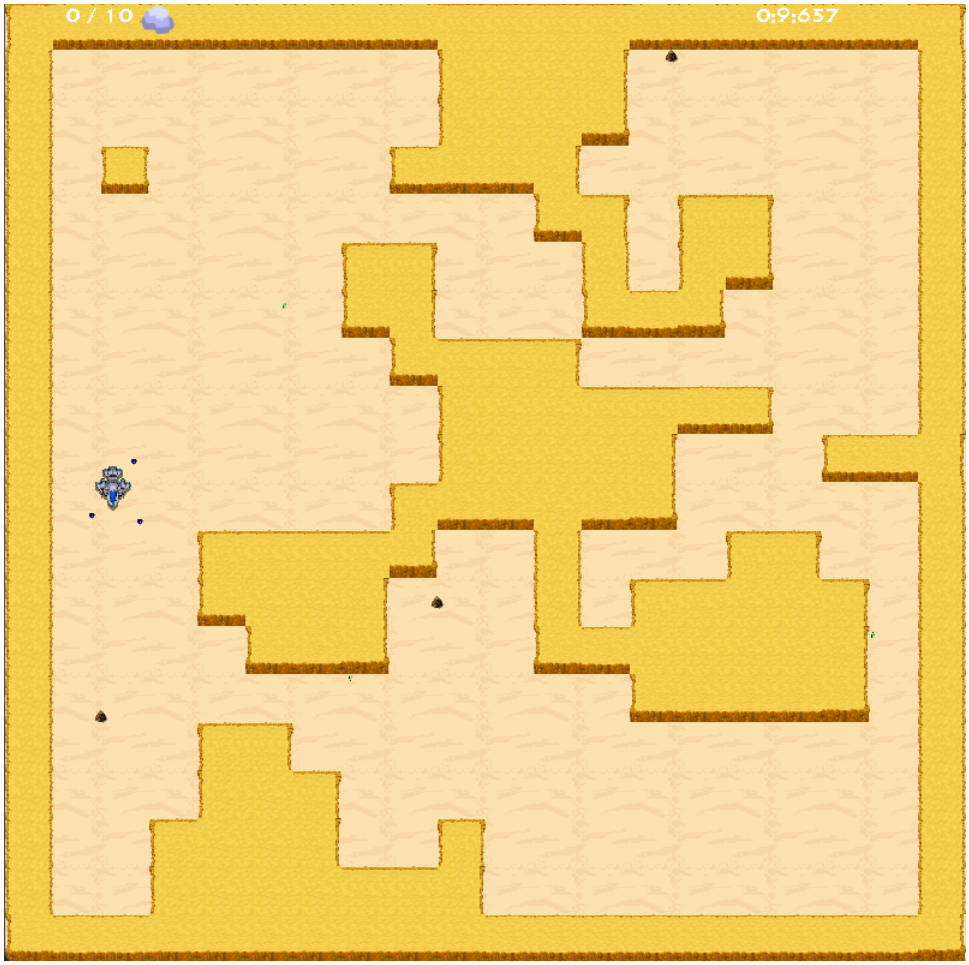
6 Résultat

Le résultat du jeu est très satisfaisant. On a réussi à implémenter nos idées, et à rendre un jeu amusant, jouable et beau. La paramétrisation des parties permet d'éviter au jeu d'être trop répétitif. Cela est aussi permis par la génération totalement aléatoire du terrain de jeu. Le choix de la difficulté est assez bien pour pouvoir challenger tout joueur sur ce jeu, des débutants aux plus adroits.

Le jeu se lance donc sur un launcher :

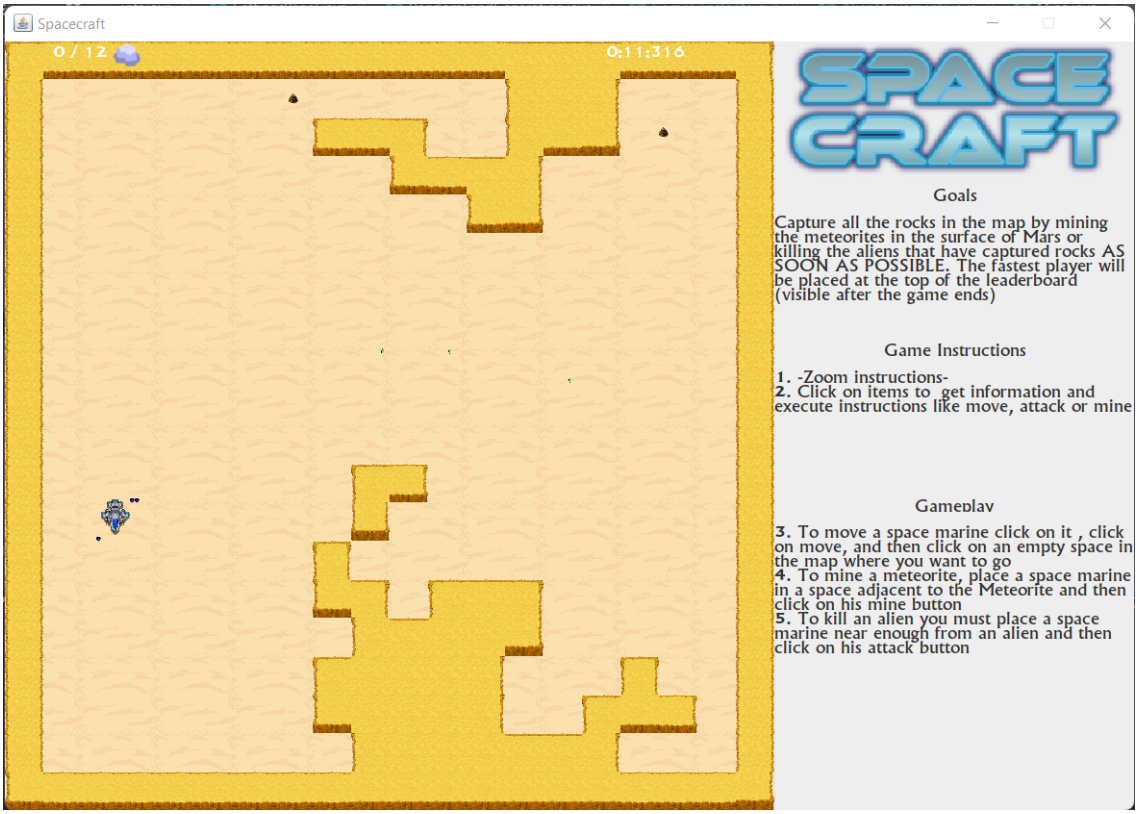
On peut y voir les modifications possibles de quelques paramètres du jeu, le bouton de réinitialisation du leaderboard, ainsi que le bouton de lancement du jeu.

Une fois cliqué sur le bouton *START*, on génère un terrain aléatoire. Les terrains générés ont l'allure suivante :

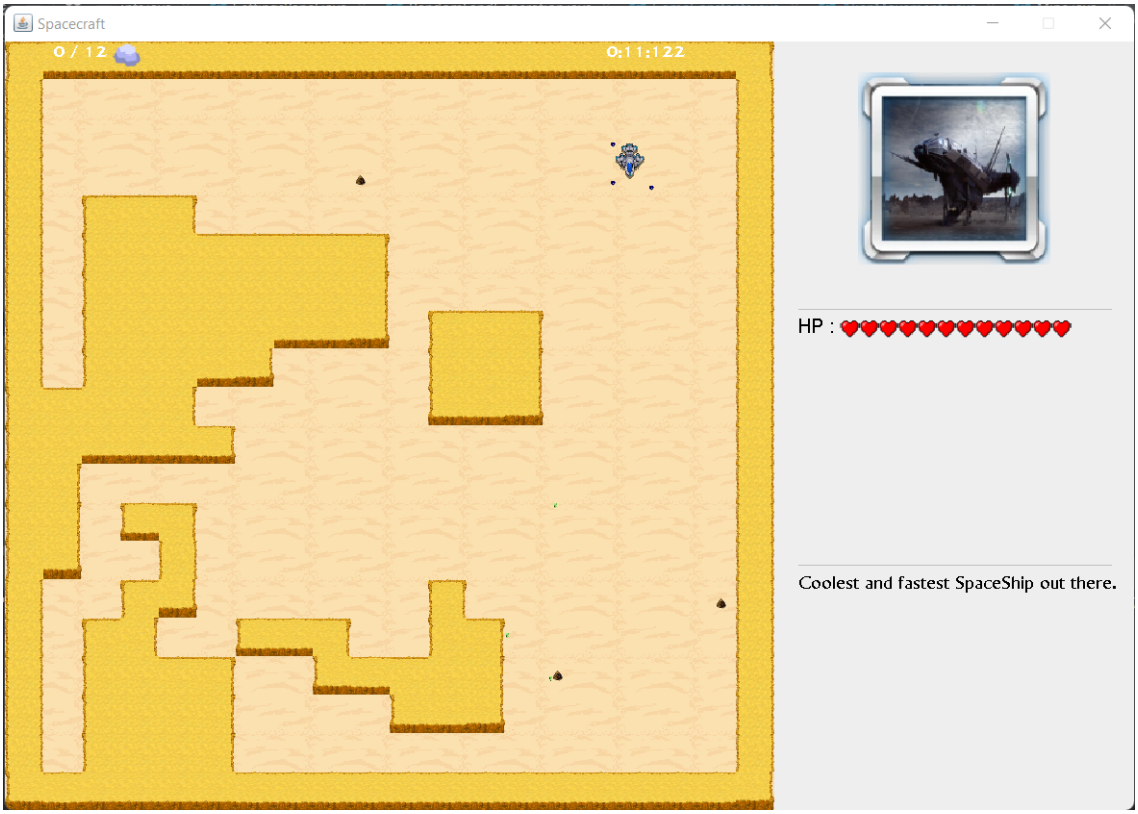


On peut y voir notamment la case d'apparition du vaisseau et des Space Marines

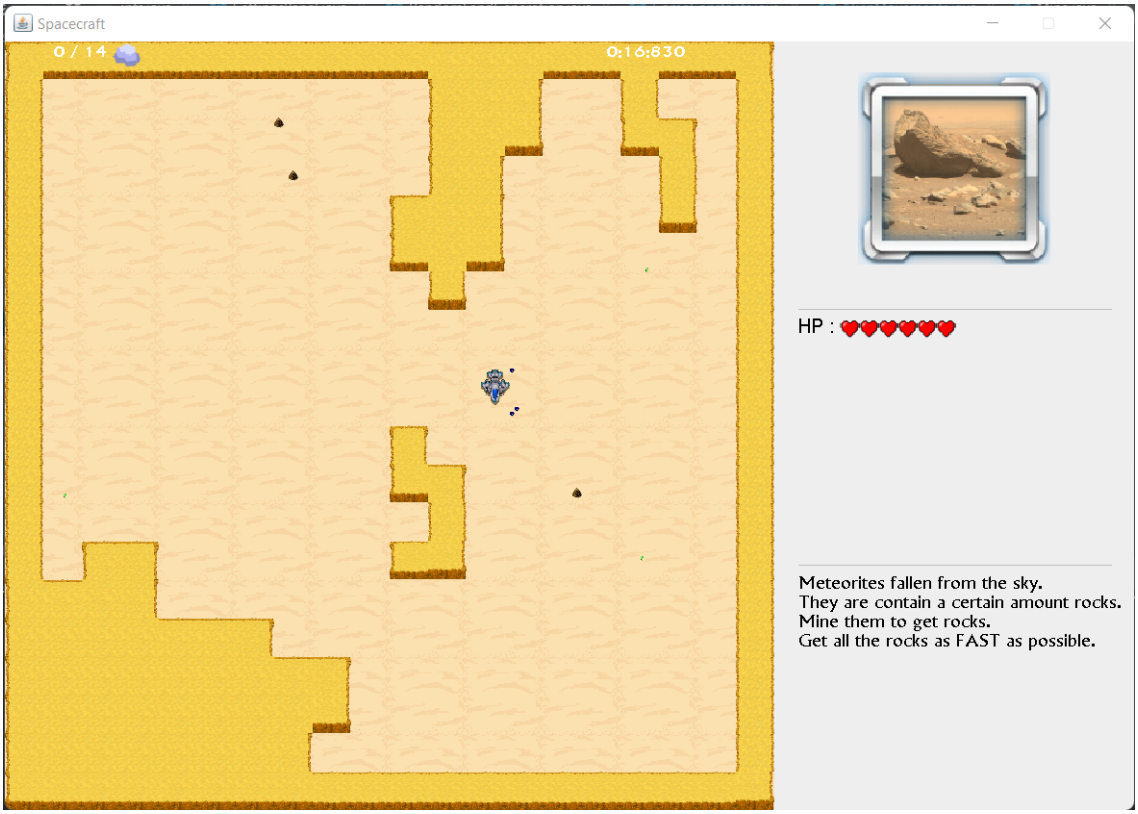
À droite de la fenêtre, se trouve le control panel, qui est initialisé avec les règles du jeu :



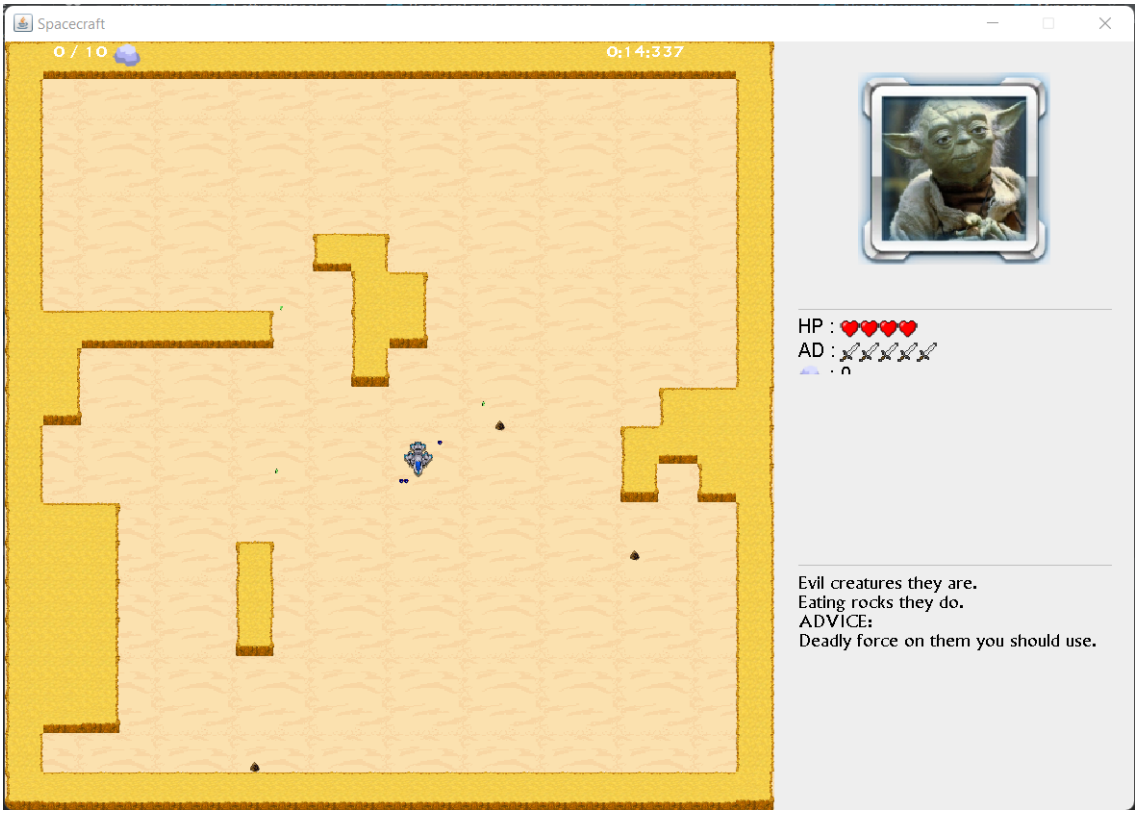
Le control panel change quand on clique sur les différents éléments du jeu.
Le Spaceship :



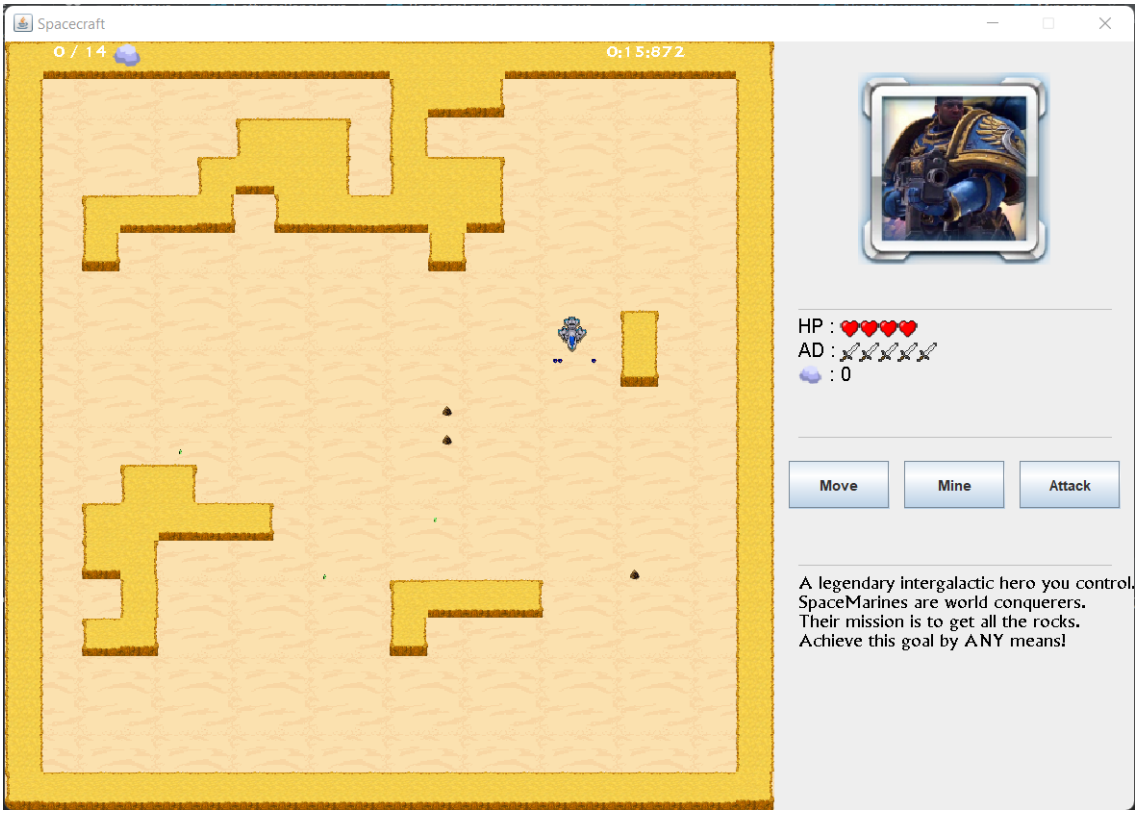
Les météorites :



Les aliens :



Les Space Marines :



On trouve de plus en haut à gauche un compteur des rochers récupérés :

Et un chronomètre en haut à droite du terrain de jeu :

Par ailleurs, les actions sont très fluides. Les clics sont précis, les déplacements fluides et les calculs faits durant la partie sont très rapide. Même les déplacements, dans un terrain très chargé, s'exécutent en une fraction de seconde. On a réussi à empêcher deux entités de se superposer en gardant le jeu fluide.

Tous ces évènements sont gérés par des threads. Une partie lancée avec beaucoup d'entités n'a que peu de latence, ce qui témoigne de la bonne gestion des threads. Il n'y a pas non plus d'accès à des zones interdites, ou de superpositions d'entités qui pourraient être dû à des accès simultanés à une ressource. Nous avons donc bien maîtrisé l'accès aux ressources critiques.

Le control panel permet de donner des informations sur l'élément sélectionné. Il permet notamment d'enclencher des actions de déplacements, de minage, et d'attaque. On y trouve aussi des informations sur le lore du jeu, ce qui participe au côté immersif.

Le jeu est immersif. En effet, les fonctions de zoom et de déplacements permettent de se plonger dans ce monde. Nous avons fait le choix de créer un grand terrain. C'était un choix délibéré, nous permettant de nous exprimer beaucoup plus sur les fonctions de générations de terrain, mais aussi pour donner un plus grand intérêt au but du jeu.

Le but du jeu est de récolter le plus rapidement possible les ressources se trouvant sur le terrain. Pour cela, il faut s'y rendre avec un SpaceMarine, puis effectuer une action de minage. Lorsqu'un SpaceMarine mine, et qu'on clique sur la météorite minée, on voit sa vie diminuer. Si le SpaceMarine est sélectionné, on voit son compteur de rochers augmenter. Il en est de même lors d'une attaque sur un Alien.

Les Aliens mangent les météorites. On a réussi à modéliser ça de façon à ce que le jeu reste jouable, sans être trivial. La difficulté principale du jeu réside dans le fait protéger les rochers

des Aliens, et si on échoue, de chasser les Aliens en les attaquants. La sélection de la difficulté est en fait une sélection du rayon de capture d'un alien. Avant de l'attaquer, il faut s'en approcher suffisamment pour l'immobiliser, puis se coller à l'alien, en enfin l'attaquer.

Quand la partie est finie, une nouvelle fenêtre s'ouvre sur le leaderboard : un tableau des records locaux. C'est un ajout très intéressant, car ça intègre un esprit de compétition au jeu.

On y voit les meilleurs temps inscrits à côté de l'identifiant entrés dans le launcher.

7 Documentation utilisateur

7.1 Lancer le jeu

Pour lancer SpaceCraft il faut lancer l'exécutable avec la commande `java -jar spacecraft.jar`, si vous n'arrivez pas à lancer le jeu, veuillez utiliser Java 17. Une fois ceci fait, le launcher va se lancer et un menu va apparaître dans lequel il faut saisir des informations en cliquant dans les différents champs de texte :

— Le nom de l'utilisateur.
— Le nombre de Space Marines (personnages contrôlés par l'utilisateur).
— Le nombre d'aliens (ennemis).
— Le nombre de météorites (il faut toutes les capturer pour gagner).
— Le nombre de chaînes de montagnes.
— Le pourcentage de la map recouvert par les montagnes.

En suite il suffit de cliquer sur le boutton start.

7.2 Comment contrôler la caméra ?

Pour faire un zoom dans la map il faut scroller avec la souris. Ensuite pour déplacer la caméra il faut click-and-drag la map. Pour une meilleure expérience de jeu on récommende de jouer avec une souris.

7.3 Comment jouer ?

Gagner une partie
Pour gagner le jeu il faut récolter tous les minerais disponibles, pour faire cela le joueur a deux options : soit il mine des météorites pour obtenir leurs minerais, soit il tue des aliens pour voler les minerais dans leur inventaire.

Déplacer un Space Marine
Pour déplacer un Space Marine, il faut cliquer sur un des Space Marines dans la map (les figures bleues), ses actions vont ensuite être affichées. Pour déplacer le personnage, il faut cliquer sur *Move* une fois, puis cliquer sur une case libre vers laquelle on veut déplacer le Space Marine.

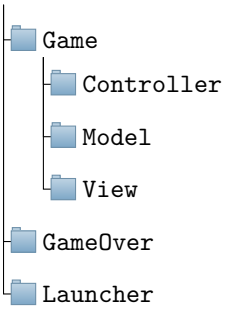
Miner une météorite pour obtenir des minerais
Pour miner un minerai, il faut déplacer le Space Marine jusqu'à une case adjacente d'une météorite. Ensuite il faut cliquer sur le Space Marine et cliquer sur le bouton Mine, la météorite adjacente sera en suite minée automatiquement.

Attaquer un Alien pour voler ses minerais
Pour attaquer un Alien, il faut placer un Space Marine dans une case adjacente à un Alien, attention ! Ceci n'est pas facile. Les Aliens deviennent immobiles quand un Space Marine se trouve trop prêt d'eux, c'est un mécanisme de défense. Le secret pour intercepter un Alien est donc de prédire sa trajectoire et déplacer un Space Marine vers le point dans lequel on a prédit la possibilité d'intercepter l'Alien.
Une fois l'alien intercepté, il faut déplacer le Space Marine à une case adjacente à l'alien et ensuite utiliser le bouton *Attack*. Quand l'Alien meurt, le Space Marine obtiendra tous ses minerais.

8 Documentation Développeur

8.1 La structure du code

Le jeu est constitué de trois éléments, le jeu en lui-même, le launcher et l'écran de fin de partie, il y a donc un package pour chacun de ses composantes. Ensuite, l'écrasante majorité du code se trouve dans le package `Game`. Comme précisé dans une partie précédente, nous avons choisis d'utiliser le motif *MVC*, le code du jeu est donc assez logiquement organisé de la même manière, il y a donc à l'intérieur du package `Game` les packages `Model`, `View` et `Controller`. Voici donc à quoi ressemble l'arborescence de fichiers de notre projet :



La classe main nommé `Main` permet de lancer le jeu, elle se trouve, elle aussi, à la racine du projet, il est cependant peu probable que vous ayez besoin de la modifier afin de faire des ajouts au jeu.

8.1.1 `Game`

`Controller` Dans ce package, se trouve uniquement la classe `BoardController`. Cette classe est responsable de la gestion des actions que l'utilisateur fait sur l'affichage du terrain de jeu. C'est cette classe qu'il faudra modifier si vous avez envie d'ajouter des nouvelles interactions avec l'utilisateur, par exemple lui permettre d'utiliser le claver pour se déplacer sur le terrain.

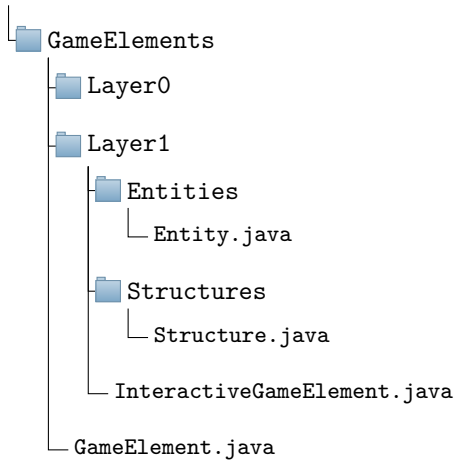
`Model` Si vous voulez rajouter de nouveaux éléments ou mécaniques de jeu, c'est dans ce package qu'il va falloir faire des changements.

`GameBoard` Ce package contient la classe `GameBoard` qui est la manière dont le jeu est représenté, a priori vous n'aurez pas à modifier cette classe. Dans ce package, on retrouve aussi la classe `RandomLandGenerator` qui génère aléatoirement les montagnes sur la carte. Si vous voulez une nouvelle mécanique comme la génération de geysers présents sur le terrain, alors, c'est de ce package qu'il faudra créer la classe `GeyserGeneration`.

`GameElements` C'est dans ce package que l'on retrouve tous les éléments du jeu, les éléments se distingue en plusieurs catégories :

- Layer0 : Les éléments lesquels on ne peut pas interagir (ex : les montagnes).
- Layer1 : Les éléments lesquels on peut interagir.
  - Entités : Les éléments interactifs mobiles.
  - Structures : Les éléments interactifs statiques.

Si vous voulez ajouter un nouvel élément au jeu, c'est à vous de définir dans quelle catégorie rentre cet élément. Vous pourrez ensuite créer une classe dans le package correspondant. Il est important de noter qu'il existe des classes abstraites qui factorisent les caractéristiques de chaque catégorie, à vous de bien définir dans quelle catégorie rentre votre nouvel élément pour profiter de ces classes abstraites. Par exemple, toutes les structures (ex : vaisseau spatial) étendent la classe abstraite `Structure` présente du package `Structures`. De la même manière, tous les éléments du jeu étendent la classe `GameElement`. Voici donc à quoi ressemble l'arborescence du package `GameElements` :



`Scoring` Ce package contient les classes `Score` et `Timer`. Vous n'aurez a priori rien à toucher dans ce package, hormis peut-être si vous voulez changer profondément le gameplay du jeu et notamment la manière dont le score est calculé.

Pour finir, on trouve dans le package `Model`, la classe `GameEngine` qui ne devrait pas vous intéresser. Cependant, on y trouve aussi la classe `GameConstants` qui elle est bien plus intéressante dans le cadre de la modification du jeu. Dans cette classe, on retrouve toutes les constantes liées au gameplay, en voici une liste non exhaustive :

— la taille du terrain
— le rayon du cercle protecteur des space marine (dans lequel les aliens ne peuvent pas entrer)
— la taille de la zone d'atterrissage du vaisseau
— les dégâts que font les entités.

`View` Dans ce package, se retrouve 2 packages, `Board` et `ControlPanel`.

`Board` Ce package contient le package `Views` dans celui-ci on retrouve les views de chacun des éléments du jeu, en effet à chaque élément du jeu une vue est associé, on a alors par exemple pour la classe `Alien`, la classe `AlienView` qui lui est associé. Par conséquent, si vous avez ajouté un élément dans le jeu comme indiqué précédemment, il vous faudra créer dans le package `Views` la vue correspondant à cet élément. Dans le package `Board`, on retrouve aussi la classe `GameBoardPanel` qui correspond à l'affichage du terrain de jeu, vous n'aurez a priori rien à modifier dans cette classe.

`ControlPanel` Dans ce package, se trouve plusieurs sous packages, on a donc le package `ElementsPanel` qui regroupe les panels correspondant à chaque élément du jeu, on a donc par exemple pour la classe `Alien`, le panel `AlienPanel` qui lui est associé. De la même manière que pour la view, si vous ajouter un élément au jeu, il faudra lui créer un panel associé dans le package `ElementsPanel`.

Ensuite dans ce panel, on retrouve aussi le package `Panels`, dans celui-ci se trouve les panels qui composent les panels de `ElementsPanel`, on y retrouve ainsi `ActionPanel`, `DescriptionPanel` ou encore `ThumbnailPanel`. Pour être plus claire, prenons par exemple le panel `AlienPanel`, il est composé des panels `ThumbnailPanel`, `StatsPanel` et `DescriptionPanel`.

Enfin, dans le package `ControlPanel` on retrouve les trois classes suivantes, `GameView`, `RessourceManager` et `ViewConstants`. `GameView` n'est a priori pas à modifier puisque c'est là où l'on crée la fenêtre, etc. `RessourceManager` et `ViewConstants` sont quant à eux bien plus susceptible de vous intéresser. `RessourceManager` est la classe dans laquelle on importe toutes les polices d'écriture et images, si vous faites ajouts au jeu ça peux donc être utile. `ViewConstants`

est quant à elle la classe dans laquelle se trouve toutes les constants liées à la vue, en voici une liste non exhaustive :

— le nombre de *fps* du jeu
— la largeur de l'affichage du terrain de jeu en pixel
— la taille de carreau composant le terrain de jeu (donc la taille des éléments affichés)

8.1.2 `GameOver`

Dans ce package, on retrouve la classe `LeaderBoardPanel` qui correspond à l'affichage de fin de la partie, à savoir le leaderboard. C'est pour l'instant, c'est une version assez simpliste pour un écran de fin de partie, mais vous êtes fortement encouragés à l'améliorer. Pour cela il vous faudra modifier la classe `LeaderBoardPanel` pour ajouter des nouveaux éléments, ici tout est à faire, faite donc preuve d'imagination.

8.1.3 `Launcher`

Ce dernier package correspond à l'affichage du launcher à savoir la fenêtre qui se lance avant de lancer le jeu afin de configurer les paramètres de la partie. Le launcher est plutôt à un stade avancé de développement par rapport à l'écran de fin de partie par exemple, nous vous conseillons donc de ne pas passer trop de temps dessus dans un premier temps. Cependant, si vous avez ajouté une fonctionnalité au jeu et que vous souhaitez laisser la possibilité à l'utilisateur de modifier les paramètres de cette fonctionnalité avant le lancement du jeu, il est possible de modifier la classe `LauncherPanel` afin d'ajouter par exemple un champ de saisi de texte ou un curseur par exemple.

9 Conclusion et perspectives

On a donc implémenté le jeu comme on le souhaitait. Cela n'a pas été facile. La génération du terrain se devait de ne pas isoler de zones. Faire tout ceci efficacement n'était pas évident et était nécessaire, car on a un grand terrain. Le grand terrain a aussi été une difficulté pour le choix du plus court chemin parcouru par les entités. Rendre la fonction exacte et rapide n'a pas été évident. Le résultat est très satisfaisant. Les fonctions de zoom ont aussi demandé de la réflexion à plusieurs, pour avoir un bon résultat. Pour le control panel, le choix de layout n'était pas évident, et à nécessiter beaucoup de documentation. En général, tout ce qui touche à l'agencement des fenêtres a nécessité beaucoup de documentation et de tests pour arriver au résultat final. On a aussi rencontré des difficultés pour éviter les collisions. On a développé des stratégies de hitboxs notamment pour résoudre ces soucis. Finalement, ce projet nous a permis d'approfondir nos connaissances dans beaucoup de domaines comme l'algorithmie, le multi-threading, l'utilisation des modules de layout de java, ainsi que des compétences graphiques sur le logiciel PhotoFiltre.

La plus grosse difficulté de ce projet a été l'organisation du modèle. On s'y est pris à plusieurs fois pour arriver au résultat actuel. C'est principalement une question de génie logiciel, qu'on a résolue sur le papier. L'échelle de ce projet étant nouvelle pour nous, notre méthode de travail à due être totalement revue et c'est là la majeure partie de ce qu'on a appris. Gérer et organiser un gros projet (une centaine de fichiers, 4400 lignes de code) à nécessité beaucoup de réflexion hors des sessions de code.

Il reste cependant beaucoup de points à améliorer ou à compléter. En plus d'approfondir les mécaniques de jeu, il pourrait être très intéressant de rendre ce jeu multijoueur, de le rendre plus beau notamment pour l'affichage des montagnes. L'une des grosses difficultés a été le zoom. Le zoom actuel est fonctionnel, mais le zoom se fait au niveau du centre de l'image. Une amélioration serait un zoom centré sur le curseur. Un point intéressant serait de donner une utilité au SpaceShip. Il existe une infinité de façons d'améliorer ce projet, mais le travail accompli nous satisfait énormément et nous a permis d'améliorer énormément de capacités.