Rapport Projet PCII

Abel HENRY-LAPASSAT, William GRAVIL, Martin MASSON, Amaury RUELLE

Introduction -

Le but de ce projet est d'implémenter un jeu de gestion en utilisant de la programmation concurrente et évènementielle, pour cela nous avons choisis un jeu dans lequel le joueur contrôle une ville et ses habitants dans le but de la faire progresser au maximum dans un laps de temps limité. Il doit faire cela en contrôlant des worker pour récolter des ressources afin de construire des bâtiments qui permettentron d'augmenter et d'automatiser la récolte de ces ressources pour augmenter au maximum la progression de la ville.

Analyse Globale -

Pour se faire les principales fonctionnalités impléméntées sont :

- Affichage du monde (map) sous forme d'une grille hexagonale dans laquelle chacun des hexagones représente un terrain avec un type prédéfinis auquel est associé une couleur précise.

- Affichage des worker sous forme d'un cercle dans le terrain sur lequel ils se situent. Le cercle est coloré en rouge dés que le travailleur est occupé. Et les mouvements de ces derniers sont affichés aussi.

- Affichage du panneau de contrôle sous la forme de plusieurs Jpanel contenant chacun des informations précises à certains éléments du jeu (illustration de la case cliquée, inventaire de la ville, actions possible & informations de la case cliquée).

- Nombreux types de terrains différents avec chacun un type de ressources associé (Forest → Wood, Mountain → Stone, Iron Deposit → Iron, Plain → None, City → None) et des spécificité d'intéractions).

- Différents rôles pour les workers selon leur capacités quant aux actions à effectuer, chaque rôle est assigné à un type de ressource et de bâtiment avec lesquels le worker en question pourra intéragir.

- Différent types de bâtiment selon leur utilité et possibilité d'intéraction dans le jeu, certains bâtiment génèrent simplement des ressources de manière plus efficace qu'un simple territoire, d'autres permettent au joueur de faire appel à plus de workers d'un certain type.

- Inventaire propre a chaque case, bâtiment, travailleur et ville, permettant de stocker, prendre, et échanger des ressources entre les différentes entités.

- Création aléatoire du monde selon des pourcentages d'apparition des terrains de base (par ordre de pourcentage d'apparition : Plain, Forest, Mountain, Iron Deposit).

- Séléction de case par clique sur la map.

- Prédicat 'occupé' pour chacun des workers qui permet d'indiquer lorsque ceux-ci effectuent une tâche, et ne sont donc pas disponible.

- Collecte de ressource par le worker le plus proche d'une case contenant des ressources (si cette dernière est la case cliquée), ayant le rôle souhaité, qui se déplace donc jusqu'à la case puis qui récolte un certain nombre de ressources associées.

- Déplacement libre des workers grâce au bouton "Move" permettant au worker le plus proche d'une case de s'y rendre.

- Création de bâtiments de diverses sortes afin d'augmenter le nombre de worker possible ou alors d'augmenter la quantité de ressources produites par un territoire.

- Algorithme de Pathfinding entre 2 cases pour tous les déplacements voulus.

- Décision des boutons d'actions à activer selon la case cliquée selon son état et son type.

- Création de nouveau travailleur grâce au bouton "Train", activé lorsque l'on clique sur un bâtiment le permettant.

- Possibilité d'acheter/vendre des ressources grâce au bâtiment 'Shop'.

- Création d'une nouvelle Jframe lors du clique sur le bâtiment 'Shop' pour accéder à l'interface de vente.

- Création d'une nouvelle Jframe lors du clique sur le bouton "Build" si on est sur une case 'City' afin de décider du bâtiment à y construire.

- Dépôt des ressources de tous les workers grâce au bouton "Drop", qui force tous les workers disponibles à se rendre au 'CityHall' pour transférer leurs ressources dans l'inventaire de la ville.

- Capacité d'inventaire limité de manière individuelle pour chaque entité.

- Dépôt automatique des ressources d'un worker lorsque ce dernier possède un inventaire plein grâce à un thread continu.

- Actualisation constante de l'affichage grâce à un thread continu.

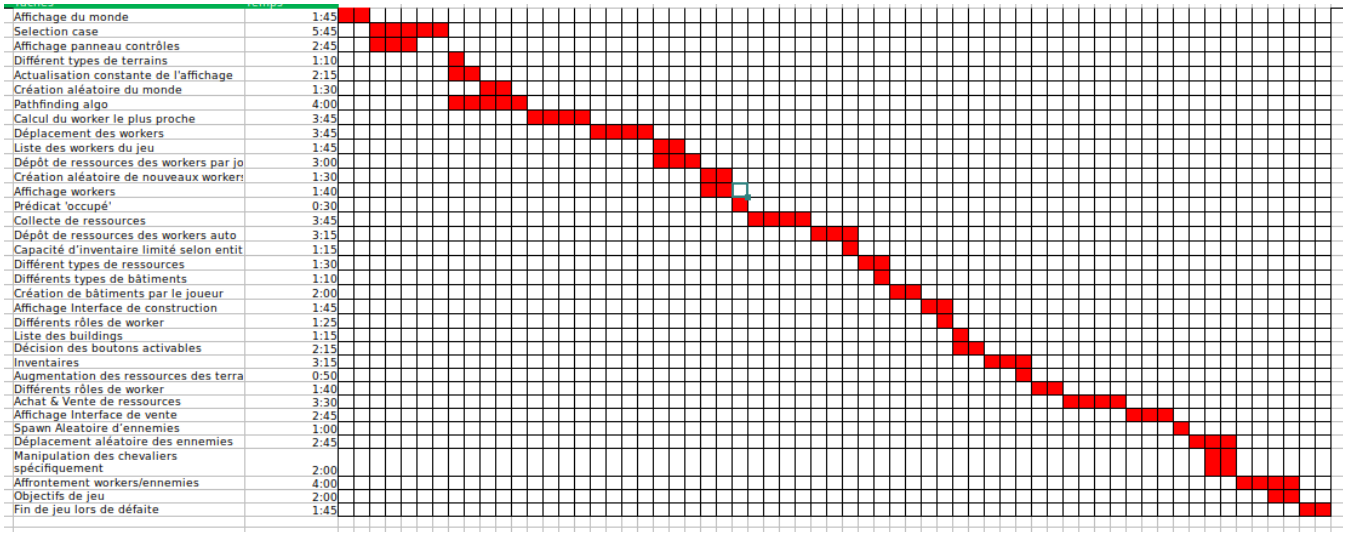
- Création d'un thread spécifique à chaque nouvelle action que le joueur souhaite effectuer.

Plan de développement – *(time in minutes)*

Fonctionnalités	Réflexion	Conception	Développement	Test
Affichage Monde	15'	30'	30'	30'
Affichage Workers	5'	45'	120'	30'
Affichage Panneau de Contrôle	15'	15'	120'	15'
Différents types de terrains	30'	10'	15'	15'
Différent types de ressources	30'	15'	30'	15'
Différents rôles de worker	45'	10'	15'	15'
Différents types de bâtiments	30'	10'	15'	15'
Inventaires	30'	30'	90'	45'
Création aléatoire du monde	15'	30'	15'	30'
Liste des workers du jeu	15'	15'	30'	45'
Liste des buildings	15'	15'	15'	30'

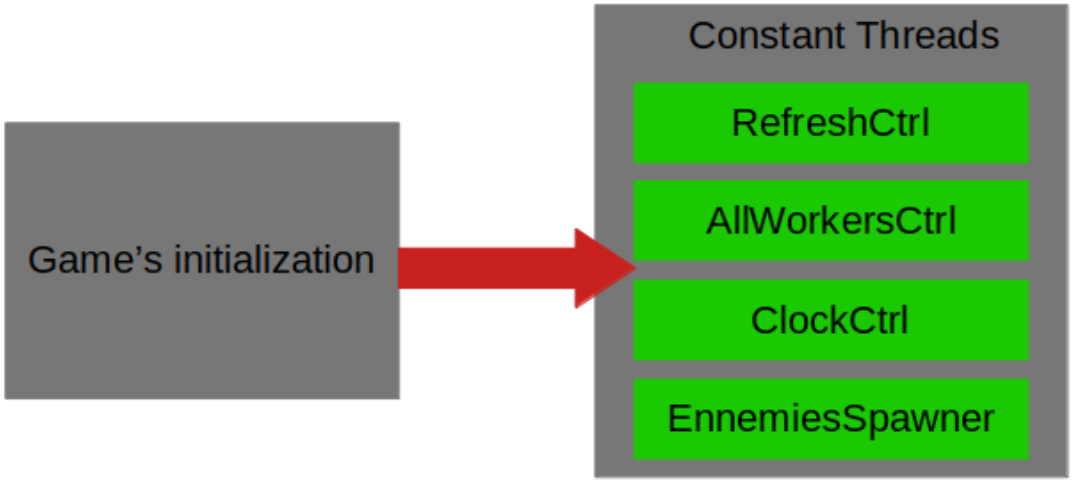
Séléction de case par clique (algo)	45’	+120’	+120’	60’
Prédicat ‘occupé’	5’	5’	5’	15’
Calcul du worker le plus proche	30’	45’	30’	120’
Collecte de ressources	30’	60’	90’	45’
Déplacement des workers	45’	60’	60’	60’
Création de bâtiments par le joueur	30’	60’	60’	30’
Pathfinding (algo)	15’	60’	120’	45’
Décision des boutons activables	30’	45’	45’	15’
Création de nouveaux workers par le joueur	15’	30’	30’	15’
Achat & Vente de ressources	30’	45’	90’	45’
Affichage Interface de vente	30’	45’	60’	30’
Affichage Interface de construction	15’	30’	45’	15’
Dépôt de ressource des workers par choix du joueur	15’	45’	30’	30’
Capacité d’inventaire limité selon entité	45’	30’	15’	45’
Dépôt automatique de ressources des workers	15’	30’	90’	60’
Actualisation constante de l’affichage	15’	30’	60’	30'
Augmentation des ressources des terrains en temps constant	5’	15’	15’	15’
Spawn Aleatoire d’ennemies	15’	30’	30’	15’
Déplacement aléatoire des ennemies	30’	30’	45’	60’
Manipulation des chevaliers spécifiquement	15’	45’	30’	30’
Affrontement workers/ennemies	45’	45’	90’	60’
Objectifs de jeu	45’	30’	15’	30’
Fin de jeu lors de	15’	15’	30’	45’

défaite				
Fin de jeu à la fin du temps	5’	5’	15’	30’
Calcul du score	15’	30’	30’	60’

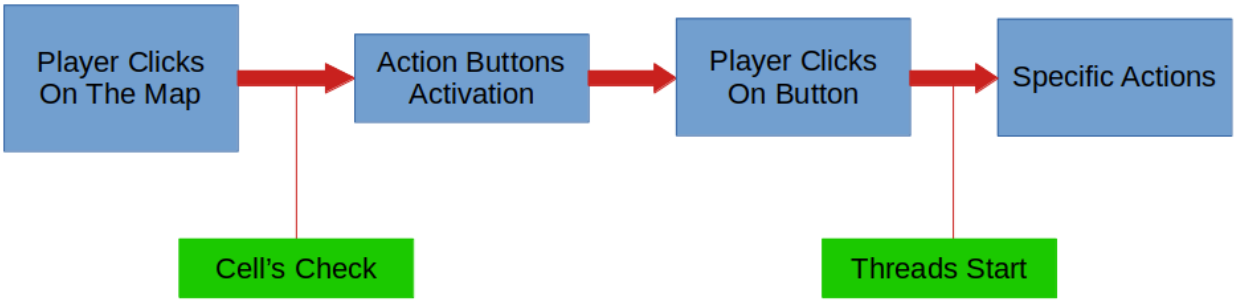


Conception Générale -

Pour ce qui est des actions “passives” du code, on y retrouve le contrôle de certaines actions des workers (dépôt automatique de ressource et recovery en cas de combats), le spawn aléatoire d’ennemis ainsi que leurs déplacements et agissements, mais aussi l’augmentation des ressources des terrains ainsi que l’actualisation constante de l’affichage. Toutes ces actions sont initialisées soit lors de la création du Game Controler, soit lors de la création de nouveaux workers/ennemy :



L’élément principal du fonctionnement du jeu est le click sur un terrain, en effet c’est lors du click sur un terrain que l’ont peut accéder à toutes les informations quant aux actions que l’on peut effectuer ou non. Donc lorsque le joueur clique sur un terrain, ce dernier est indiqué comme le terrain courant, ce qui entraîne plusieurs évènements spécifiques, et ensuite les boutons d’action sont activés/désactivés selon l’état de ce terrain courant :



Donc une fois qu'on a cliqué sur un terrain et que les boutons d'actions se sont activés, on peut cliquer sur eux, et chacun d'entre eux lance alors une série d'événements spécifique à sa fonction :

Le bouton "Build" → Récupère le worker adéquat le plus proche et l'envoie sur la case voulue avant de faire appelle aux fonctions de création d'un nouveau bâtiment du type voulu.

Le bouton "Upgrade" → Si l'upgrade se fait sur un bâtiment, alors récupère le worker le plus proche et l'envoie sur la case voulue avant d'appeler les fonctions de LevelUp adéquat pour un bâtiment. Sinon appelle simplement les fonctions de LevelUp adéquat pour un worker

Le bouton "Move" → Récupère le worker "Knight" le plus proche et l'envoie sur la case désignée.

Le bouton "Collect" → Récupère le worker au rôle correspondant au type de ressource de la case désignée et l'envoie dessus pour y récolter le montant maximal.

Le bouton "Train" → Crée un nouveau worker du type associé au bâtiment d'entraînement en question sur la case occupée par ce dernier.

Le bouton "Expand" → Récupère parmis tous les workers ("Knight" exclu) le plus proche afin de l'envoyer sur la case désignée pour transformer le type du terrain en "City".

Le bouton "Shop" → Instantie un nouveau set (Model+View+Controler) pour gérer une interface de Shop intéractive avec le joueur.

Le bouton "Drop" → Selon le cas de figure, instantie un ou plusieurs thread "WorkerDrop" forçant le/les workers concernés à transferer leur inventaire vers celui de la ville.

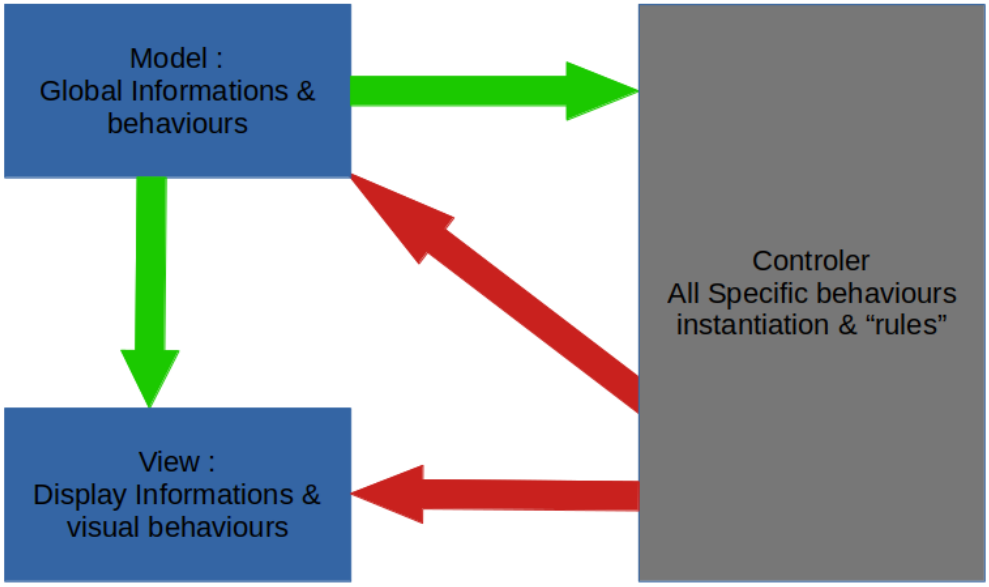
Pour ce qui est du fonctionnement général du jeu, nous avons utilisé un modèle MVC (Model-View-Controler), dont le fonctionnement général est le suivant :

Le modèle représente les données et la logique métier de l'application. Il est responsable de la récupération, de la mise à jour et de la persistance des données.

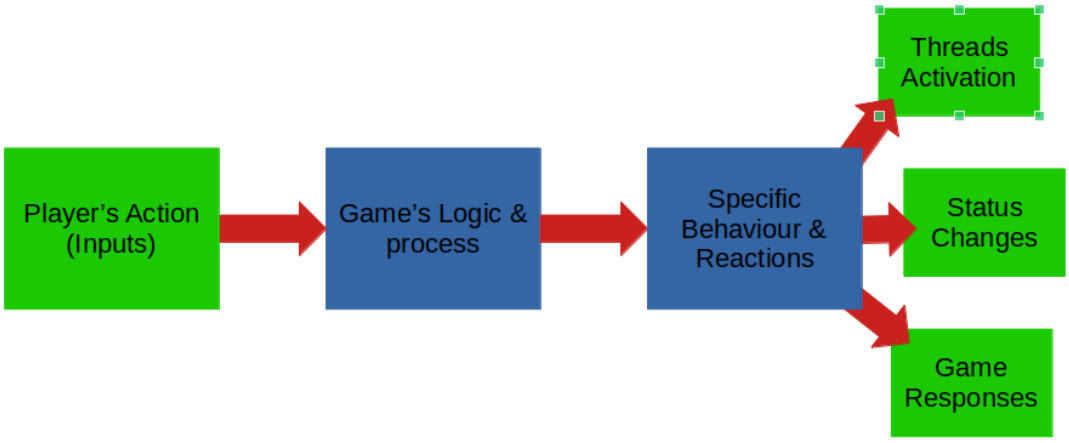
La vue est responsable de l'affichage des données. Elle récupère les données depuis le modèle et les présente à l'utilisateur de manière compréhensible.

Le contrôleur agit comme un médiateur entre le modèle et la vue. Il reçoit les entrées de l'utilisateur via la vue et utilise ces entrées pour effectuer des actions sur le modèle. Il peut également mettre à jour la vue en fonction des modifications apportées au modèle.

En résumé, le modèle MVC sépare les responsabilités de l'application en trois parties distinctes afin de rendre l'application plus facile à maintenir et à étendre.



Qui plus est nous utilisons de la programmation événementiel et concurentielle, ce qui fait que le controller s’occupe d’attendre certaines actions du joueur, d’instantier des threads spécifiques à ces dernières, mais aussi d’en instantier suite à certains événements interne au jeu.



Conception Détaillée -

Pour CHAQUE fonctionnalité → principales structures de données utilisées + algo + conditions limites + utilisation du code par autres fonctionnalités

Affichage du monde :

Pour l’implémentation du monde nous utilisons une grille hexagonale avec une disposition “even-q” verticale, qui est représentée dans le code par unr ArrayList 2D dans la classe ‘MapModel’. Pour ce qui est donc de l’affichage nous utilisons un algorithme qui pour chacune des coordonnées de la grille associe l’hexagone correspondant de la grille dans un Jpanel dédié au

monde contenu dans la classe ‘MapView’. Pour cet affichage nous utilisons les coordonées attribuées à chaque terrain de la grille lors de l’initialisation afin de dessiner l’hexagone correspondant à l’endroit souhaité, en calculant sa position par rapport aux coordonnées d’origine de la grille et les dimensions pré-définies. Il n’y a pas vraiment d’algorithme pour dessiner chacun des terrains cela se fait par calculs trigonométriques en fonction du centre de ce dernier.

Affichage des workers :

Chaque entités d’un worker possède une instance de WorkerView qui va permettre d’afficher le Worker. Il n’y a ici aucune structures de données particulier si ce n’est la classe workerModel qui contient toute les informations du joueur que l’on souhaite affiché.
Les constante du Modèle sont la taille et la hauteur du worker, celle-ci sont stocker directement dans workerModel et initalisé à 10.
Les autres Constante du Modèle sont les coulleurs des workers qui dépendent de leur Role.

Dans la classe WorkerView deux fonctions s’occupe de l’affichage du joueur en fonction de si le worker est entrain de se déplacer.

- Dans le cas ou le jour est immobile :
  Il s’agit de la Méthod DrawWorker qui s’occupera de l’affichage.
  Elle récupera la position du joueur sur la map, puis dessinera un oval de hauteur et largeur 10 au centre de la cellule.

- Dans le cas ou le worker est entrain de ce déplacer :
  Il s’agit de la méthode drawMove qui prendra le relai.
  Cette fonction ira chercher dans le modèle les coordonnées sur l’écran du worker et dessinera un oval de hauteur et largeur 10.

Ces codes sont appellé par la mapView et la mapView devra faire attention a la disjonction de cas enfonction de la variable booleen Moving. En parralèle un Thread ici WorkerMoveController s’occupera de mettre à jours la position du joueur sur l’écran lorsqu’un jours se déplace. La position sur l’écran sera calculé simplement en faisant la somme de la positions de départ et et de la position d’arrivé puis l’on divise par un compteur qui s’incrémente a chaque frame.

Affichage du Panneau de Contrôle :

Pour l’affichage du panneau de contrôle, nous avons décidé de le découper en 3 sous JPanels ayant chacun leur spécificités, que nous avons organisés à l’aide d’un simple BorderLayout().

Le premier JPanel, la CellInfoView, affiche des informations relatives à la case sélectionnée, ainsi qu’au potentiel bâtiment construit dessus.

Le deuxième JPanel, la CityInfoView, affiche des informations relatives à l’inventaire global (dans notre jeu, de la ville) ainsi qu’à tous les workers.
Enfin, le troisième et dernier JPanel, l’ActionView, fait office de panneau de contrôle avec en son sein tous les boutons permettant au joueur d'interagir avec le jeu.

La logique derrière ce découpage a été la suivante. Les boutons étant toujours affichés et ayant donc une position fixe, ils ont naturellement été isolés dans un panel dédié. Pour ce qui est des informations relatives à la ville, elle diffère des deux entre dans la mesure où elle ne change que rarement (drop de ressources, création/mort d’un worker) est ne nécessite par conséquent pas d’être régulièrement mise à jour. De l’autre côté, les informations sur la cellule, tout comme les boutons, changent constamment car pratiquement chaque action demande la sélection d’une case. Ces deux derniers ont donc été séparés dans deux JPanel différents, afin d’avoir une mise à jour plus simple et propre pour chacun d’entre eux.

- CellInfoView : Pour l’affichage du CellInfoView, nous avons utilisé un GridBagLayout. Les JLabels ont été séparés en 2 colonnes avec les informations complémentaires (ressource max,

niveau) dans la colonne de droite. Une point d'attention est qu'une grande partie des informations disponibles sur une cellule ne peuvent en fait pas être modifiées, c'est par exemple le cas pour les bâtiments sur les Plaines ou la quantité de ressource sur les Villes.

L'idée a donc été de cacher ces informations non pertinentes pour n'afficher que celles utiles en fonction pour chaque case de son type de se la présence où non de bâtiment. Afin d'avoir ces fonctionnalités, tout en ayant constamment les JLabels alignés poussés contre le bord du haut, nous avons choisi d'attribuer une GridBagConstraints à chaque JLabel, afin de les modifier facilement dynamiquement en fonction de la case sélectionnée.

Ainsi, dans la fonction update() qui met à jour les JLabel, la fonction setLowest() est appelée sur le dernier JLabel qui va être affiché pour cette case (le plus bas) et ajuste son weighty afin qu'il prenne toute la place disponible et compresse les autres JLabel vers le haut. Cette fonction fait elle-même appel à son début à la fonction resetConstraints() qui remet à jour les contraintes de chaque JLabel afin notamment celles de l'ancien dernier JLabel ayant son weighty ajusté.

Finalement, la cellule sélectionnée est affichée en agrandie au bas du JPanel à l'aide des fonctions déjà existantes pour leur affichage sur la carte surchargée afin d'accommoder à leur nouvelle taille. La fonction d'update() est finalement appelée à chaque nouveau clic sur la carte ainsi que périodiquement par le CellCtrl, le thread responsable de la régénération naturelle des ressources, afin de pouvoir observer dynamiquement la quantité de ressource de la case augmenter dans la View.

- CityInfoView : L'affichage du CityInfoView, utilise lui aussi un GridBagLayout mais les JLabel étant statiques, seulement deux GridBagConstraints sont utilisés, une par colonne. L'affichage est lui plus simple et ne fait appel qu'au différents model pour récupérer les informations requises afin de les afficher. La mise à jour se fait alors à chaque action effectuée par un worker qui modifierait la quantité de ressources disponible (dépôt, construction, amélioration, entraînement d'un nouveau worker), ainsi qu'à chaque combat au cas où un worker surviendrait à mourir.

- ActionView : Enfin, pour l'affichage de ce dernier JPanel, nous avons aussi utilisé un GridBagLayout. Pour utiliser ce GridBagLayout il faut définir une GridBagConstraints. Elle nous sert à faire respecter certaines contraintes chaque élément que l'on ajoute à notre JPanel.

Tout d'abord on définit que chaque élément prendra toute la place dans la case. Puis, on définit la taille minimum de chacune des cases du tableau. On utilise la variable ipadx ou ipady. Cette variable est le padding interne de la case. Il faut aussi prendre en compte que les boutons que l'on rajoute dans les cases ont eux aussi une taille minimum. Par exemple, pour faire un bouton de 20 de large on utilisera la commande Constraint.ipadx = 20 - button.minimum Size(). On définit aussi de la marge extérieure entre chaque élément du tableau avec la classe Insets. Ensuite on sélectionne une case du tableau à l'aide gridx et gridy.

On crée alors un bouton avec un bouton auquel on attribue un nom, un tooltip et une icône. Il faut faire attention que l'icône que l'on à ajouter au bouton augmente la taille minimum de celui-ci. Pour finir on ajoute à notre JPanel avec add(button, contraint). Il suffit juste alors de créer 8 autres boutons et des ajoutés au JPanel. Ces boutons sont ensuite mis dans une liste et qui peut être obtenue avec getButtonlist(). Il est aussi possible d'obtenir un bouton selon son nom avec GetButtonFromName().

Différents types de terrains :

Chacun des terrains créé se voit attribué a l'initialisation du monde un types spécifique, définis dans le type enuméré 'CellId', qui permettra par la suite de définir les différentes actions possible quant à ce terrain, notamment pour la construction de bâtiment, qui est différente selon chacun des terrains présent sur la map.

Différents types de ressources :

Dans le jeu il est possible (voir nécéssaire) de collecter des ressources, pour se faire nous avons définit dans le type énuméré ‘Resource’ toutes les ressources existantes, correspondant chacune à un type de terrains sur lequel elles apparaissent. Chacune d’entre elle sera aussi utilisé dans des quantités différents selon les cas, mais globalement nous avons instauré un système de “rareté” où le bois est la ressources la plus courante, vient ensuite la pierre et enfin le fer, nous avons aussi instauré l’argent comme ressource non pas collectable mais échangeable.

Différents rôles de workers :

Comme expliqué plus tôt chaque worker se voit attribué à sa création un rôle spécifique déterminant les actions que ce dernier devra/pourra effectuer lorsque le joueur le lui demandera. Ces rôles sont définit dans le type enuméré ‘WorkerRole’, et la principale différence est le type de ressources avec lequel le worker interagira, en effet le “LumberJack” interagira avec le bois, et la construction de “SawMill”, le “Miner” avec la pierre ainsi que la construction de “Mine”, et le “QuarryMan” avec le fer ainsi que la construction de “Quarry”. Le dernier type est lui définit de sorte à ce que les workers créés comme des “Knight” ne servent qu’à défendre les autres face aux ennemies présent dans le jeu.

Différents types de bâtiments :

Tous les bâtiments du jeu sont définis au même titre que les workers, lors de leur initialisation, comme ayant un type immuable, définit par le type enuméré ‘BuildingId’. Ils sont classés tout au long du code en 2 catégories, les bâtiments de production et les bâtiment d’entrainement, les bâtiments de production sont liés aux terrains qui produisent des ressources, et ceux d’entrainement sont liés aux terrains “city”. Grâce aux bâtiment de production le joueur peut récolter des ressources plus rapidement, et grâce aux bâtiments d’entrainement il peut créer de nouveaux workers.

Inventaires :

Pour les inventaires, le but étant d’associer des quantités entières à des types de ressources, le choix s'est très rapidement tourné vers des *HashMa*p. Cependant, de part l’utilisation de thread, et donc la possibilité de modification simultanée de la structure de données, nous avons finalement décidé d’opter pour des *ConcurrentHashMap*.

Étant prévu pour être utilisés comme stockage de ressource par les Cases, les Buildings, les Workers et la Ville, différents constructeurs ont été créés afin de donner un maximum de flexibilité et d’aisance d’utilisation de la structure. Ainsi, en plus des getters/setters spécialisés modifiant la quantité d’une seule ressource, ont été ajoutés des fonctions comme *transfer()*, permettant de transférer le contenu d’un inventaire vers un autre, afin de faciliter l’échange de ressources entre les inventaires des tous les différents modèles du jeux. En outre, la quantité de ressource maximum *max_resource* pouvant être stockée dans l’inventaire est elle aussi modifiable afin de permettre facilement, un moyen d'améliorer de tous les objets ayant un Inventaires

Création Aléatoire du monde :

Lors de l’initialisation du monde dans le constructeur de la classe ‘MapModel’, les types de terrains sont créés de manière aléatoire, selon un certains pourcentage d’apparition. Le plus probable d’apparaître étant les terrains de type “Plain” suivi par ceux de type “Forest” puis ceux de type “Mountain” et enfin ceux de type “Iron_Deposit”. Après cela le point de départ de la ville est définit aléatoirement, puis le bâtiment principal “CityHall” est construit dessus et les deux cases adjacentes sont transformées en “City” avec un bâtiment “Shop” sur l’une d’entre elles.

Après cela sont définis les terrains sur lesquelles apparaitront les ennemies, nous avons choisit de définir 10 terrains de type “Forest” aléatoiremetn comme dfes ‘spawners’ de loup, ce de manière aléatoire, dans une bouvle while.

Liste des Workers existants :

Tous les modèles des workers du jeu sont contenu dans une ArrayList de ‘WorkerModel’ de la classe ‘GameModel’, qui a évidemment sa liste correspondante pour les vues liées aux modèles, grâce à laquelle ils sont accessibles à tout instants et actualisé à chacune des actions qu’ils effectuent (idem pour les view). Cette liste est aussi actualisée lors de la création d’un nouveau worker, ainsi que lorsque l’un d’eux est tué.

Liste des Bâtiments cosntruits :

Similairement aux listes utilisées pour les workers, nous avons fait des listes pour les bâtiments du jeu, organisées de la même manière que celles des workers, à la seule différence que les bâtiments ne peuvent pas être supprimés car rien n’est fait pour les détruire dans le jeu.

Selection de case :

Pour la séléction d’une case lors d’un clique nous avons utilisé un ‘mouseListener’ dans la classe ‘MapCtrl’ qui lorsque le joueur clique dessus, récupère les coordonnées en pixels du click, puis grâce à la méthode ‘GetCoordFromClick()’ de la classe ‘MapModel’ récupère les coordonnées dans la grille du click effectué, cela grâce à un algorithme qui nous a demandé plusieurs heures de réflexions et différentes implementations…

Les différentes méthodes imaginées furent les suivantes :

- Découper les hexagones représentant les cases en plusieurs parties et interpoler le clique selon la partie de l’hexagone, après avoir ramener les coordonnées a un groupe de seulement 4 hexagones.
- Récuperer d’abord la colonne du clique puis regarder dans cette colonne à quelle case il appartenait. (idem avec la ligne en première).
- Découper la grille en groupe d’hexagones eux-mêmes découpés en groupe, qui une fois affinés permettraient de déterminer grâce à des modulos la région des hexagones à laquelle le click appartient (suggérée par notre professeur).
- Récuperer approximativement les coordonnées du clique dans la grille puis affiner grâce à de la trigonométrie parmis les différentes possiblités obtenues.

Bien que toutes ces méthodes se ressemblent nous avons finalement décidé de choisir la dernière car c’était la plus concluante, et voici une explication plus approfondie de l’algorithme présent dans la fonction ‘GetCoordFromClick()’ :

Tout d'abord, il calcule les dimensions de chaque cellule hexagonale en fonction de la taille de la cellule (en pixels) fournie en entrée. Il utilise la trigonométrie pour calculer la hauteur et la largeur de chaque cellule hexagonale.

Ensuite, lorsque l'utilisateur clique sur la grille, l'algorithme calcule une colonne et une ligne approximatives en fonction de la position x et y du clic. Pour cela, il divise la position x par la largeur de la cellule et arrondit le résultat vers le bas pour obtenir la colonne approximative. Il divise la position y par la hauteur de la cellule et arrondit également le résultat vers le bas pour obtenir la ligne approximative.

Après avoir obtenu les colonne et ligne approximatives, l'algorithme utilise encore une fois la trigonométrie pour ajuster les indices de colonne et de ligne en fonction de la position exacte (plus ou moins) du clic sur la cellule hexagonale.

L'algorithme calcule deux variables f1 et f2 qui représentent la position des deux diagonales d'une cellule hexagonale. f1 est la diagonale qui va du coin supérieur gauche au coin inférieur droit, et f2 est la diagonale qui va du coin supérieur droit au coin inférieur gauche. Ces variables sont calculées en fonction de l'angle de chaque hexagone et de la position x du clic sur la cellule.

Ensuite, l'algorithme compare la position y du clic avec les variables f1 et f2. Si la position y est supérieure à f1, cela signifie que le clic est au-dessus de la diagonale f1. Dans ce cas, l'algorithme décrémente l'indice de colonne de 1 pour déplacer la cellule sélectionnée vers la gauche. Si la position y est inférieure à f2, cela signifie que le clic est en dessous de la diagonale f2. Dans ce cas, l'algorithme décrémente à la fois l'indice de colonne et l'indice de ligne de 1 pour déplacer la cellule sélectionnée vers le haut et vers la gauche.

Finalement, l'algorithme renvoie les coordonnées (colonne et ligne) de la cellule hexagonale sélectionnée en ajustant un peu les coordonnées.

Malgré certaines imprécisions surtout sur les côtés gauches de chaque cases, cet algorithme est le plus précis que nous ayons obtenus…

Prédicat 'occupé' des workers et bâtiments :

Afin de faciliter la distinction des workers & bâtiment entre lorsqu'ils effectuent une tâche et lorsqu'ils sont disponible, nous avons décidé d'utiliser un prédicat "occupied" qui indique lorsqu'il est passé à "true" que le worker/buildings en question n'est pas disponible pour une quelconque autre action que celle qu'il effectue actuellement. Cela est représenté visuellement pour les workers par leur couleur qui est passée en rouge.

Calcul du workers adapté le plus proche:

Pour la plupart des actions implémentées, nous avions besoin de trouver le workers adapté le plus proche afin qu'il se rendent là où cette action se déroulaient. C'est pourquoi nous avons créé la fonction 'GetNearestWorker()' qui possède 2 versions, chacune utilisant le même algorithme principal mais avec certains tests plus précis pour la deuxième.

En effet pour chacune des deux versions voici l'algorithme utilisé :

On définit d'abord une variable "shortest" équivalent à l'infini, puis on parcourt la liste des workers présent dans le jeu, et pour chacun d'entre eux, on calcul le chemin le plus court jusqu'a la case sur laquelle on veut qu'il se rende pour effectuer l'action, puis on calcul la longueur de ce chemin, et si cette dernière est inférieure à la distance "shortest" alors cette dernière est définit comme la plus courte (elle remplace "shortest"). Puis à la fin de la boucle, on récupère le worker associé à la distance la plus court "shortest" et on le désigne comme étant le worker le plus proche de la case sur laquelle se trouve l'action.

Pour la deuxième version l'algorithme est globalement le même mais cette fois-ci on ne calcule pas la distance pour tous les workers, mais seulement ceux dont le role correspond à la tâche à effectuer passée en paramètre. Les différentes tâches possibles sont définit dans le type enuméré 'Actions', permettant de tester plus efficacement qu'avec des types string les actions voulues.

Déplacement des workers :

Le déplacement d'un worker va faire intervenir la classe WorkerMoveCtrl et WorkerModel. Le workerModel va contenir toute les informations actuelle du worker et WorkerMoveCtrl va mettre a jours petit à petit la position sur la map du worker.

Ici la seul constante dans ce Thread est le taux de rafraichissement entre chaque action (48 actions par seconde). Lorsqu'un joueur aura cliqué sur une cellule, le Boutton Move va devenir disponible et le joueur pourra cliquer dessus. Lorsque le boutton est activé, il lance le Thread de WorkerMoveController qui prend la coordonée de la case sélectionner. A ce moment là le joueur va sauté de case en case jusqu'à sa destination. Pour savoir sur qu'elle case le worker doit se déplacer, un algorithme de dijkstra à été implémenter. Avant de sauté d'un case à l'autre, le Thread va calculer les positions successives sur l'écran du worker et les enregistré dans une variable de worker afin que WorkerView puisse dessiné l'animation de déplacement

Les conditions pour que le déplacement soit fonctionelle est que l'on ne puisse pas cliqué en dehors de la map. Pour cela la fonction qui sélectionne une cellule a partir d'un clique doit toujours renvoyer une cellule qui existe.

Récolte des ressources :

L'action principale de ce jeu est l'action "Collect" qui permet au joueur d'envoyer un worker récolter les ressources d'un terrain ou d'un bâtiment si il clique sur une case le permettant.

Pour se faire, on crée un nouveau thread "WorkerCollect()" prenant en argument un worker et la cellule destination, le worker étant calculé grâce à la fonction précédemment décrite 'GetNearestWorker()' à laquelle on à passé la tâche "Collect". Le thread alors instantié signale le joueur comme occupé, puis déplace jusqu'à la case désignée, en utilisant l'algorithme décrit dans le déplacement des workers, puis fait appel selon le cas (présence d'un bâtiment ou non sur la case) aux fonctions 'Collect()' ou 'Harvest()' de la classe 'GameModel()', qui fonctionnent plus ou moins de la même manière :

Les fonctions regardent quelle quantité de ressources est disponible sur la case désignée, puis si cette dernière est supérieure à la quantité maximale qu'un worker peut récolter alors il récolte tout ce qu'il peut, et si ce n'est pas le cas alors il récolte l'entièreté des ressources de la case. La récolte constitue simplement en un transfert de ressource d'un inventaire à l'autre, entre l'inventaire de la case et l'inventaire du worker, selon la quantité indiquée.

Création de bâtiments :

Tout au long du jeu une autre des actions possible pour le joueur est la création de bâtiment, pour se faire il y a plusieurs possiblités, toutes gérées par la classe 'ActionBuild()' du package Controler, soit le joueur a cliqué sur une case "City", auquel cas on lance un thread "WorkerBuildTraining()" un peu particulier, soit il a cliqué sur une case produisant des ressources, auquel cas on lance un thread plus classique "WorkerBuildProduction()". Ces deux threads produisent le même résultat mais avec un cheminement un peu différent :

- Le thread "WorkerBuildProduction()" récupère simplement le worker le plus proche ayant le rôle correspondant au type de la case (correspondance logique par type de ressource comme expliqué plus tôt), puis lui indique le chemin jusqu'à la case, et enfin crée un nouveau bâtiment du type lié à la case.
- Le thread "WorkerBuildTraining()" crée quant à lui un nouveau modèle ainsi qu'un view et enfin le controller spécifique "BuildingChoiceCtrl()". Qui ouvre une Jframe grâce à laquelle le joueur pourra séléctionner le type de bâtiment qu'il veut construire en utilisant les boutons mis à sa disposition, et selon le bouton cliqué, un thread "BuildingTrain()" est instantier avec en paramètre le type de bâtiment désiré. Puis le fonctionnement est le même que pour "WorkerBuildProduciton()", le worker désigné se rend sur la case en question et construit le bâtiment.

Pathfinding :

Nous avons évoqué plus tôt la notion de chemin le plus court entre deux cases, mais afin de détèrminer ce dernier nous avons implémenté une version de l'algorithme de Dijkstra aussi appelé "algorihtme du plus court chemin" :

On commence par prendre en entrée deux points, celui de départ et de fin du chemin que l'on souhaite déterminer.

L'algorithme fonctionne en assignant une distance de départ à chaque noeud, en utilisant une file de priorité pour maintenir les noeuds non visités, et en itérant à travers les noeuds adjacents pour mettre à jour la distance à partir du noeud de départ jusqu'à chaque noeud voisin.

L'algorithme commence par initialiser un tableau de points représentant le graphe, une HashMap pour stocker les distances à partir du point de départ et une autre HashMap pour stocker les noeuds précédents dans le chemin le plus court. Les distances sont initialisées à une valeur maximale pour tous les points, sauf le point de départ qui est initialisé à une distance de 0.

Ensuite, les noeuds sont ajoutés à la file de priorité en utilisant leur distance actuelle à partir du point de départ comme clé de priorité. L'algorithme itère ensuite sur les noeuds adjacents pour mettre à jour les distances en vérifiant si la distance à travers le noeud actuel jusqu'au noeud voisin

est plus courte que la distance actuelle stockée pour ce noeud. Si c'est le cas, la distance est mise à jour et le noeud précédent est enregistré dans la HashMap.

Le processus se poursuit jusqu'à ce que le noeud d'arrivée soit atteint ou que tous les noeuds aient été visités. Si la distance à partir du point de départ jusqu'au point d'arrivée est infinie, cela signifie qu'il n'y a pas de chemin entre les deux points et la fonction renvoie null. Sinon, la fonction retourne la liste des points représentant le chemin le plus court entre le point de départ et le point d'arrivée.

Cette implementation de l'algorithme fait appelle à la partie fonctionnelle de Java, qui bien que n'étant pas forcément optimale nous a semblée intéréssante à utiliser dans le cas présent.

Décision des boutons d'actions activés :

A chaque click du joueur sur une case, on détermine lesquels des boutons d'actions seront activés ou non, afin d'indiquer au joueur visuellement lesquelles des intéractions avec le jeu il pourra effectuer :

- Le bouton d'action Build est rendu cliquable si la case désignée est une case produisant des ressources ou alors si elle est de type "City. Auquel cas un nouveau thread "WorkerBuildProduction()" ou "WorkerBuildTraining()" est instantié.
- Le bouton d'action LevelUp est rendy cliquable si la case désignée contient un worker ou un building, même si techniquement une case ne peut pas en contenir on regarde en fait parmis les listes de workers et buildings si au moins l'un d'entre eux possède les même coordonnées courantes que la case désignée. Puis on fait isntantie selon le cas un thread "BuildingUpgrade()" ou "WorkerUpgrade()".
- Le bouton d'action Move est rendu cliquable si au moins un worker du jeu est de type "Knight", auquel cas il instantie un thread "WorkerMove()" pour l'envoyer sur la case désignée.
- Le bouton d'action Collect est rendu cliquable si la case désignée produit un quelconque type de ressource, puis instantie un thread "WorkerCollect()".
- Le bouton d'action Train est rendu cliquable si la case désignée contient un bâtiment d'entraînement (si ce dernier partage les mêmes coordonnées que cette dernière si on veut être plus précis). Puis instantie un thread de création de nouveau worker "BuildingTrain()".
- Le bouton d'action Expand est rendu cliquable si la case désignée est d'un type différent du type "City" et si c'est le cas instantie un thread "WorkerExpand()" qui envoie le worker le plus proche sur cette dernière afin d'en changer le type.
- Le bouton d'action Shop est rendu cliquable si la case désignée est celle sur laquelle se trouve le bâtiment de type "Shop", et si c'est le cas instantie un nouveau set (Model+View+Controler) lié au shop "ShopModel" "ShopView" et "ShopCtrl".
- Le bouton d'action Drop est rendu cliquable si la case cliquée est celle sur laquelle se trouve le bâtiment principal "CityHall", auquel cas instantie autant de thread "WorkerDrop()" qu'il y a de workers (autres que "Knight") dans le jeu, afin que ces derniers transfert toutes les ressources dans leur inventaire à celui de la ville.
- Le bouton d'action Objectifs est tout le temps cliquable car il ne permet au joueur que d'avoir un aperçu de sa progression quant aux différents objectifs de jeu, en instantiant une nouvelle view "GoalsView" liée au modèle déjà existant "GoalsModel".

Création de workers :

Tout au long du jeu le joueur peut décider de créer de nouveau workers grâce aux bâtiment d'entraînement, ce qui crée un nouveau set (Model+View+Controler) grâce auquel le nouveau worker peut être librement utilisé. Ce dernier est ajouté aux listes déjà existantes (pour le model et la view) et un nouveau thread "WorkerCtrl" est instantié.

Achat & Vente de ressources :

La ressource "Gold" n'est accessible au joueur que lorsqu'il décide de vendre des ressources, pour se faire il doit cliquer sur le shop et choisir un montant de ressources qu'il peut soit

décider de vendre s'il a besoin de plus d'argent -qui est une ressource nécéssaire aux actions qu'il voudra effectuer-, soit d'acheter si jamais la récolte d'autres ressources est trop compliqué ou qu'il ne désir pas envoyé de worker récolter des ressources.

Pour cela on instantie à chaque fois un nouveau Model une nouvelle View ainsi qu'un nouveau controler pour gerer les achats et ventes du joueur. La View permet au joueur de specifier sa requête, suite à quoi selon le bouton cliqué (buy ou sell) l'actionListener associé "ActionBuy" ou "ActionSell" s'occupe de faire l'ajout ou la suppression de ressources dans l'inventaire du joueur (de la ville) et la suppression ou l'ajout d'argent dans ce meme inventaire. Chacun de ces boutons permet aussi de fermer la fenêtre instantiée pour l'action de manière indépendante (non liée à la fermeture de la fenêtre de jeu).

Affichage de l'interface de shop :

Pour l'affichage de l'interface shop une petite fenêtre apparait au milieu de l'écran. Ici c'est shop qui hérite de JFrame qui apparait à l'écran. Comme pour l'affichage du panneau de contrôle la fenêtre utilise un grid bag layout. Pour commencer ont créé 3 slider qui correspondent aux trois ressources du jeu. Ces sliders possèdent un change listenner qui appelle la procédure stateChanged() quand l'utilisateur les fait varier. On leur ajoute aussi des labels pour les identifiés. On ajoute aussi deux label Buy price et sell price qui nous indique la valeur des ressources sélectionnée à l'aide des slider. La valeur de ses slider est ajuster quand l'utilisateur fait varier les sliders à l'aide de stateChanged().
On crée ensuite deux boutons buy et sell. Buy devient griser si l'utilisateur essaye d'acheter des ressources qui coute plus que ça quantité d'or. Ce changement est effectué en utilisant la fonction setEnable() du bouton qui est appelé par stateChanged()

Capacité d'inventaire spécifiques aux entités de jeu :

Pour chacune des entités du jeu, nous avons définit une quantité maximale et un type précis de ressources correspondant au type du terrain ou worker associé, cette quantité n'est pas définit autrement qu'officieusement en ne faisant des tests pour permettre les transferts de ressources autres que celles souhaitées.

Dépôt des ressources des workers dans l'inventaire de la ville :

Grâce au bouton d'action "Drop", le joueur peut de décider de trasnférer toutes les ressources des inventaires des workers disponibles vers l'inventaire de la ville. Pour cela on instantie seulement autant de thread "WorkerDrop" qu'il y a de workers concernés, mettant leur prédicats "occupied" à "true" et les déplaçant vers le "CityHall" et les faisant transférer leur inventaire grâce aux fonctions adéquats "WorkerDropsInventory()". Ensuite les prédicats "occupied" des workers sont remis à "false" et ils sont de nouveaux disponible pour effectuer d'autres tâches.

Dépôt automatique des ressources :

De la même manière que le dépôt de ressource initié par le joueur, le dépôt automatique quant à lui n'est instantié que lorsqu'un des workers se retrouve a avoir son inventaire plein, ce qui est calculé en fonction de l'attribut "max_amount", grâce à la fonction "InventoryIsFull()" qui renvoie "true" si cet attribut est egal au nombre de ressource que l'inventaire contient.

Cette fonction prédicat est utilisée dans le thread "WorkerCtrl" pour le worker correspondant, et si il renvoie "true" alors un nouveau thread "WorkerDrop" est instantié.

Actualisation constante de l'affichage :

Pour l'actualisation constante de l'affichage, nous utilisons un thread instantié dans le constructeur du game controller, dans la classe "GameCtrl", qui sert à constamment demander au

Jpanel de “MapView” de s’actualiser grâce à la fonction “repaint()” puis de se redessiner en ayant bien actualisé toutes les informations grâce à la fonction “revalidate()”.

Augmentation constante des ressources de certains terrains :

Dans le contrôleur des terrains “CellCtrl” instantié dans le constructeur de “MapCtrl”, la fonction “CellBehaviour()” est appelée et permet selon le type du terrain d’incrémenter plus ou moins rapidement les ressources contenu par le terrain. Et si il y a un bâtiment sur la case concernée, alors les ressources sont incrémentées d’autant plus rapidement, et placées dans l’inventaire de ce dernier.

Spawn aléatoire d’ennemies :

Dans le constructeur du “GameCtrl”, le thread “EnnemiesSpawn” est instantié, et ce dernier permet de faire apparaître des ennemies sur les cases prédéfinies comme des “spawn_cell” lors de la création du monde dans le constructeur de “MapModel”, ces dernières étant placées dans une liste appartenant au “MapModel”, ce thread peut y accéder et aléatoirement faire apparaître un nouvel ennemis toutes les 30-60 secondes. Instantiant un nouveau thread “WorkerCtrl” pour chacun d’entre eux.

Interface d’objectifs :

Pour l’affichage de la fenêtre des objectifs on utilise la classe GoalsView. Celle est hérité d’une JFrame. Dans cette fenêtre il y a 3 Jpanels :

- Une pane qui contient les objectifs principaux elle à un box layout et contient les objectifs TrainingBuilt, Collect Ressources, ExpandSlots, Production built et KilledEnnemies.
- Une pane qui contient les objectifs secondaires elle à un box layout et contient les objectifs SoldRessources, BoughtRessources et Trained Workers.
- Une pane qui contient les deux autres

Ces objectifs sont définis Types.Gloals. Pour visualiser ces objectifs on utilise des JProgresseBar avec l’avancement du dit objectif à l’aide du GoalsModel.
Affichage de l’interface de construction :
Pour l’interface building on affiche un une fenêtre de la classe BuldingChoiceView qui hérite de JFrame. Dans cette Jframe on ajoute un Jpanel qui à un grid bag layout. Dans cette JFrame on à tout d’abord un label d’explication. Puis il y à 4 boutons qui correspond au bâtiment qui peuvent être construit sur cette case. Chacun de ces boutons possèdent une action listener implémenter par BuildingChoiceCtrl.

Déplacement semi-aléatoire des ennemies :

Chacun des threads “WorkerCtrl” test si l’ennemie concerné n’est pas mort, et si il ne l’est pas alors il test si un des workers se trouve dans un rayon de 2 cases par rapport a lui, grâce à la fonction “EnnemySearchForCitizen()” qui prend en argument les cases voisines de l’ennemie (representant son champ de vision proche) et qui regarde si les cases voisines de ces dernières sont occupées par un worker ou non, retournant si c’est le cas une pair avec un booleen “true” et la coordonnée de la case sur laquelle se trouve le worker, et une pair avec un boolean “false” et une coordonnée nulle sinon.

Suite a quoi si le boolean est “true” alors l’algorithme entame une poursuite du worker en question :

- On récupère le citoyen se trouvant aux coordonnées présentent dans la paire indiquée. Puis on initie un booleen “citizen_caught” et deux integers “pursue_time” & “pursued_time” qui serviront de conditions pour la suite de l’algorithme, respectivement à “true”, une valeure entière aléatoire entre 5 et 10 et la valeur entière 0.
- Tant que le booleen “citizen_caught” n’est pas “true” ou que “pursued_time” n’est pas égal à “pursue_time”, alors on récupère la position du worker repéré plis tôt, on calcule le chemin qui

nous en sépare et on se rend sur la première case de ce chemin qui n'est pas notre case actuelle (en tant qu'ennemie).

- Avant cela on verifie si notre position est egale à celle du joueur pourchassé, et si c'est le cas on passe le booleen "citizen_caught" à "true".
- Enfin on fait bouger l'ennemi grâce à la fonction "MoveTo()" et on incremente le compteur "pursued_time".
- Finalement on attend 1 seconde avant de relancer le test de boucle.

Suite à cela, si le booleen "citizen_caught" est "true" alors le controler appel la fonction "EnnemyAttacksWorker()" pour effectivement faire se rencontrer le worker et l'ennemie concernés par cette rencontre, et appliqué les règles de jeu qui s'en suivent.

Et sinon l'ennemie est juste appelé à se déplacer d'une case choisit aléatoirement parmis les cases voisines de sa position actuelle. Puis le thread est mit en pause pour 1.5 secondes.

Déplacement spécifique des chevaliers :

Le bouton d'action Move permet au joueur de contrôler librement les workers ayant pour rôle "Knight", afin de pouvoir organisze au mieux sa défense contre les ennemies environnants. Pour se faire un click sur le bouton instantie un nouveau thread "WorkerMove" qui cherche le worker "Knight" disponible le plus proche et l'envoie se déplacer sur la case désignée.

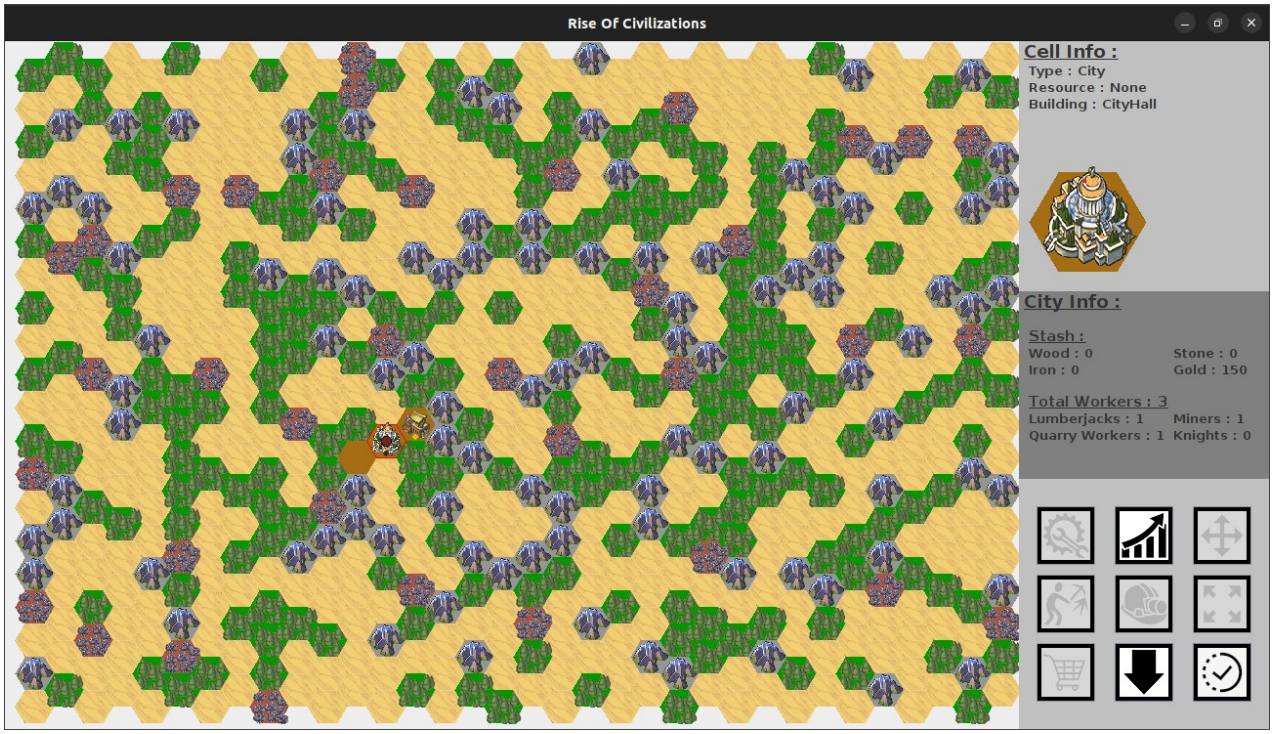
Affrontement Worker/Ennemies :

Lorsqu'un ennemie rencontre un worker (ou vice-versa) alors on test si le worker est un "Knight" ou non. Si c'est le cas, alors le worker comme l'ennemie prennent des dégats grâce aux fonctions "TakeDamage()" respectivent aux deux classes "WorkerModel" et "EnnemyModel", et si le type du worker est différent, alors il est le seul à subir des dégats.

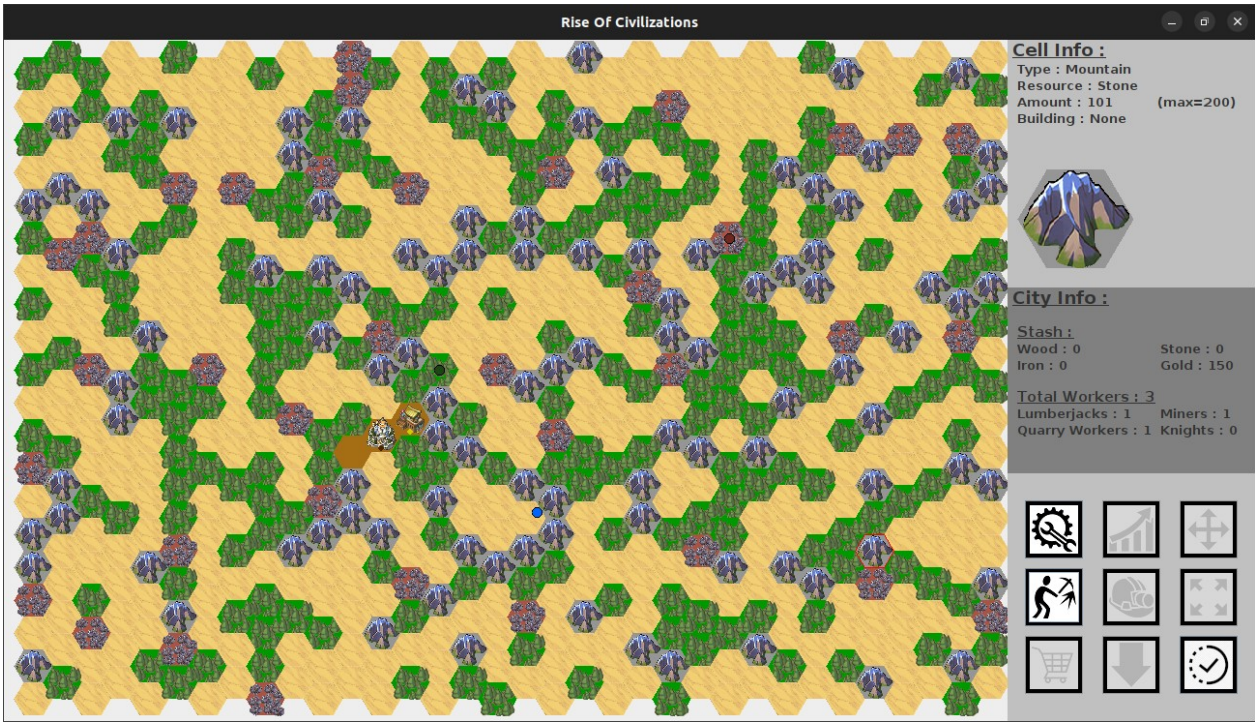
Résultat -

Pour le résultats final, voici quelques rendus visuels du jeu :

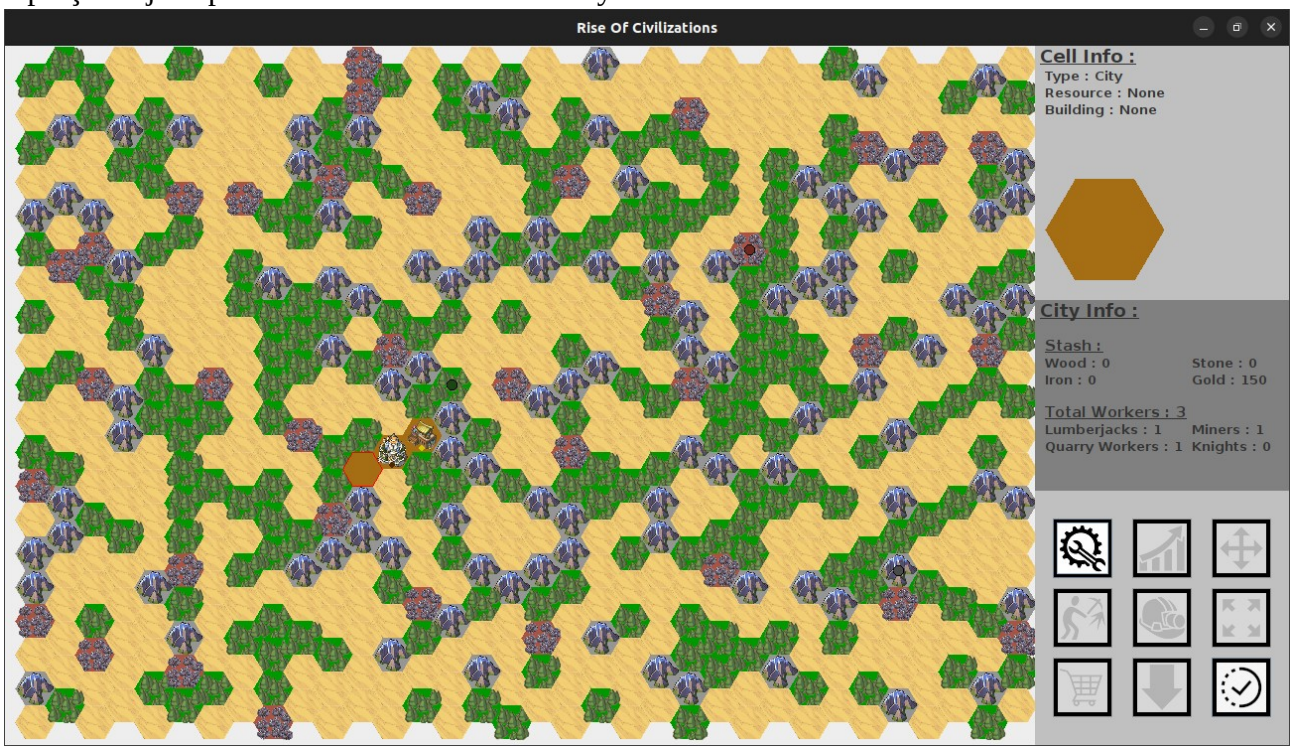
Aperçu du jeu a son lancement →



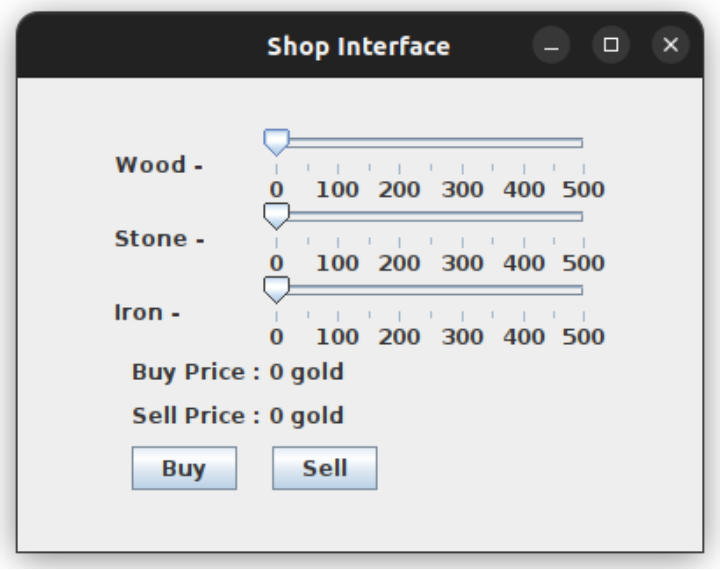
Aperçu du jeu après déplacement de 2 workers et d'un clique sur une case "Mountain" →



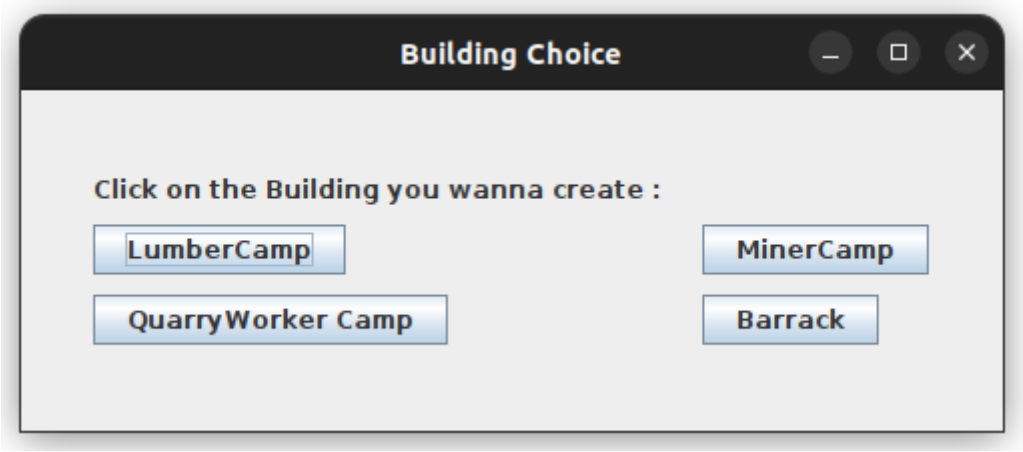
Aperçu du jeu après un click sur une case “City” →



Voici maintenant les aperçus des interfaces annexes, à commencer par l’interface des objectifs et du shop →



Et enfin voici un aperçu de la dernière interface présente dans le jeu, la view de choix de construction →



Documentation Utilisateur -

Pour lancer le jeu il suffit d'import le projet sous l'IDE Eclispe, puis de lancer le jeu via la classe "Main.java". Puis pour jouer le principe est simple, tout se fait en choisissant une des cases de la grille et en cliquant ensuite sur le bouton d'actions correspondant.

Par exemple voici un scenario simple, je veux recolter les ressources d'une case et les deposer dans l'inventaire de ma ville :

- Click sur la case de ressource souhaité → le bouton "Collect" s'active.
- Click sur le bouton "Collect" → le worker du type voulu se déplace jusqu'à la case clickée et récolte les ressources.
- Click sur la case où se trouve le joueur → le bouton "Drop" s'active.
- Click sur le bouton "Drop" → le worker présent sur la case cliquée se rend au "CityHall" pour y transférer son inventaire.

Et c'est ce type de scénario qu'il faudra répéter pour pouvoir jouer au jeu, en choisissant les bonnes actions afin de résister aux attaques ennemis et de faire prospérer sa ville au mieux.

Une stratégie commune et simple est de créer une "Barrack" le plus vite possible afin de pouvoir résister aux attaques ennemies, puis de multiplier les cases "City" pour empêcher les ennemies de s'y rendre, tout en construisant les bâtiment de production voulus.

Pour sécuriser des bâtiments la stratégie est simple aussi, il suffit de constuire un bâtiment de production sur une case, puis d'en faire une case city, ainsi n'importe quel worker pourra s'y rendre pour récolter des ressources sans aucun danger.

Documentation Développeur -

Les principales classes à regarder sont globalement les controlers, car ce sont eux qui instantient les comportements voulus, et c'est en partant d'eux qu'on peut remonter jusqu'à toutes les informations nécéssaire à la compréhension complète du jeu.

Par exemple, pour prendre la fonctionnalité centrale du jeu, le click sur une case, il faut aller regarder la classe "CellCtrl" qui indique selon la case cliquée quels boutons sont activés. Puis une fois les boutons activé on veut savoir ce qu'ils appellent, pour cela il suffit de regarder dans "GameCtrl" quels 'actionListener' sont associé aux boutons voulus. Enfin on retrace facilement la cascade d'évènement qui s'en suit.

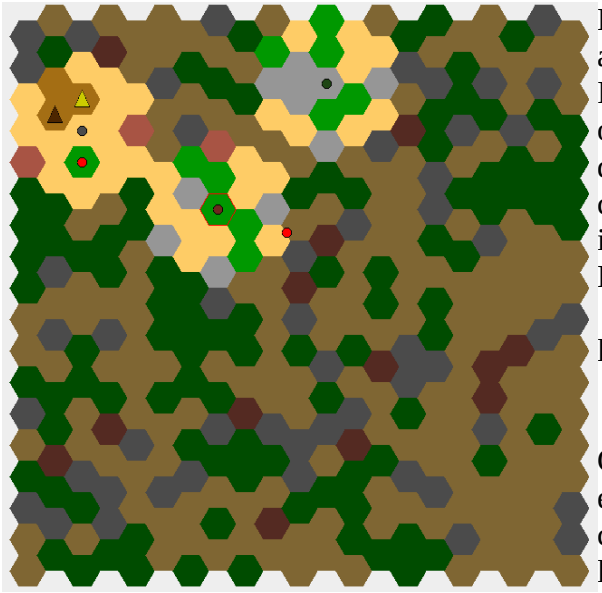
Une autre possibilité serait de vouloir savoir ce qu'il se passe si on ne fait rien. Dans ce cas il suffit de regarder dans "MapModel" comment le monde est initialisé, puis de regarder dans

“GameCtrl” les threads continus qui sont instantiés. Et grâce à cela on peut comprendre quels sont les comportements continus du jeu.

Enfin pour une compréhension plus approfondie des fonctionnalités du jeu, porter son attention sur les classes contenus dans le package ‘Thread’ est une bonne piste, car c’est là que sont contenus la majeur partie des actions implementées de manière brute, faisant appel au minimum de fonction nécéssaires.

Dernière information, si vous souhaitez changer le temps de jeu dirigez vous vers la classe “Main.java” du package ‘Main’ et changez les 2e et 3e parametre du GameCtrl (comme indiqué en commentaire).

Les aspects du jeu que nous aurions aimé implementer sont les suivants :
- le fog of war →cette fonctionnalité a été implementée avec un succès partiel, mais retiré car elle ralentissait trop le jeu et nous n’avions pas le temps de l’implémenter de manière plus élégante :



Lors du dévellopement du jeu un brouillard de guerre avait été implémenter.
Il avait pour objectif de grisé les cellules sur qu’elle on avait pas la vision et permettait aux ennemies en dehors de la vue des workers de disparaître. Pour cette fonctionnalitées on devait utilisé les informations contenue dans CellModel, EnnemyModel et WorkerModel.

On avait une constante représentait la distance à la qu’elle pouvait voir chaque worker.

A chaque fois qu’un worker changeait de Cellule on recalculait pour toute les Cellules si elles étaient visible ou non et à chaque changement de cellule d’un ennemy si celui-ci devenait visible pour le joueur. Une amélioration facile a faire pour évité des calcules inutiles aurait été pour les ennemies de définir leur visibilités en fonction de si la cellules sur la qu’elle ils sont est visible.
Une autre amélioration aurait été a chaque déplacement calculer seulement les nouvelles cellules visible et pas recalculer pour toutes les cellules. Cependant Cette fonctionnalitée a été retiré avec l’ajout d’image sur les cellules.

- Déplacement plus fluide des ennemies → à l’image du déplacement des ennemies, nous aurions aimé pouvoir rendre le déplacement des ennemies plus agréable et fluide, ou au moins le rendre moins brutal. Mais encore une fois manque de temps onlige à faire des choix, nous avons préféré le garder tel quel et qu’il fonctionne comme voulu plutôt que son affichage nous apporte plus de soucis à corriger.

- Accentuation de l’aspect ‘combat’ → idéalement nous aurions aimé pouvoir exploiter un peu plus les ennemies, avec l’idée de créer une base ennemie qui grandirait aussi de son côté et qui attaquerait même la base du joueur, mais cela faisait trop de nouvelles choses à implémenter.

- Génération d’un monde plus grand →une des fonctionnalités initialement prévue était la possibilité de naviguer dans un monde plus vaste qui nécéssiterait notamment l’utilisation d’une minimap (que nous avons remplacé par la CellInfoView au final).

Conclusion & Perspectives -

Nous avons finalement réussi à impléménter la majeur partie des fonctionnalités que nous souhaitions retrouver dans notre jeu. Et malgré plusieurs soucis innatendus nous avons réussi dans la globalité à impléménter un jeu de gestion que nous estimons jouable (bien que pouvant être encore visuellement bien amélioré) qui implémente une grande partie des fonctionnalités que l'on pourrait retrouver dans les jeux du même style, en évidemment moins poussé.

Parmis les complications, le choix d'une grille hexagonale nous en a apporté plusieurs, notamment pour l'algorithme de recherche des coordonnées d'un click ainsi que ceux d'affichage de la grille, mais ce fut un exercice intéressant. Se rajoutent dans cette catégories les soucis liés au multi-threading, qui nous ont plusieurs fois donné du fil à retordre. Et notamment notre organisation quant au code fut quelques peu chaotique, et nous a à plusieur reprise coûé un temps précieux...

Nous avons eu un peu de mal à nous organiser en tant que groupe mais l'exercice fut tout de même intéressant et nous aura permit de savoir surmonter ces problèmes pour l'avenir. Pour ce qui est des possibilités d'évolution de ce projet, elles sont nombreuses, à commencer par l'implémentation de toutes les fonctionnalités énoncées à la fin de la 'Documentation Développeur'. Mais aussi les amélioration au niveau du Game Design et d'un potentiel Lore pour le jeu.