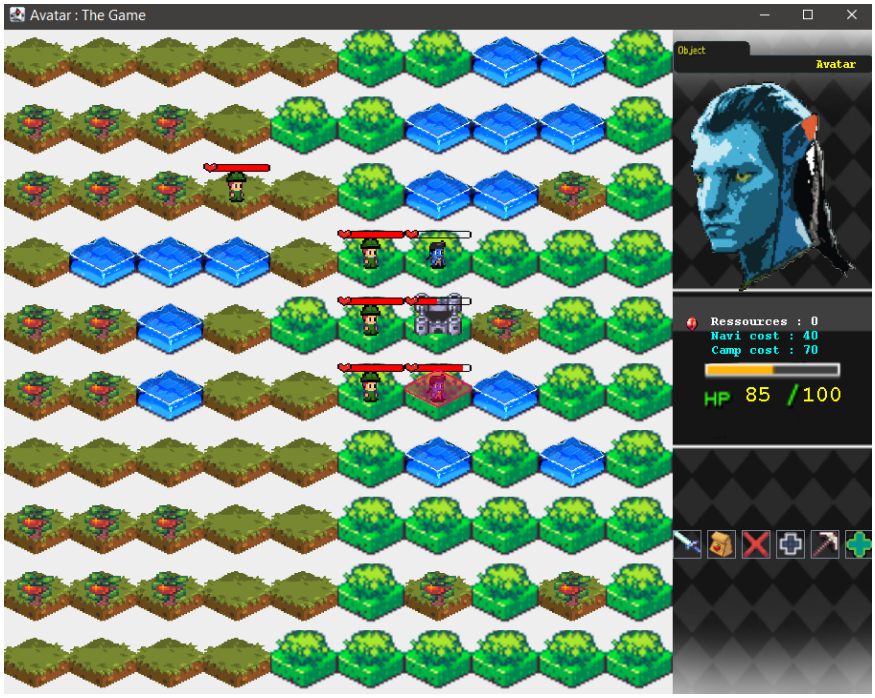
Rapport de PCII : ”Avatar”

Par Lola Austin, Jiren Ren, Seryozha Hakobyan, Pierre Derathe

Introduction :



Dans ce projet, nous présentons un jeu codé en Java basé sur le film "Avatar". Il consiste en une carte en 2D dans laquelle le joueur joue le rôle d'un avatar. Son objectif principal est de survivre et protéger l’arbre sacré de sa planète, pour se faire il lui faudra se battre contre des ennemis venus de l'extérieur, collecter des ressources, ainsi qu’agrandir son clan pour avoir un maximum d’alliés.

Le jeu se déroule dans un monde fictif où l'arbre sacré est considéré comme la source de vie de tous les habitants. Le joueur doit placer stratégiquement son avatar sur la carte et se déplacer pour affronter l'ennemi. Aidé par plusieurs alliés, le joueur doit éliminer tous les ennemis avant qu'ils n'atteignent l'arbre sacré, sinon la partie sera terminée. Le jeu est conçu pour être stimulant et captivant, offrant des heures de divertissement aux joueurs.

Lors du développement, l’idée de l’arbre sacré a été abandonnée et remplacée par la protection des camps.

Ce rapport détaille le processus de développement du jeu, de sa conception à sa mise en œuvre finale. Nous discuterons de nos choix, des décisions prises en matière de conception et des défis rencontrés au cours du développement. Enfin, nous évaluerons la version finale de notre jeu et suggérerons des améliorations possibles.

Analyse globale :

Le jeu se déroule dans un monde animé et coloré représenté par une carte en 2d composée de cases, peuplée de différents types d'ennemis et de na'vis. Les joueurs incarnent un na'vi chargé de défendre les camps contre des vagues d'ennemis. Le jeu comporte plusieurs niveaux, chacun avec sa propre difficulté.

La carte est composée de cases prédéterminées. Les cases peuvent être de l'eau, des champs, de la forêt ou des baies. L'eau est inaccessible et les na'vi peuvent récolter de la nourriture sur les baies.

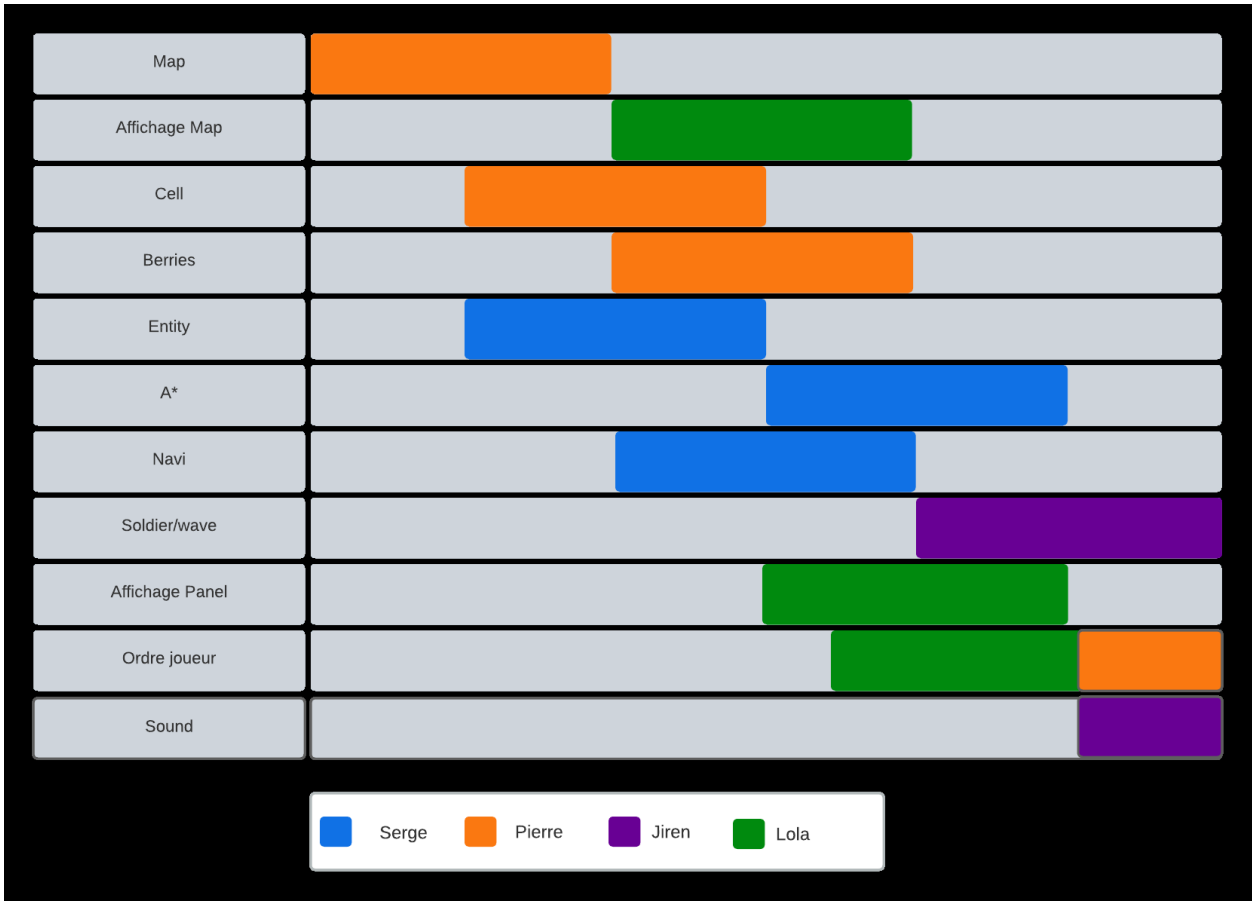
Les na'vi sont des entités contrôlables qui peuvent bouger, attaquer, récolter de la nourriture et construire un camp. Les camps peuvent produire des na'vi. Les soldats sont des entités non-contrôlables qui peuvent bouger et attaquer, et de nouveaux soldats apparaissent avec le temps. Si le nombre de soldats passe deux fois de nombre de na'vi, il va s'arrêter d'apparaître nouveau soldat. Toutes les entités peuvent mourir.

Les baies est en quelque sorte la monnaie du jeu, elles peuvent être utilisées pour effectuer diverses actions telles que l'amélioration des avatars ou la création de nouveaux camps entre autres.

Le joueur a le choix entre plusieurs types de tâches, contrôlées par le tableau de contrôle positionné à droite de la carte. En effet selon le personnage sélectionné, une image et une description de celui-ci s'affiche accompagnée de différentes actions. Le joueur doit donc placer ses avatars de façon stratégique pour bloquer le chemin de l'ennemi et l'empêcher d'atteindre l'arbre sacré.

Les plus difficiles fonctionnalités sont AStar,Affichage et Ordre de joueur dans le contrôle package. Comparé aux trois, Entity, Terrain et Map sont un peu plus faciles. Cependant, les trois dernières fonctionnalités sont prioritaires, sans elles, la structure n'existe pas et rien n'est possible.

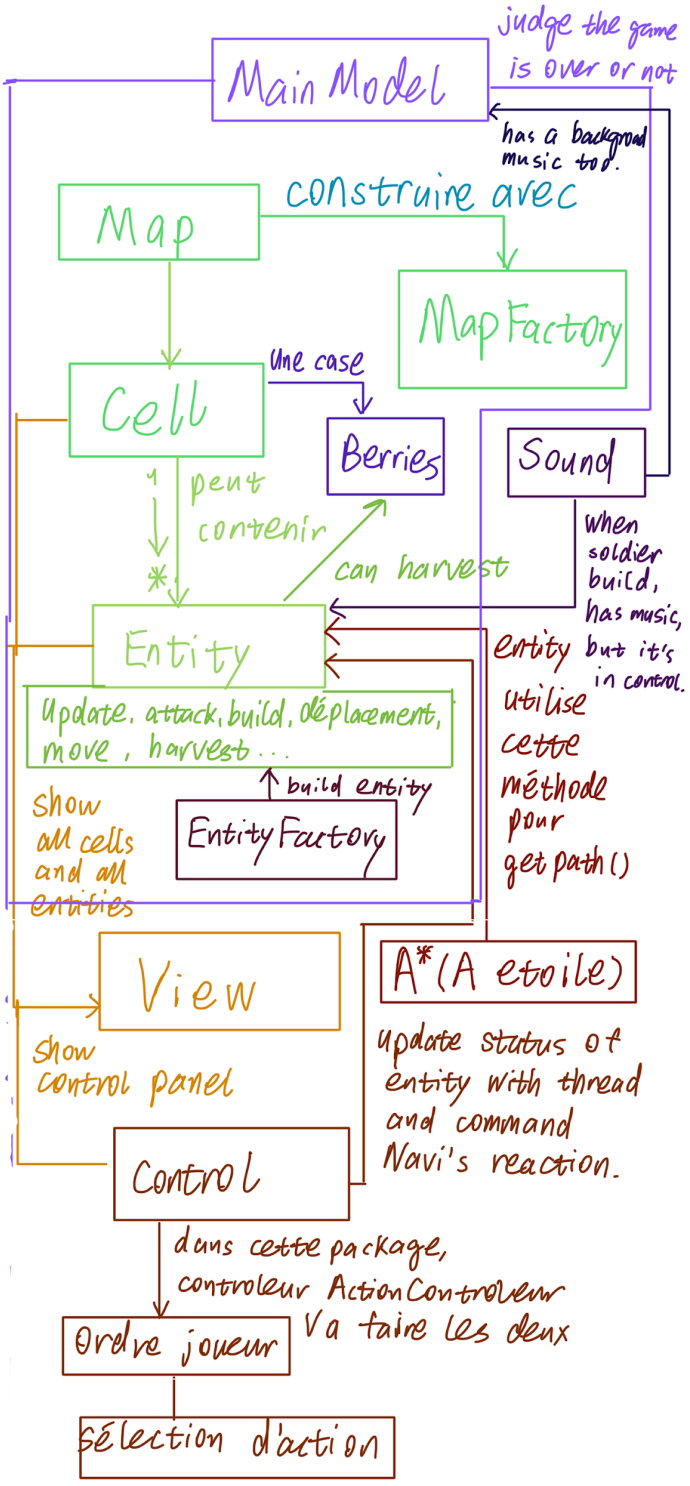
Plan de développement :



Conception générale :

Le projet suit le modèle MVC.

Un diagramme de classe complet est dans les fichier


judge the game is over or not
Main Model
has a background music too.
Map
construire avec
Map Factory
Cell
Une case
Berries
Sound
peut contenir
can harvest
when soldier build, has music, but it's in control.
Entity
entity utilise cette méthode pour get path()
Update, attack, build, déplacement, move, harvest...
build entity
Entity Factory
show all cells and all entities
View
A*(A etoile)
show control panel
update status of entity with thread and command Navi's reaction.
Control
dans cette package, controleur ActionControleur Va faire les deux
Ordre joueur
Sélection d'action

Conception détaillée :

Map (Grille)

La grille est un tableau de case de dimension 2 de taille sizeGrid. On la construit avec MapFactory, une classe basée sur les patterns singleton et factory. On lui donne le nom d'une carte et il la renvoie en la construisant à partir d'un tableau de string.

Cell (Case)

Une case peut contenir une entité au plus. Elle peut enlever ou ajouter l'entité de la case. Elle doit avoir un type de terrain :

- Water
- Field
- Forest
- Berries

Parmi tous ces types de cases, seule "Water" est inaccessible aux entités, les autres leur sont accessibles et ont le même coup pour traverser.

Dans la suite, une case sera considérée comme *accessible* si elle n'est pas de type "Water" et s'il n'y a personne dessus.

Une case est créée par CellFactory à partir de sa position et son type.

Berries(baies)

Il est important de noter que "Berries" fleurit continuellement, donc a de plus en plus de ressources à offrir sans dépasser MAX_FOOD, pourtant, si la quantité de ressources tombent à 0, la case devient un simple Field.

Pour gérer cela, ControlBerries a accès à toutes les cases berries et toutes les secondes appelle update() sur chaque Berries. update() ajoute REGEN à la nourriture de la case sans dépasser MAX_FOOD et si la case est épuisée alors elle est retirée de ControlBerries.

Entity (Entité)

Une entité peut être un soldat ennemi, un Na'vi (pion ami) ou même un camp de Na'vi a vitesse nulle. Les trois classes `Navi`, `Soldier` et `Camp` héritent de la classe mère `Entity` et implémentent `IEntity`.

La classe `Entity` regroupe les fonctionnalités dont toutes les entités ont besoin.

`Update` :
A chaque laps de temps, le contrôleur demandera à tous les objets de `Entite` de faire ce qu'il doit. Pour parvenir à ce but nous avons un attribut `protected Action currentAction`.
Pour lancer les méthodes correspondantes à chaque action, nous avons :

```
switch(this.currentAction) {
    case ATTACK -> this.attack();
    case HARVEST -> this.harvest();
    ...
}
```

Quand l'entité meurt, le contrôleur la débarrasse du tableau.
Il est à noter que la classe *Entity* contient les déclarations, mais pas les implémentations, des méthodes `protected void harvest()` ; `protected void build()` ; `protected void create()` qui sont utilisées exclusivement par la classe *Navi*, ce choix a été fait afin de ne pas réimplémenter (Override) la fonction `update()` dans les trois classes filles. Les méthodes mentionnées sont donc implémentées dans la classe *Navi*.

Le déplacement :
Le déplacement est fait via l'algorithme A* (ci-dessous plus d'explications sur l'implémentation de l'algorithme). Sur la fenêtre graphique, le joueur sélectionne l'entité qu'il voudrait faire bouger, puis sélectionne le bouton de mouvement et fixe la case d'arrivée.

Les méthodes de déplacement sont : `public void generatePath()` ; `public boolean canMove(ICell c)` ; `public void move()`.

Le début de déplacement commence par la génération de chemin. Ceci est fait dans la méthode `generatePath()`. La méthode crée une nouvelle instance d'AStar, avec comme

position de départ celle du pion et celle d'arrivée l'attribut *destination* et récupère le chemin qu'il stocke dans l'attribut `protected Stack<ICell> path`.

La méthode `move()` est appelée par le contrôleur, a chaque laps du temps. Il fait avancer le joueur en se servant de `path`, s'il est vide, il rappelle `generatePath()` .

`move()`, à son tour, fait appel à `canMove(ICell c)` pour savoir si le pion peut se déplacer à la case *c*. `canMove` renvoie vrai si la case *c* est accessible et si elle est à côté de la position du pion.

L'attaque :
Pour faire attaquer un Na'vi, après l'avoir choisi sur une case, le joueur devra préciser la case à attaquer. Ainsi, l'action courante sera 'ATTACK' et pour destination il aura la case choisie.

Les méthodes d'attaque sont : `public boolean isEnemy(IEntity ent)` ; `public IEntity canAttack()` ; `public boolean attack()` ; `public void sufferAttack(int impact)`
Dans notre implémentation l'attaque se fait automatiquement.

En effet, pendant son déplacement, à chaque unité de temps, l'entité regardera autour de soi et attaquera l'entité ennemie s'il y en a une. Ceci est codé dans la méthode `public IEntity canAttack()` qui sera appelée de `public boolean attack()`, lui-même appelé de `Update`.

Autre que attaquer, une entité peut lui-même subir une attaque. La méthode `public void sufferAttack(int impact)` sera appelée et l'entité attaquée perdra *impact* de points de vie.

La construction :
Cette fonctionnalité est réservée aux Na'vis.
La fonctionnalité est réalisée en deux méthodes : `public void build()` ; `public boolean canBuildCamp(ICell c)`.

Le Na'vi, pour construire un camp, devra se déplacer et se positionner sur une case adjacente de la case ou le camp devra être construit (case demandée par le joueur). Le Na'vi construira le camp s'il se trouve sur une case adjacente de la case *c*, si *c* est accessible

et le joueur a `COUT_CAMP` (constante) de ressources. Ces conditions sont donc vérifiées par `canBuildCamp(ICell c)`.

En cas de réponse positive de `canBuildCamp` à `build()`, un nouveau camp sera ajouté à la liste des camps, sinon le Na'vi attendra de nouveaux ordres.

La récolte :
Cette fonctionnalité est réservée aux Na'vis.
La fonctionnalité de récolte est composée de deux méthodes : `public void harvest()` ; `public boolean canHarvest()`.

`harvest()` appelle `canHarvest()` pour savoir si le Na'vi est en mesure de récolter. Il en est capable s'il se trouve sur **le** champ de baies demandé et si celle-ci a encore des baies.
Si la récolte n'est pas possible, et s'il est sur le case demandée, le Na'vi ne fera rien, autrement dit, attendra de nouveaux ordres.

Quand le Na'vi récolte, les ressources sont directement ajoutées à la "banque centrale" du joueur, le Na'vi n'a pas à retourner et déposer les collect dans l'un des champs.

La création de Navis :
Cette fonctionnalité est réservée aux Na'vis, elle est réalisée par Camp.
La création de Na'vis est également divisée en deux méthodes : `protected void create()` ; `public boolean createNavi()`.

Quand la méthode `create` est appelée, le camp vérifie si le joueur possède assez de ressources pour construire un Na'vi, si oui, il appellera la méthode `boolean createNavi` qui ensuite regardera autour du camp et en cas de case libre, posera un Na'vi.

Le mouvement des Soldats
Les soldats ont un but : faire perdre le joueur. Pour accomplir cette soif de destruction, ils vont constamment bouger vers les camps. Le camp cible est choisi aléatoirement à partir de la liste de camp. Ensuite, une case adjacente accessible est trouvée. C'est depuis cette case que le soldat attaquera le camp.

A * (A etoile)

Comme algorithme de recherche de chemin nous avons choisi l'A*.

Notre choix est tombé sur cet algorithme car nous avons des obstacles (*Water* et entités sur chemin) sur le terrain et la différence possible de poids d'un *Cell* a l'autre. Pour l'implémentation nous avons procédé de manière suivante :

Nous avons trois tableaux d'entité de dimension 3 de taille [sizeGrid][sizeGrid][4]: `G`, `H` et `F`, où

- `G` indique le poids d'un *Cell* voisin
- `H` indique la distance entre le point d'arrivée et la position du Cell d'où on essaie de l'atteindre
- `F` la fonction heuristique

Dans ces tableaux-ci, il y a 2 valeurs qui nous intéressent : si la case a déjà été visitée (0/1) et sa valeur. Le tableau `F` a également les positions des cases parentes.

A la création d'un objet les tableaux sont initialisés aux valeurs 0 0 -1 -1. On peut le remarquer dans la méthode `init` de AStar.

```
private void init(int A[][][]) {
    for (int i = 0; i < this.map.sizeGrid; i++)
       for (int j = 0; j < this.map.sizeGrid; j++) {
             A[i][j][0] = 0; // visit
             A[i][j][1] = 0; // value
             // for f:
             // from what node parents have been initialised
             A[i][j][2] = -1; // x
             A[i][j][3] = -1; // y
       }
}
```

Une fois l'initialisation est faite, on peut commencer l'exécution de l'algorithme en appelant la méthode `public void aStar(ICell parent)`. `aStar` étant une méthode récursive, le paramètre `parent` sera la case de début au commencement et celle d'intermédiaire pendant son exécution.

On commence par calculer les valeurs des fonctions heuristiques des cases voisines à la position de départ à l'aide de la fonction `private void calcS(ICell inter)` , où inter est la position de la case sélectionnée. Après le calcul, on pourra choisir la case ayant le poids minimal et ainsi, enchaîner le processus jusqu'à ce que l'on se trouve sur la position d'arrivée.

Une fois ce processus terminé, on aura le chemin dans le tableau `F`, à ce stade, il ne restera que de l'extraire. Ce sera fait dans `public Stack<ICell> getPath()`. La méthode commence par vérifier si `aStar` a trouvé un chemin. Si la méthode a bien terminé, on doit avoir la valeur 1 dans le tableau `F`, en cas d'absence de 1, on peut conclure qu'il n'y a pas de chemin entre les 2 points.

```
int x = -1 , y = -1;

for (int i = 0; i < this.map.sizeGrid; i++)
       for (int j = 0; j < this.map.sizeGrid; j++)
             if(F[i][j][0] == 1 && F[i][j][1] == 1) {
                   x = i; y = j;
             }

if (x == -1) return null;
```

Si x n'est pas -1, on commence à entasser les Cell a suivre pour atteindre l'objectif et en le renvoie.

Comme structure de données pour représenter le chemin, nous avons choisi FILO / Stack.

Ordre joueur :

La transmission des ordres du joueur se fait en deux parties. La transmission des informations au modèle et le traitement de ces informations par le modèle. Par ailleurs il y a deux types d'ordre : le clic sur une case de la carte et le clic sur une action.

Clic sur une case de la carte :

Le controleur ActionControleur de la case cliquée envoie sa case qui a un MouseListener au modèle par la méthode clic.

Si le modèle n'est pas en attente d'une case, il sélectionne l'entité sur la case (null si il n'y en n'a pas). Sinon il transmet l'ordre à l'entité sélectionnée.

Le modèle est en attente d'une case quand il a une entité sélectionnée et que son action sélectionnée est MOVE, ATTACK, HARVEST, STOP, CREATE ou BUILD selon le type d'entité choisi.

La transmission d'ordre consiste à vérifier que l'ordre est correct puis à fixer la destination et l'ordre courant de l'entité sélectionnée.

Sélection d'acton :

Le contrôleur ActionControleur de l'action cliquée envoie son action qui a un MouseListener au modèle par la méthode selectAction.

Si l'action n'est pas valide, rien n'est fait. Si c'est une action directe, la transmet à l'entité sélectionnée. Sinon, la retient dans l'action sélectionnée.

Une action est valide si une entité est sélectionnée et que l'action appartient aux actions possibles de l'entité.

Les actions directes sont *STOP* et *CREATE*.

Sound :
Dans le Sound class, on importe certains wave filles pour notre jeux.

Il y a 4 wave files respectivement pour le background musique , move affect sound, appear effect sound et forbidden effect sound.

Il a aussi besoins de aller voir d'autre classes pour ajouter le correct sound. playMusic méthode est pour le background musique, donc il y a loop méthode, le musique peut recommencer quand il arrête. Et playSE méthode est pour le sound effect , il est vraiment rapid , donc il n 'y a pas de loop pour continuer. Et on justement appele playSE méthode respectivement dans les action méthode, il va avoir un sound .

L’affichage :

Dans l’optique de procéder à l'affichage de notre jeu, nous avons créé 3 classes différentes mais interconnectées : il s'agit de MainView, ViewMap et ViewControlPanel.

MainView à été choisi comme le cœur de ce système : son rôle est de réunir ViewMap et ViewControlPanel sous un même JFrame afin d'afficher d'un côté la carte et de l'autre le tableau de contrôle. Dans l’optique de les agencer correctement, nous utilisons un layout nommé GridBagConstraints dans lequel on ajoute librement nos éléments. En définitive, ViewMap sert à actualiser l'affichage dans son ensemble via la méthode update().

ViewMap est la classe qui dirige l'affichage des cellules de notre jeu et leur contenu. On y stocke notamment un ArrayList<JButton> cases qui nous donne l'occasion de conserver les JButton associés à chaque case, ce qui s'avère utile pour la récupération d'un clic comme expliqué plus tôt.

Lors de son initialisation, chaque case reçoit un ActionController qui, couplé avec le JButton, permet de renseigner le Mainmodel du jeu sur la case cliquée par l'utilisateur. À chaque update effectué, ViewMap va vérifier à travers les méthodes model.isWin() et model.isGameOver() si la partie a été perdue ou non. Dans les deux cas, un JOptionPane affiche un message indiquant à l'utilisateur la conclusion de la partie.

Si aucune de ces deux conditions ne sont remplies, l’affichage est alors mis à jour. La liste des cases est parcourue, et pour chacune d'entre elles la fonction type(ICell c, Graphics g, int x, int y) vérifie le type de la case étudiée afin de terminer les graphismes à afficher. Peu importe le type de de case et d'entité qui s'y trouve, la méthode frame(Graphics g, String s, int x, int y, int width, int height) est appelée afin de dessiner sur Graphics g l’image associée aux coordonnées correspondantes.

ViewControlPanel est la classe responsable de l’affichage du panneau de contrôle de notre jeu. Il contient à l’écran des informations sur l'entité sélectionnée, la quantité de ressources possédées, des indications sur le nombre de "berries" nécessaires à la création d'un camp / Navi, et pour finir des boutons à l'image des actions réalisables. Tout ces éléments sont dessiné avec update() sur le Graphics g en suivant la même logique que pour ViewMap.

Les boutons en question sont gardés dans un ArrayList<JButton> cases afin de détecter leur clic sur le même principe de fonctionnement que pour les cases, à la différence que lors de leur création un String leur est associé. Celui-ci correspond au nom de l'action associée au JButton, ce qui nous permettra non seulement de récupérer l'action voulue mais également d'afficher l'image correspondante. Un élément ToolTipText leur est également attribué, ce qui servira à afficher des détails sur une action donnée lorsque le curseur passera sur son JButton.

Les contrôleurs :

Le fonctionnement de nos contrôleurs se divise en 2 parties distinctes : l'une contenant simplement ActionController servant pour les JButton uniquement, et l'autre regroupant un contrôleur principal (MainControl) englobant deux autres contrôleurs indépendants (EntityControl et BerriesControl). Ces derniers sont le squelette des Threads de notre jeu : ils sont tous conservés et démarrés au cours de la création du MainControl depuis le Main principal.

Tout d'abord, ***EntityControl*** se charge de diriger toutes les entités selon leurs paramètres. À chaque appel de run(), celui-ci parcourt le tableau model.entities dans son entièreté et applique à chaque élément la fonction update() rédigée dans IEntity. De plus, à la suite de cette opération EntityControl étudie si les conditions sont réunies pour générer un nouveau Soldier sur la carte grâce à la méthode fournie createSoldier().

Quant à ***BerriesControl***, cette classe est calquée sur le même fonctionnement que celui d'EntityControl. Cette fois-ci, c'est l'attribut ArrayList<Berries> berries qui est scruté lors du lancement de la méthode run(). Cette liste abrite toutes les "berries" existantes sur la carte, ainsi sur chacune d'entre elles la fonction update() est appliquée. Cela à pour effet la modification, si elle a lieu d'être, de l'image liée au "berries" via b.newFrame() selon la quantité de baies qui leur reste.

Dans le cas de ***ActionController***, il s'agit tout bonnement de capter l'origine d'un clic de l'utilisateur. Chaque JButton étant associé à ce type de contrôleur, on peut par conséquent y stocker le JButton en question ainsi que la ICell associée si elle existe. Une fois MousePressed() activé, s'il s'agit d'un bouton d'action un switch se chargera de détecter de

quelle action il s'agit et d’envoyer cette valeur à model.selectAction. Autrement en présence d'une ICell, on lance la méthode model.clic(cell) qui décidera de la suite d'action à accomplir pour une entité donnée.

Résultat :

Finalement, nous avons un jeu fonctionnel. Les buts ont été globalement atteints.
Le jeu a un joli affichage dans une fenêtre graphique, la fenêtre reçoit les clics et accomplit les ordres du joueur.

Les Na’vis sont capables de trouver un chemin vers les cases demandées, récolter des baies, construire des camps et attaquer en suivant bien les ennemis.

Les soldats bougent vers les camps pour les détruire, une fois tous les camps détruits le jeu s'arrête.

Les camps peuvent être construits, détruits et peuvent également générer des Na’vis.

Les ressources des baies ne se régénèrent pas une fois toutes consommées.

Documentation Utilisateur

Notre équipe est composée d'amateurs de logiciels libres (open source). Le code est donc libre à tous, il n'y a pas de fichier compilé.

Pour jouer à notre jeu , il faut avoir un compilateur de java, la plupart d'entre nous avons utilisé ceux de VSCode, Éclipse, IntelliJ. L'avantage de ceux-là est que chacun peut modifier le code du jeu, l'adapter à ces goûts et caprices.

Le code source est disponible sur notre repos Git.

Quand le jeu se lance, le joueur verra une fenêtre graphique avec des pions et de différentes cases. Les cases bleues représentent l'eau, cases inaccessibles. Aucun pion ne peut aller dessus ou les traverser. Celles-la sont les seules cases infranchissables. Les cases contenant un arbre sont les champs de forêt, celles avec des gros points rouges sont les champs de baies. Finalement, les cases vertes sont les cases terrains.

Les pions ont la même vitesse sur toutes les cases.

Il y a deux pions dans le jeu : Na'vi et Soldat. Le joueur joue pour les Na'vis, son but est de protéger les camps le plus longtemps possible des soldats qui se dirigent toujours vers les camps différents. Le jeu s'arrête une fois tous les camps détruits par les soldats. En revanche, si la totalité des Soldiers ont été tués c'est une victoire pour le joueur.

Le joueur commence avec un camp et deux Na'vis. Sur les champs de baies il pourra récupérer des ressources, tout en faisant attention à ne pas les récolter toutes, car si le champ de baies est vidé de baies, il n'en reproduira plus. Les ressources servent à construire de nouveaux camps et à faire naître de nouveaux Na'vis.

Seuls les Na'vis et camps sont contrôlables.
Pour s'y prendre, le joueur en cliquant sur un Na'vi doit sélectionner une action parmi construction, récolte, attaque et déplacement. Ces actions sont accompagnées par un nouveau clic sur la case ou il voudrait aller / construire un camp / récolter, l'ennemi qu'il voudrait attaquer.

Documentation Développement

Si quelqu'un reprend notre code pour créer une version améliorée de notre projet, il devra examiner les classes ControlEntity et Entity. Dans la classe ControlEntity, il pourra modifier les fréquences de mise à jour, comme le temps qu'il faut pour créer un nouveau soldat. Dans la classe Entity, il pourra voir les actions des camps, des soldats et de Navi, et il pourra modifier les caractéristiques des entités. La méthode "main" se trouve dans la classe "Main", et les variables statiques de réglage sont situées dans leur classe respective. Ces variables auraient pu être centralisées pour un contrôle plus intelligent. Pour améliorer le code, il devra examiner la partie "conclusion et perspectives".

Conclusion et Perspective :

En conclusion, le développement de ce jeu Avatar a été un processus passionnant et stimulant. Tout au long du développement, nous avons été confrontés à divers défis, tels que la conception des mécanismes du jeu, la création des graphiques et l'optimisation de l'agencement de nos différentes classes. Cependant, en tirant parti de nos compétences en programmation et de notre travail d'équipe, nous avons pu surmonter ces difficultés et créer un jeu fonctionnel et divertissant.

En rétrospective, il y a certains aspects du projet que nous aurions pu améliorer si nous avions disposé de plus de temps. Tout d'abord, nous aurions pu mettre en place une gestion plus sophistiquée du déplacement et des actions automatiques des ennemis afin de rendre le jeu plus exigeant et plus captivant : en effet, leur vitesse n’a au final pas été prise en compte pour le déplacement.

Deuxièmement, nous aurions pu ajouter plus de niveaux avec des thèmes et des niveaux de difficulté différents pour augmenter la rejouabilité du jeu. Ajoutons également que des animations pour les cases étaient prévues (cf : les frames dans View/Ressources) mais nous n'avons pas eu suffisamment de temps pour les implémenter.

Un aspect important à souligner est que dans la phase de brainstorming de notre jeu, nous avions pour ambition d'ajouter une variété d'animaux qui auraient aidé nos joueurs à se déplacer plus vite entre autres. Ce concept aurait été intéressant à creuser afin d'augmenter les possibilités de stratégie. Enfin, nous aurions pu ajouter une fonctionnalité multijoueur pour permettre aux joueurs de s'affronter.

Nous n’avons pas mis en place un système de score ou d’augmentation progressive de la difficulté qui aurait entraîné un sentiment de progression. L’idée de protection de l’arbre sacré a été abandonnée et remplacée par la protection des camps.

Malgré ces points à améliorer, la réalisation de ce jeu nous a permis d'acquérir une expérience précieuse en matière de création de jeux et nous a donné un aperçu des possibilités de ce langage dans le domaine de la réalisation de gameplay à la fois interactifs et attrayants.