Université de Lille 1 SCIENCES ET TECHNOLOGIES

# Rapport Projet Individuel (101)

# Adaptation PC du jeu de Société

# Full Métal Planète

Tristan Camus & Arnaud Cojez

Encadré par Julien Iguchi-Cartigny

# Table des matières

# Remerciements

Nous remercions le corps enseignant de l'Université Lille 1, d'avoir dispensé leurs cours avec patience et pédagogie.

Nous remercions particulièrement Monsieur Julien Iguchi-Cartigny, tout d'abord d'avoir accepté notre idée de Projet Individuel, ainsi que d'avoir proposé de porter le jeu Full Métal Planète.

Nous le remercions également de nous avoir fait confiance et laissé en autonomie, mais aussi de la disponibilité et de l'aide qu'il nous a apporté, en réponse aux problématiques que nous soulevions lors des suivis hebdomadaires.

Nous remercions les créateurs de Full Métal Planète, Gérard Delfanti, Gérard Mathieu, Pascal Trigaux, ainsi que l'éditeur Ludodélire.

Nous remercions les développeurs des bibliothèques et outils utilisés ; ainsi que les auteurs des livres et tutoriels qui nous ont été d'un grand secours tout le long du projet.

Nous remercions chaleureusement notre testeuse officielle Latifa, ainsi que nos proches et amis, pour leurs nombreux retours et encouragements.

Enfin et surtout, merci à vous, qui lisez ce rapport de projet.

Nous vous souhaitons une bonne lecture !

# Introduction

Tous deux passionnés de jeux vidéo, notre but commun est de travailler dans ce domaine. Déjà avant d'avoir choisi ce sujet, nous développions chacun de notre coté des projets personnels afin de nous familiariser avec les outils et techniques liées au développement de jeux. En effet, le jeu vidéo étant un milieu réputé difficile d'accès, il est important d'avoir de l'expérience et des projets concrets à présenter lors d'entretiens d'embauche afin de mettre toutes les chances de notre coté.

Malheureusement, les projets que nous avions commencés jusqu'alors n'avaient jamais été approfondis faute de temps. Par conséquent, le PJI nous semblait être une excellente occasion d'accumuler un maximum de compétences qui nous seraient utiles pour le jeu vidéo. Ayant déjà expérimenté le développement d'autres types de jeux, nous nous sommes mis d'accord sur le réalisation d'un jeu de stratégie au tour par tour.

Nous avons soumis cette idée et rapidement Julien Iguchi-Cartigny nous proposa d'adapter le jeu de société Full Métal Planète en jeu vidéo. Nous avons trouvé cette idée intéressante, puisque travailler sur l'adaptation d'un jeu existant nous permettait de nous concentrer sur le développement même du jeu en nous évitant la création et l'équilibrage des règles.

Puisqu'il est l'un des langages les plus utilisé dans le développement de jeu, et que nous ne l'avons que peu abordé à l'université, nous avons décidé de réaliser notre projet à l'aide du langage de programmation C++. Il ne nous restait plus qu'à choisir une bibliothèque permettant de gérer la partie "média" (affichage, son, etc.) du jeu. L'un d'entre nous ayant déjà utilisé la bibliothèque multimédia SFML, c'est cette dernière que nous avons choisie.

# Présentation du projet

## Full Métal Planète

Full métal planète est un jeu de société de stratégie au tour par tour édité par Ludodélire en 1988. Il propose aux joueurs, de 2 à 4, d'incarner une société d'exploitation minière et de tenter d'amasser un maximum de minerais sur la Full Métal Planète, une planète riche en minerais mais instable qu'il faudra évacuer avant sa destruction lors du 25ème tour de jeu. Les marées et les autres joueurs viendront rendre cette tâche plus difficile. En effet, en fonction du niveau des marées, à chaque tour de jeu, certaines zones de la carte seront inaccessibles. Les joueurs pourront créer différentes unités terrestres et maritimes afin de récolter les précieux minerais ou encore de s'attaquer entre eux et ainsi ralentir la progression des autres joueurs en détruisant ou en s'emparant de leurs véhicules.

*Photo du jeu de société Full Métal Planète*

# Organisation

## Répartition des tâches

Le but principal du projet fut de réussir à implémenter un maximum de règles lors du temps qui était à notre disposition. Lors des premières semaines du projet nous avons donc tout naturellement commencé par la lecture des règles du jeu, et imaginé les différents concepts qu'il nous faudrait implémenter afin de développer Full Métal Planète. L'une des première décision fut de développer d'un coté une API représentant les données du plateau de jeu et permettant de les manipuler et d'un autre coté une interface se dessinant en fonction des données de l'API.

Bien qu'ayant tous deux commencé en travaillant sur l'API, nous avons vite réparties les tâches en laissant Arnaud continuer le développement de l'API pendant que Tristan s'attelait à la création de l'interface. À partir de là, dès qu'un élément du jeu était ajouté à l'API par Arnaud, Tristan ajoutait l'élément lui correspondant à l'interface. Tristan travailla aussi sur la classe principale du projet : Le contrôleur liant l'API et l'interface, gérant aussi les entrées clavier et les interactions à la souris, plus souvent appelé boucle de jeu dans le domaine du jeu vidéo. Nous reviendrons sur les différents aspects techniques et détails de ces parties du projet plus tard dans ce rapport.

## Emploi du temps

En plus d'une répartition des tâches, nous avons aussi dû organiser notre temps libre afin de travailler sur le PJI. En effet, il nous fallait à la fois garder du temps pour nos TPs et autres activités tout en s'imposant un minimum de travail hebdomadaire sur le projet. Nous avons donc décidé de travailler, à l'université, tous les jeudi intégralement sur le PJI.

Bien souvent, nous travaillions aussi le soir ou le week-end chacun de notre coté sur nos parties respectives en fonction du travail demandé par les autres matières.

# Environnement de travail

## C++

C++ est un langage de programmation orientée objet très répandu, et l'un des langages principalement utilisés dans le milieu du jeu vidéo. Un programme compilé avec ce langage a la particularité de s'exécuter directement sur la machine, sans passer par une machine virtuelle ou un interpréteur. Par conséquent, le programme y gagne en vitesse, mais la programmation en devient plus exigeante.

En effet, nous avons rencontré de nombreux problèmes relatifs à la gestion de la mémoire dont nous parlerons plus tard : dépendance circulaire, destructeurs et pointeurs, etc.

Nous avons aussi appris à nous servir des conteneurs (list, vector, map, etc.) de la STL (*Standard Template Library*) de C++, qui ont tous des applications différentes.

## SFML

Pour gérer l'affichage et la gestion des ressources, notre choix s'est porté vers la bibliothèque SFML (*Simple and Fast Multimedia Library*). Cette bibliothèque est inspirée de la plus célèbre SDL (*Simple DirectMedia Layer*).

À l'instar de la SDL, la SFML est multi-plateforme, nous pouvons donc travailler et lancer le programme sous Windows, OSX et GNU/Linux. Elle permet de gérer non seulement le chargement et l'utilisation de ressources multimédia, des vues/caméras afin de se déplacer dans une scène, mais aussi le réseau. Il est possible d'utiliser OpenGL à partir d'une application conçue en SFML. Mais contrairement à la SDL, elle est conçue pour être plus simple d'accès et orientée objet, sans perdre en efficacité.

## YAML

Se posa ensuite le problème du stockage des informations relatives aux éléments du jeu. Nous voulions stocker les cartes autrement que dans une classe C++. Nous avons donc choisi de stocker les données de la carte en YAML (*YAML Ain't Markup Language*).

YAML est un format de représentation de données qui met l'accent sur la lisibilité, tout en proposant des fonctionnalités proches d'un format tel que JSON. Nous avons utilisé le parser yaml-cpp, nous permettant de lire un fichier .yaml et d'en extraire les données en utilisant des containers STL.

Ce choix nous a permis de retranscrire la carte originale beaucoup plus facilement qu'avec un autre format de données ou en C++.

Voici un exemple d'une carte simple, de 4x4 cases, representée en YAML :

```
name : "terrain1"
cells : [ [[0,0], [0,10], [0,20], [0,30]],
       [[0,1], [0,11], [0,21], [0,31]],
       [[0,2], [0,12], [0,22], [0,32]],
       [[0,3], [0,13], [0,23], [0,33]] ]
```

Une case est représentée par un couple [A,B], A représentant le type de case, et B la zone dans laquelle se trouve la case.

## Catch : Tests

La nécessité d'un framework de test est évidente. Mais nous voulions un framework le plus simple d'utilisation et d'intégration possible. Nous avons fait des recherches et avons trouvé le framework Catch. Ce framework est à la fois :

- simple d'installation : il suffit d'ajouter le fichier catch.hpp dans l'application et de l'inclure dans les fichiers de test.
- simple d'utilisation : un test est représenté comme ceci :

```
TEST_CASE( "stupidTest", "Prove that var equals 2" ) {
    int var = 1;
    REQUIRE( var == 2 );
}
```

Ce test échouera (1 =/= 2) et son résultat sera indiqué à l'exécution des tests.

## CodeLite

Dans un premier temps, nous avons utilisé l'environnement de développement intégré QtCreator. Nous avons ensuite migré vers CodeLite car celui-ci proposait entre autres des fonctions intéressantes, comme la possibilité de trier les fichiers dans des dossiers virtuels, une fonction de reformatage du code, une gestion facilitée des dépendances entre sous-projets (le projet tests nécessite le projet api mais pas le projet interface)

## GitHub

Pour versionner notre code et travailler efficacement, notre choix s'est naturellement porté vers git. Nous avons utilisé GitHub pour créer un dépôt distant. Celui-ci est trouvable à l'adresse suivante : https://github.com/arnaudcoj/pji_full_metal_planete .

# Développement de jeux vidéo

Le développement de jeux vidéo présente beaucoup de spécificités et de notions à comprendre et à intégrer avant de pouvoir correctement le pratiquer. Il faut d'abord bien comprendre les différents mécanismes qui compose un jeu et avoir connaissance des outils et méthodes utilisés pour les mettre en place.

Le développement de jeu vidéo est assez diversifié. Il peut regrouper un grand nombre de compétences intégrant l'utilisation d'images, de sons, de réseaux, de manettes, la mise en place de la logique du jeu ou encore de l'interface. Tout autant de choses que nous n'avons pas forcement étudié à l'université. En effet, nous n'avons réalisé que très peu d'interface et aucun développement multimédia. La réalisation de choses telles que l'animation d'images, leur manipulation ou encore la simple détection de collisions sont des concepts qui nous étaient encore inconnus avant ce PJI et qui parfois sont moins simples qu'ils n'y paraissent si on les réalisent naïvement sans se renseigner sur les bonnes pratiques.

# Projet autonome

## Projet Concret

L'aspect qui nous a sans doute le plus motivé lors de ce PJI était de travailler sur un projet concret. En effet, nous avions eu l'occasion à l'université de travailler sur de nombreux aspects de l'informatique mais sans jamais les lier les uns aux autres. Nous avons avec ce PJI pu mettre en pratique un grand nombre de connaissances dans un projet cohérent qui allait bien plus loin que le simple exercice de travaux pratiques. Ce projet pourrait donc davantage nous valoriser auprès des entreprises et en particulier dans le domaine vidéoludique.

## Se documenter

Ce projet étant un projet autonome et qui présentait un grand nombre de point nouveaux pour nous, notamment liés au développement de jeux vidéo, nous avons dû apprendre énormément de choses par nous même et pour cela, nous documenter. Internet fût notre première source de recherche et les forums de développement nous furent d'une très grande aide.

Le site *redblobgames.com* nous aida également beaucoup. En effet, ce site parlant de développement de jeux vidéo, propose entre autre des explications sur le développement de jeu à grille hexagonal et les algorithmes de pathfinding dont nous parlerons par la suite.

Afin d'avoir des bases plus solides et complètes, nous avons aussi fait l'acquisition de livres liés au développement de jeux vidéo. Arnaud étudia le livre *Game Programming Patterns* de Robert Nystrom et le livre *Thinking in C++* de Bruce Eckel afin de pouvoir mieux travailler sur le développement de l'API. Tristan quant à lui étudia le livre *SFML Essentials* de Milcho G. Milchev afin d'avoir des bases solides pour développer l'interface en utilisant la bibliothèque multimédia SFML.

# Architecture Modèle-Vue-Contrôleur

## Le M.V.C

L'architecture MVC est un concept principalement mis en oeuvre dans la réalisation d'application web. Ce concept n'est cependant pas borné à ce seul domaine.

Ce modèle repose sur la séparation des données (sous-projet api), de la vue (interface), et du contrôle (game).

En effet, dans note projet l'API (le modèle) représente les données du jeu. Celles-ci sont récupérée par la boucle de jeu (le contrôleur) et sont ensuite transmises par ce dernier à l'interface (la vue) qui s'occupera de dessiner les éléments graphiques du jeu en fonction des données qui lui sont communiquées.

Avantages :

- Séparation des tâches et travail d'équipe plus simple
- Permet de changer la méthode d'affichage à tout moment
- Organisation du projet plus claire.

Inconvénients :

- Multiplication du nombre de fichiers
- Complexe à mettre en place
- Peut sembler lourd pour un "petit projet"

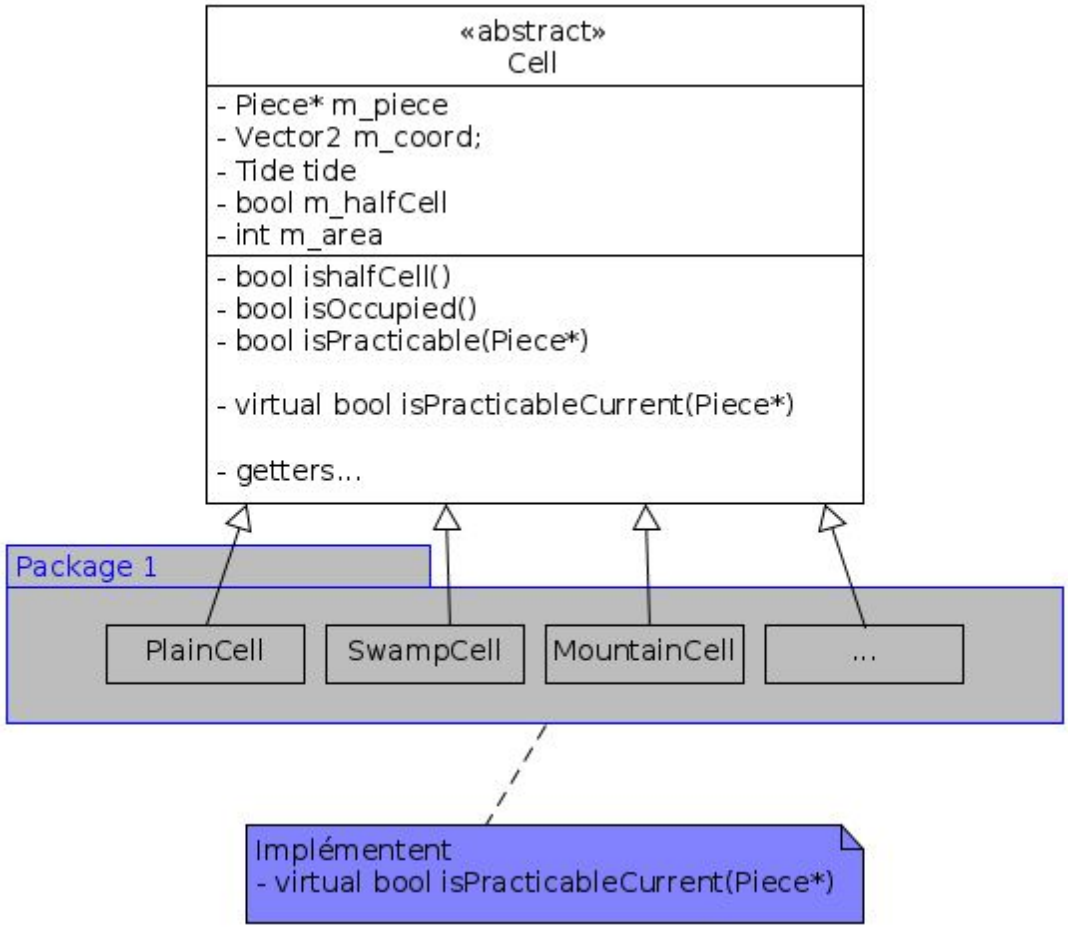
Bien que nous ayons rencontré quelques problèmes par rapport à la philosophie "Orienté Objet", l'utilisation du modèle MVC nous a rendu service. Non seulement au point de vue de l'architecture du programme, car il nous a permis de mieux découper le projet en sous-projets, mais il nous a surtout fait gagner du temps.

En effet, étant mieux organisés et nous mettant d'accord sur les fonctions nous permettant de communiquer les informations entre les couches, nous pouvions travailler chacun de notre côté et mettre le code en commun très simplement. Nous étions par conséquent bien plus efficaces.

# Développement de l'API

## Représentation des éléments du jeu

### Représentation d'une case




Une *Cell* représente une case de la grille.

Elle contient un booléen permettant de savoir si la case est une demi-case ou non. Une demi-case est une case qui chevauche le bord du plateau. Il est possible d'y stocker du minerai mais pas de s'y déplacer.

Ce sont les cases en rouge dans le schéma ci-contre.

Une *Cell* connait ses coordonnées et contient une référence vers le pion qu'elle contient.

Le plateau du jeu est constitué de zones permettant de choisir où faire atterrir l'astronef en début de partie. Ici nous avons implémenté ces zones en utilisant un entier dans chaque *Cell*, représentant le numéro de la zone correspondante.

*bool isPracticable(Piece*)* permet de savoir si une case est praticable par un pion donné ou non, en fonction de la marée si la case est une case Marais ou Récifs.
Nous conservons par conséquent la marée courante dans la *Cell*, afin de savoir si la case est submergée ou non et donc de décider si la case est praticable.

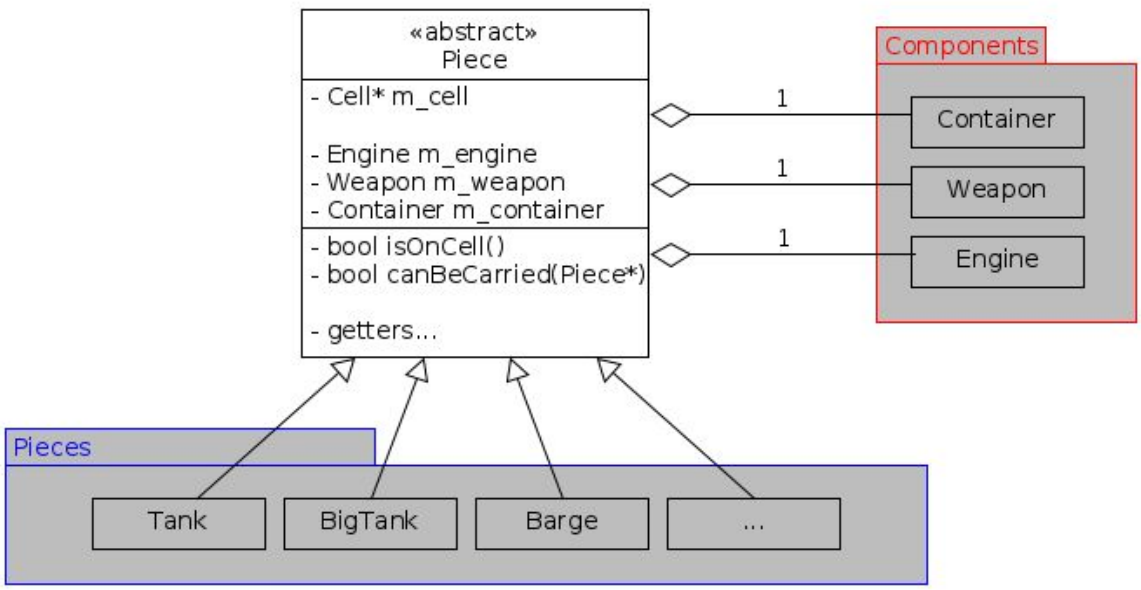
Les cases Plaines sont praticables par les unités terrestres, les cases de montagnes ne sont pas praticables par les Tanks Lourds. La mer n'est praticable que par les unités marines. Les Marais et Récifs changent d'état en fonction des marées. Voir ci-dessous.

### Représentation d'un pion

Les pions du jeu sont représentés par la classe *Piece*. Cette classe contient une référence vers la *Cell* sur laquelle le pion est posé.

La particularité de cette classe est d'utiliser un système de Components qui permettent de diversifier simplement le comportement des unités.
Ce principe sera abordé plus loin dans le rapport.

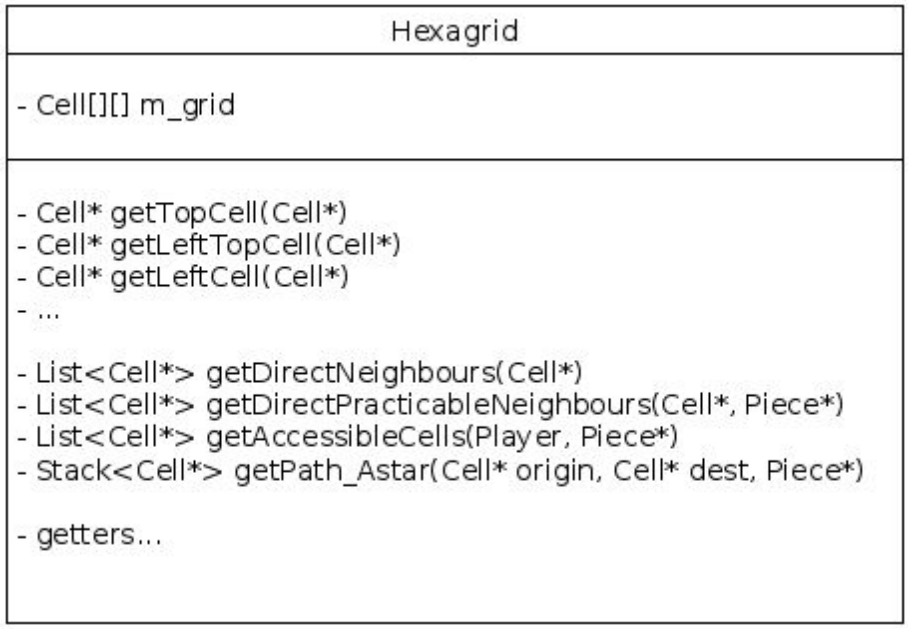
Les différentes unités sont le Tank, le Tank Lourd, la Vedette, le Crabe, la Pondeuse Météo, la Barge, le Ponton et l'Astronef. Nous ne pouvons malheureusement pas nous attarder sur la fonction de ces pièces, mais les règles du jeu sont facilement trouvables sur le Web.



## Représentation de la grille

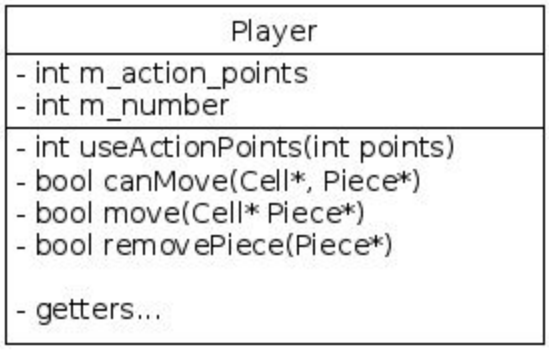
Cette grille est composée de cases hexagonales. Nous l'avons représentée en mémoire à l'aide d'une tableau en 2 dimensions de *Cell*.

La classe *Hexagrid* est dotée de méthodes permettant de renvoyer un pointeur de *Cell* a partir de coordonnées, d'obtenir les cases voisines (praticables ou non) d'une *Cell*, et de calculer le plus court chemin entre 2 cases. Nous reviendrons sur l'implémentation de ces algorithmes dans la partie suivante.




## Représentation d'un joueur

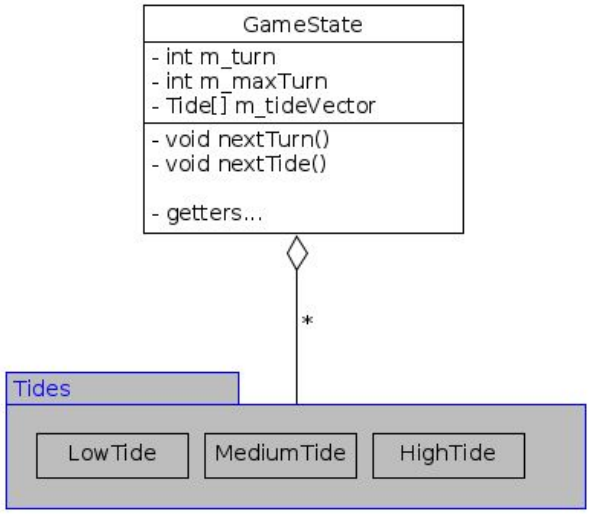
Un *Player* possède un entier représentant le nombre de points d'actions restant et un entier servant d'identifiant. C'est cette classe qui permet l'interaction entre cases et les pions.



## Représentation de l'état du jeu

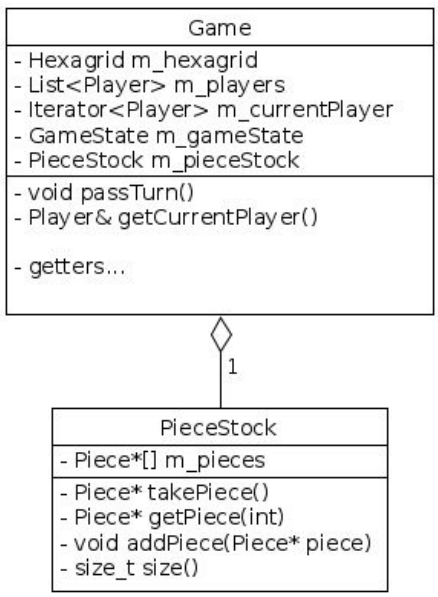
Le *GameState* contient un entier représentant le nombre de tours maximum, et un autre représentant le tour actuel.

Il contient un vecteur de Marées, ou *Tide* (*Low, Medium, High*) qui sera mis à jour à chaque tour à partir du tour 3, avec la même méthode que pour le jeu de plateau. Il contient par conséquent des méthodes permettant de mettre à jour le tour et la marée.




## Représentation du jeu

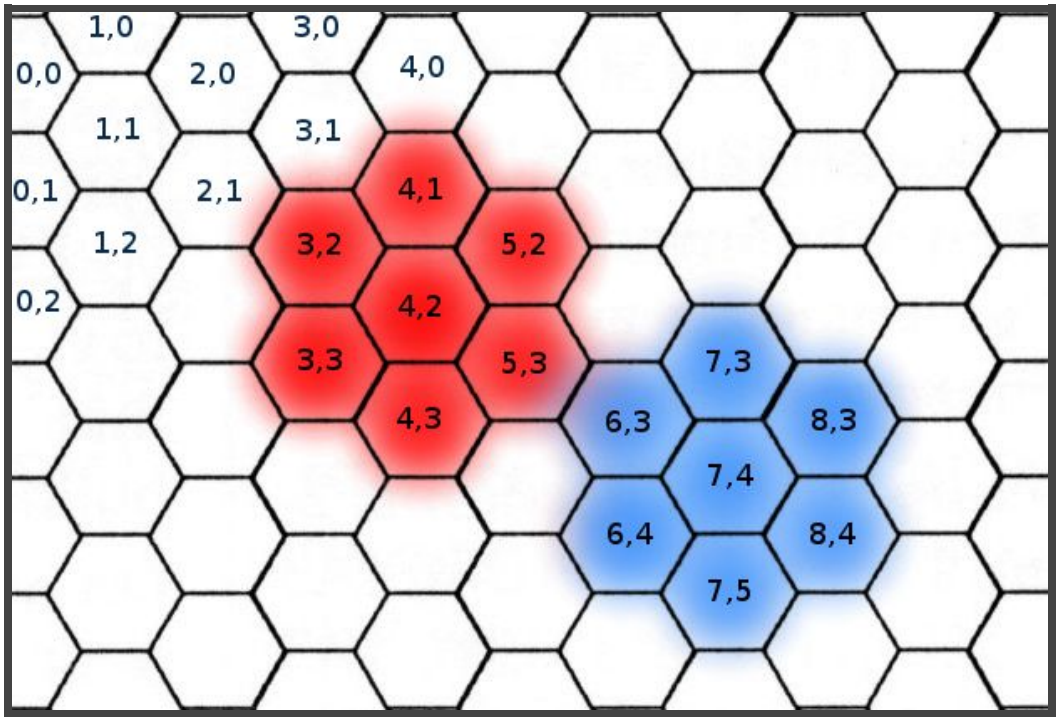
Le *Game* contient une liste de *Players* et de *Pieces*, l'*Hexagrid* et le *GameState*. Il permet de récupérer ces éléments, le joueur courant et de passer les tours.



## Interaction entre les éléments du jeu

### Agencement de la grille

Il se trouve qu'une grille possédant des cases hexagonales est suffisamment régulière pour que l'on puisse la représenter sous forme de tableau à 2 dimensions.

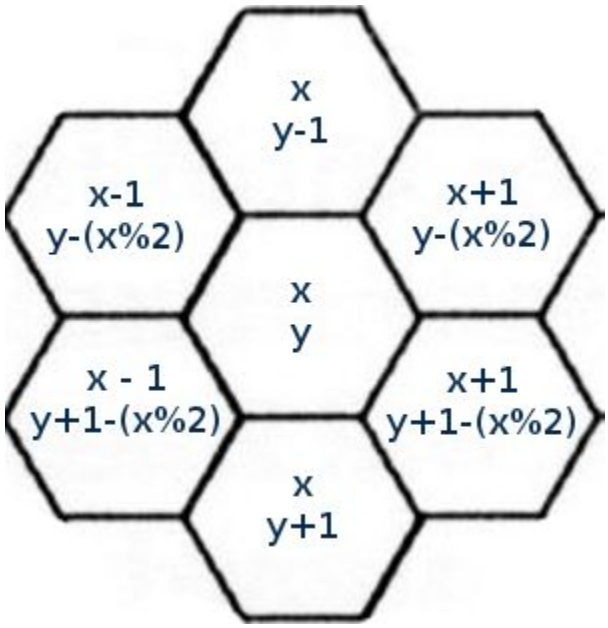



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0,0 | 1,0 | 2,0 | 3,0 | 4,0 | 5,0 | | | | | |
| 0,1 | 1,1 | 2,1 | 3,1 | 4,1 | | | | | | |
| 0,2 | 1,2 | | 3,2 | 4,2 | 5,2 | | | | | |
| | | | 3,3 | 4,3 | 5,3 | 6,3 | 7,3 | 8,3 | | |
| | | | | | | 6,4 | 7,4 | 8,4 | | |
| | | | | | | | 7,5 | | | |

Nous savons donc stocker une grille de ce type, mais nous ne savons pas encore comment l'utiliser. En effet, nous pouvons directement récupérer une case sans problème à l'aide de ses coordonnées, cependant les cases hexagonales ont 6 voisins alors que les cases d'un tableau n'en ont que 4. Cela pose problème car certains algorithme utilisés plus tard (notamment la recherche de chemins) ont besoin de connaître les voisins d'une case.

Nous avons constaté que les voisines d'une case dont le x est pair sont organisées d'une certaine façon (cases rouges sur le schéma), alors que quand le x est impair, elles sont organisées symétriquement par rapport à l'axe des x (cases bleues).

Nous pouvons donc trouver les calculs permettant d'obtenir les coordonnées de la case voulue et fusionner ces deux cas en un seul grâce à un simple modulo 2.



## Interactions entre Cases et Pions

Dans un premier temps, nous voulions faire en sorte qu'une case contienne un pion et qu'un pion possède un attribut case. Nous avons rencontré un problème de dépendances circulaires. Une dépendance circulaire apparaît quand une classe A contient un attribut d'une classe B, qui contient elle-même un attribut de classe A.
Par conséquent, A doit créer un objet de type B, qui doit lui aussi créer un A, etc.

Pour régler ce problème, nous avons dans un premier temps utilisé des pointeurs classiques. Les objets n'étant plus directement stockés, la dépendance circulaire s'efface et laisse place… À une inclusion circulaire.

Le problème ici est que le compilateur, pour créer B, a besoin de savoir que la classe A existe, même s'il n'a pas besoin de son implémentation.

Pour résoudre ce problème, on va donc utiliser une déclaration avancée (forward declaration). L'idée est de déclarer, avant la définition de B, que A existe. Ainsi B pourra être défini sachant que A existe, puis A sera définie ensuite. Pour résumer, nous allons dire au compilateur "Il va y avoir une classe A, elle sera définie plus tard mais elle existe vraiment."

Ce problème étant réglé, nous sommes tombés sur un nouveau problème. Les objets *Cell* et *Piece* n'étaient pas détruits. Cela était dû au fait qu'une Piece et une *Cell* se référencent mutuellement. Par conséquent, lorsque l'on veut détruire A, A cherche a détruire B, donc B cherche à détruire A, etc. Nous pouvons résoudre ce problème en indiquant les niveaux d'appartenance des pointeurs.

En C++11, nous avons à notre disposition des *smart-pointers*. Ceux-ci permettent de ne plus avoir à gérer la suppression des objets liés aux pointeurs soi-même. Les deux types de pointeurs qui nous intéressent sont les *shared_ptr* et les *weak_ptr*.

Les *shared_ptr* agissent comme des pointeurs classiques, mais à chaque fois que l'on pointera un objet, un compteur de références sera incrémenté. Il sera décrementé quand on détruira le pointeur. Lorsque le compteur retombe à 0, on considère que l'objet en question n'est plus utilisé et on le détruit.
Les *weak_ptr* représentent un lien plus faible. Ils référencent un objet mais ne le "possèdent" pas. Cela signifie qu'ils n'incrémentent pas le compteur de référence.

On considèrera donc que la *Cell* possède une *Piece* (utilisation de *shared_ptr*), et qu'une *Piece* fait juste réference à une *Cell*.

Par conséquent, quand le compteur de références d'une *Cell* tombe à 0, son *shared_ptr* vers la *Piece* est détruit. Le compteur de la *Piece* en question tombe donc à 0. La *Piece* est détruite. Comme son pointeur est faible, il n'a pas d'influence sur la *Cell*. Les deux objets sont bien détruits.

## Piece et components

Comme nous l'avons vu précédemment, les *Piece*s sont constituées de *Components*.

Les *Components* permettent de spécialiser le comportement d'un objet, en donnant à l'objet en questions des *Components* avec un comportement spécifique. L'avantage de ce patron de conception est que nous pouvons considérer la classe *Piece* en tant que container de *Components*. Cela permet d'éviter la création d'arbres d'héritages trop complexes, sachant que l'on peut soit choisir de faire des héritages selon le mode de déplacement d'une *Piece*, ou alors sur le type d'armements, etc.

Ici, on peut créer une classe uniquement en spécifiant le type de *Component* utilisé, ce qui simplifie l'implémentation et la compréhension du programme.

Nous avons donc 4 types de moteurs (*Engine*) :

- Terrestre
- Terrestre Lourd
- Maritime
- Aucun Moteur

3 types d'armes (*Weapon*) :

- Classique
- Longue portée
- Aucune Arme

et 4 types de transporteurs (*Container*) :

- Container Classique
- Grand Container
- Container de Minerai
- Aucun Container

Donc par exemple, voici comment on composera un Tank :

- *Engine* : Terrestre
- *Weapon* : Classique
- *Container* : Aucun

Une Barge :

- *Engine* : Maritime
- *Weapon* : Aucune
- *Container* : Grand Container

Ou une pondeuse météo :

- *Engine* : Terrestre
- *Weapon* : Aucune
- *Container* : Container de Minerai

etc.

On constate qu'il aurait été beaucoup plus complexe de définir tous ces comportement sans ce patron de conception.

### Player, contrôleur gérant les actions d'un joueur

Comme dit plus haut, la classe *Player* permet d'agir sur une case et sur un pion donné, *Player* suit donc le design pattern *Controler*. Il offre les méthodes *move* et *removePiece*.

La méthode *move*, comme son nom l'indique, permet de bouger une pièce sur une case en fonction de son accessibilité. *removePiece* permet de retirer une pièce d'une case et de la remettre dans le stock du joueur.

### Game, contrôleur gérant la logique du jeu

Alors que *Player* gère les actions relatives aux joueurs, *Game* permet de contrôler l'évolution d'une partie. Il permet de passer au joueur suivant, puis une fois que tous les joueurs aient joué une fois, de changer de tour. En fonction du tour, *Game* donnera plus ou moins de points d'actions aux joueurs, demandera au *GameState* d'incrémenter le compteur de tour, et éventuellement de changer de marée.

C'est donc à partir de *Game* que l'on pourra permettre le déroulement du jeu. Soit en récupérant le *Player* courant pour l'utiliser comme contrôleur, soit en demandant directement au *Game* d'agir sur l'état du jeu.

### Effets des marées et des tours sur le jeu

Le premier tour de jeu correspond à l'atterrissage des aéronefs. Le second tour au déploiement des unités. Ces deux tours ne sont pas implémentés pour l'instant, les unités étant déployées dès le début du jeu.

À partir du troisième tour, les marées changeront à chaque tour et des points d'action seront donnés aux joueurs (tour 3 = 5 points, tour 4 = 10 points, tour 5 et suite = 15 points)

Il existe 3 types de marées : *Low*, *Medium* et *High*. Ces marées agissent sur le terrain en rendant certaines cases impraticables pour telle classe de pions.

Tableau représentant les unités pouvant circuler sur les cases sujettes à la marée :

| | Marée Basse | Marée Normale | Marée Haute |
|---|---|---|---|
| Marais | Unités Terrestres | Unités Terrestres | Unités Marines |
| Récif | Unités Terrestres | Unités Marines | Unités Marines |

À chaque changement de tour, *Game* demandera à *Hexagrid* de mettre à jour les cases avec la *Tide* courante. Ainsi, les cases pourront connaître leur état, et pourront indiquer quelle texture afficher.

## Intelligence Artificielle

### Choix des marées

La fonction *initTideVector()* de *GameState* a pour but de créer une liste de *Tide* représentant une pioche de cartes marées. On enlèvera la marée en tête de liste à chaque changement de tour. Lorsque la liste sera vide, on appellera à nouveau *initTideVector*.

La pioche est crée de cette façon :

- on remplit la liste de 5 marées de chaque type. Dans le jeu de plateau, cela correspond au nombre de cartes marées disponibles ;
- on exécute un *shuffle* sur cette liste afin de reproduire le mélange des cartes ;
- finalement, on enlève les 3 marées en têtes. Cela correspond au fait que dans le jeu xxqclassique, il faut retirer 3 cartes au hasard.

De cette façon, les marées auront les mêmes chances de sortir que dans le jeu de plateau.

### Vérification de praticabilité des cases par les pions

Afin de vérifier si un pion peut bouger sur une case, la classe *Player* possède une méthode *canMove* qui prend une *Piece* et une *Cell* en paramètre.

Son exécution se déroule comme suit :

- dans un premier temps, nous vérifions que la case cible n'est pas déjà occupée ;
- ensuite, nous confirmons que la case cible est praticable par la *Piece* en question.

Nous regardons si la *Piece* est déjà sur une case. Le cas échéant, nous contrôlons le fait que la case de départ soit praticable aussi. En effet, si la *Piece* est embourbée sur sa case de départ, elle ne peut pas se déplacer.

Afin de savoir si une *Cell* est praticable par une *Piece*, nous appelons la méthode *isPracticable* de la classe *Cell*. Cette méthode prend une *Piece* en paramètre.

Cette méthode vérifie dans un premier temps si la *Cell* est une demi-case, puis appelle une méthode virtuelle propre à chaque type de *Cell* : *isPracticableCurrent*.

Cette dernière récupère l'*Engine* de la *Piece*, et signale que la case est praticable, ou non, en fonction du type de case, et éventuellement de la marée.

### Connaître les voisins accessibles d'une Cell

La méthode *getDirectPracticableNeighbours* est très basique. On récupère les *Cell* voisines de celle donnée en paramètre en appelant toutes les méthodes *get{Top/TopLeft/etc.}Cell()*.

Si la *Cell* est praticable par la *Piece* donnée elle aussi en paramètre, nous l'ajoutons à la liste des voisines. Puis nous retournons cette liste à la fin de la méthode.

### Connaître toutes les cases accessibles à partir d'une Cell

La méthode nommée *getAccessibleCells* nous permet de récupérer l'ensemble des *Cell* accessibles par une *Piece* à partir de sa *Cell* d'origine. Elle prend en compte les points d'action et la praticabilité des *Cell*. Elle sera utilisée entre autres, pour afficher les déplacements possibles d'une unité dans l'interface.

L'implémentation de cette méthode suit celle de l'algorithme de remplissage par diffusion (ou flood-fill).

L'algorithme se déroule comme suit :

On dispose de 2 "franges" qui contiennent les *Cell* mises en attente d'être traitées par l'algorithme. Une frange représente les *Cell* pouvant être accédées avec *k* points d'actions.

Nous utilisons aussi un ensemble, simplement nommée "*cells*", qui contient les *Cell* considérées comme accessibles. L'avantage d'utiliser un ensemble est la garantie qu'un même élément ne pourra apparaître plus d'une fois.

Le principe est de parcourir la frange avant. Pour chaque *Cell* de cette frange, on va récupérer les cases voisines praticables grâce à *getDirectPracticableNeighbours*. On va ensuite les ajouter dans l'ensemble *cells*, et dans la frange arrière. Une fois cette frange parcourue, on *décremente* les points d'action et fait passer la frange arrière vers l'avant. On recrée une frange arrière et réitère l'opération, jusqu'à épuisement des points d'actions.

On renvoie finalement l'ensemble cells, qui contient désormais les cases pouvant être parcourues par la *Piece* d'origine.

### Connaître le plus court chemin entre deux Cells

Cette méthode *getPath_Astar* permet d'avoir le chemin le plus court possible entre deux cases, en fonction de la mobilité des *Piece*.

Nous avions le choix entre plusieurs algorithmes de recherche de chemin. Un des plus célèbres étant l'algorithme de Dijkstra.

Cet algorithme permet bien de trouver le plus court chemin, mais étant un parcours en largeur, il fait une recherche circulaire avec un rayon de plus en plus grand. Ce qui fait que l'on perd en efficacité, en particulier car nous avons une intuition de la direction à prendre, vu que nous savons où se trouve la case d'arrivée par rapport à la case de départ.

L'algorithme A* résout ce problème en utilisant une heuristique, ici très simple. On choisit la case voisine vers laquelle se déplacer en calculant sa distance "à vol d'oiseau". Nous nous dirigerons donc simplement vers la *Cell* destination jusqu'à rencontrer un obstacle. À partir de ce moment, on va faire un parcours en largeur afin de contourner l'obstacle et nous diriger de nouveau grâce à l'heuristique.

Il nous faudrait un chapitre entier pour expliquer les motivations et le déroulement de l'algorithme A*, donc nous n'entrerons pas dans le détail. De nombreuses ressources sont disponibles en ligne, telles que le site [redblobgames](redblobgames) qui introduit très bien le concept.


*Capture d'écran du jeu. Les cases sombres sont les cases inaccessibles, et les plus claires représentent le chemin à parcourir pour passer d'une case à une autre.*

# Contrôleur

## Boucle de jeu

L'élément le plus important d'un jeu vidéo est la boucle de jeu. En effet, c'est cette boucle de jeu qui va contrôler l'ensemble des éléments du jeu. Dans le cadre de notre MVC, c'est la boucle de jeu qui fera donc office de contrôleur.

La boucle de jeu se divise en 3 grandes partie. Une première partie qui permet de gérer les inputs : les actions du clavier et de la souris. Une deuxième partie qui permet de mettre à jour les différents éléments qui compose le jeu et une troisième partie qui dessine ces éléments.

La première partie gérera, de manière plus générale, les évènements. Ceux-ci déclencheront alors des modifications de l'état des objets.

Les différentes méthodes de mise à jour des objets, les fonctions update(), s'occuperont alors d'effectuer toutes les modifications à prendre en compte à chaque tour de boucle sur les objets du jeu.

La méthodes draw de chaque entité sera alors appelée et dessinera les objets en prenant en compte leur état.  À chaque tour de la boucle de jeu, la fenêtre du jeu est effacer pour être redessinée à la fin.

Ainsi, il est possible grâce à la boucle de jeu d'interagir avec les objets, de les faire évoluer dans le temps et en fonctions des interactions et de les afficher correctement à l'écran.

Exemple de boucle de jeu simple :

```
while (window.isOpen())
{
    //Handle events
    sf::Event event;
    while (window.pollEvent(event)) {
        if (event.type == sf::Event::EventType::Closed)
               window.close();
    }

    //Update scene
    objects.uptdate();

    //Render cycle
    window.clear();
    objects.draw();
    window.display();
}
```

# Interface

## Représentation des éléments du jeu

A chacune des classes de l'API ayant besoin d'un rendu graphique correspond une classe de l'interface permettant de la dessiner.

Chacune de ces classes présente au minimum deux fonctions permettant son rendu. Un méthode update(), appelée à chaque tour de la boucle de jeu, ainsi qu'une méthode draw(), appelé à la suite des update(). C'est dans la méthode update() que seront modifiés toutes les propriétés graphiques d'une entité tels que sa position, sa taille ou encore son angle de rotation. La méthode update() transmettra aussi, si besoin, les instructions aux données de l'API pour qu'elles se modifient.

La méthode draw() appelée suite à une mise à jour d'entité dessinera alors tout simplement l'entité en fonction de son état au moment de l'appel.

Partons d'un exemple concret :

Lorsqu'un véhicule se déplace, la méthode update() vient simplement à chaque tour de la boucle de jeu déplacer l'image du véhicule de quelques pixels dans la direction du déplacement. À la fin du déplacement elle notifie alors l'API du déplacement du véhicule et les données du plateaux sont alors modifiée. Bien entendu entre chaque appel à la méthode update() la méthode draw() est appelée et affiche le véhicule à sa nouvelle position. C'est ce qui donne l'impression d'un véritable déplacement du véhicule.

## AssetManager

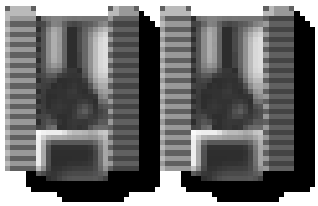
Un jeu vidéo manipule un grand nombre de ressources aussi appelés assets. Ces assets sont, entre autres, les images, les sons, les musiques et les polices d'écritures utilisés par le jeu. Une erreur à éviter est d'aller chercher directement depuis leur fichiers les ressources à chaque fois que l'on en a besoin dans le déroulement de l'application. En effet, le chargement des assets est une opération coûteuse. C'est pour cela qu'il nous a fallu mettre en place un AssetManager afin de gérer correctement les ressources. Celui-ci chargeant et gardant une fois pour toutes les assets en mémoire et évitant ainsi d'avoir à les recharger à chaque tour de boucle. L'AssetManager gardera les objets des ressources en mémoire et une référence vers ces objets permettra de les utiliser.

## Animator

Afin de pouvoir animer les éléments graphiques du jeu, nous avons mis en place une classe Animator.

Les images représentant des éléments dans un jeu sont appelé des sprites. Une entité telle qu'un minerai est un objet immobile et ne nécessite pas d'être animée. Un simple sprite suffit donc à le représenter. Un objet tel qu'un tank, qui est capable de mouvement, doit disposer d'animation afin de rendre le jeu plus agréable. Pour réaliser des animations, on utilise généralement dans la programmation de jeux 2D des spritesheets qui sont des images contenant un ensemble de sprites qui, affichés les uns après les autres permettent de rendre l'animation.

Sprite du minerai. Image représentant une entité immobile n'ayant pas besoin d'animation



*Spritesheet du tank. Image regroupant plusieurs sous images qui affichées les unes à la suite des autres réalisent l'animation du tank.*

L'Animator nous a permis de créer des animations en indiquant pour chaque élément animable, quels sprites, contenue dans une spritesheet, seraient utilisés pour rendre son animation et à quelle vitesse ils défileraient. Pour pouvoir afficher un unique sprite et pas la spritesheet complète, il nous faut connaître la largeur et la hauteur d'un sprite puis afficher seulement la partie de l'image correspondant au sprite qui nous intéresse. La SFML nous a beaucoup aidé dans la conception de cette fonctionnalité de l'Animator puisqu'elle met a disposition une fonction permettant d'afficher une partie précise d'image et ainsi de n'afficher qu'un sprite d'une spritesheet.

## Manipulation de la caméra

La carte du jeu étant relativement grande, il nous fallait, afin d'afficher toutes les entités à l'écran en même temps, les représenter très petites. Elles devenaient alors très difficiles à discerner. Nous avons donc dû mettre en place une caméra afin de pouvoir zoomer et dézoomer sur la carte et se déplacer sur celle-ci.

La SFML nous fut encore très utile puisqu'elle met à notre disposition un objet View permettant de gérer facilement une caméra en n'affichant qu'une partie de la carte avec la posibilité de zoomer et dezoomer.

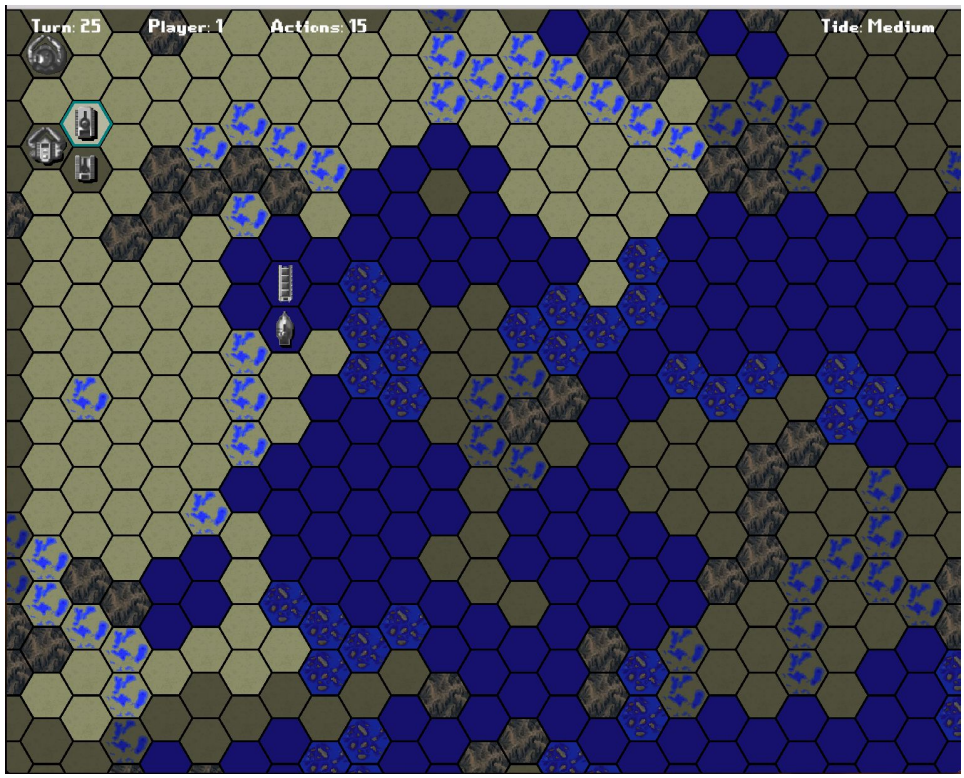
Le déplacement d'une caméra 2D est aussi appelé scrolling. Afin de réaliser celui-ci, il nous a simplement fallu déplacer la View en fonction des déplacement de la souris dans la fenètre de jeu. Nous avons du tester différents comportements afin de trouver une manière qui nous semble agréable de déplacer la caméra à la souris.

## Feedbacks visuels

Nous avons ajoutés d'autres éléments visuels afin de rendre la manipulation du jeu plus agréable et intuitive. En effet, nous avons ajoutés de nombreux feedbacks visuels afin de mettre en avant les éléments importants des interactions.

Voici quelques exemples de ces feedbacks :

*Coloration de la case du véhicule sélectionné*




*Assombrissement des cases non accessible lors des déplacements*

*Éclaircissement des cases du pathfinding*

# Conclusion

Étant passionnés de jeux vidéo, nous souhaiterions un jour être capable de créer nos propres jeux. Pour cela, il nous faut tout d'abord étudier les aspects techniques de la création vidéoludique. En effet, la création d'un jeu vidéo touche un grand nombre de domaines et nécessite une réelle maîtrise de ceux-ci afin de pouvoir être correctement pratiquée aussi bien du coté du développement que de celui de la création des règles et du design.

C'est donc cet aspect qui était au coeur de notre Projet Individuel. En développant cette adaptation de Full Métal Planète, nous avons eu l'occasion de découvrir plusieurs pratiques liées à la programmation de jeu vidéo. Ces pratiques nous serviront à coup sûr à entrer plus facilement dans le monde de la production vidéoludique.

Nous n'avons pas pu implémenter toutes les règles du jeu. Cependant, étant conscients que nous ne pourrions pas livrer un produit fini, nous avons décidé de nous concentrer sur certains éléments qui nous paraissaient importants. Par conséquent, nous avons implémenté toute la logique de déplacement et de marées, les terrains et la spécialisation des unités. Nous avons également travaillé sur la représentation des entités du jeu, les animations et retours visuels, ainsi que la manipulation de la caméra.

En plus de ces concepts, nous avons pu approfondir nos connaissances sur le langage C++, et découvrir la bibliothèque SFML. Nous avons également travaillé sur la communication, l'organisation et la répartition des tâches afin d'être les plus productifs possibles. De plus, nous avons mis en place un maximum de tests unitaires, nous permettant de vérifier le bon fonctionnement du programme.

Nous essayions de travailler un maximum en autonomie, mais nous tenions à faire part de nos réflexions à notre responsable de PJI, qui a su nous guider tout au long du projet.

Fort de cette expérience, nous avons gagné en confiance et en connaissances, et sommes encore plus motivés à travailler dans le domaine du jeu vidéo. Désormais, nous pensons qu'il pourrait être intéressant, lors d'un futur projet, d'utiliser un moteur de jeu, afin de nous concentrer sur la conception et le développement de règles et de mécaniques de jeu.