

YARTS

Yet Another Real Time Strategy (game)

powered by **libGDX**



Auteurs :

Enseignantes :

Sol Rosca, Edwin Claude, Nathan Latino

Rizzotti Aïcha, Punceva Magdalena

He-Arc - développement logiciel

projet Java 2^{ème} année

19 juin 2019

Table des matières

1. Introduction	1
2. Contextualisation	1
2.1. Jeu de stratégie en temps réel	1
2.2. Java	2
2.3. LibGDX	2
3. Planification	3
3.1. Objectifs	3
3.2. Organisation et logistique	4
3.2.1. Trello	4
3.2.2. TeamViewer	5
3.3. Répartition des tâches	5
3.4. Planning	6
4. Conception	7
4.1. Techniques de programmation	7
4.1.1. Extreme programming	7
4.1.2. Développement itératif	7
4.2. Git flow	8
4.3. Spécifications	9
4.3.1. Jeu	9
4.3.2. Carte	9
4.3.3. Entités	10
4.3.4. Récolte	11
4.3.5. Inputs	11
4.3.6. Déroulement d'une partie	12
4.4. Conventions	12
4.5. Premier diagramme de classes	12

5. Réalisation	14
5.1. Architecture	14
5.1.1. Vue d'ensemble	14
5.1.1.1. Framework	16
5.1.1.2. Client	16
5.1.1.3. Avantages	18
5.1.2. Approche OOP	18
5.1.3. Approche OOP + composition (ECS naïf)	20
5.1.4. Approche ECS	21
5.1.4.1. Component	21
5.1.4.2. Entity	22
5.1.4.3. System	22
5.1.4.4. Implémentation	22
5.1.5. Conclusion sur l'architecture	24
5.1.5.1. Performance	24
5.1.5.2. Complexité du code	24
5.1.5.3. Maintenabilité	25
5.2. Mécanismes	25
5.2.1. Debugging et configuration	26
5.2.2. Champ de vue, portée et état	30
5.2.3. Notifications	35
5.2.4. Pathfinding	36
5.2.5. Brouillard de guerre	36
5.3. Design	38
5.3.1. Map	38
5.3.2. Sprites	40
5.3.3. Animations	41
5.3.4. Gestion des ressources	42
5.3.5. Interface utilisateur	42
5.3.5.1. Mini-map	43
5.3.5.2. Panel actions et sélections	43
6. Récapitulatif	44
6.1. Objectifs	44
6.2. Bugs	45
6.3. Améliorations	46
7. Conclusion	46

8. Ressources	47
8.1. papier	47
8.2. internet	47
8.2.1. LibGDX	47
8.2.2. Développement de jeux en général	47
8.2.3. Etat de l'art	47
8.2.4. ECS	48
8.2.5. Steering behaviors	48
8.2.6. Pathfinding	48
8.2.7. Brouillard de guerre	48
8.2.8. Level Design	48
8.2.9. Ui Design	49
8.2.10. Repo des mini projets faits pour se familiariser avec LibGDX	49
9. Annexes	49

1. Introduction

Dans le cadre du Projet 2.2 de la formation DLM de l'He-Arc de Neuchâtel il est demandé de réaliser une application graphique avec le langage Java par groupe de trois personnes. Le projet choisi est un jeu de stratégie en temps réel sobrement nommé Yet Another Real Time Strategy (game) ou *YARTS*¹.

Ce document tente de relater l'histoire de YARTS. Comment ce qui aurait pu n'être qu'une formalité en périphérie d'un semestre déjà bien chargé est devenu l'objet d'une passion ainsi qu'un excellent professeur pour une gamme très large de points clé de la formation d'un ingénieur en développement logiciel.

2. Contextualisation

Ce chapitre est un complément à l'introduction. Il vise à poser les fondations contextuelles du projet ainsi que de définir une base de communication pour que le lecteur puisse se faire une idée de comment les rédacteurs conçoivent certains éléments clé du contexte dans lequel le projet s'inscrit.

2.1. Jeu de stratégie en temps réel

Traditionnellement les jeux de stratégie se jouent au tour par tour. Aussi bien sur plateau qu'avec des cartes. Les jeux de stratégie sont légion, le plus célèbres d'entre eux étant probablement les échecs. Tous ces jeux ont en commun la nécessité pour atteindre la victoire de réfléchir à une stratégie et de l'adapter au fur et à mesure que le ou les autres joueurs tentent eux aussi de développer leur propre stratégie. En général dans ce genre de jeux, le temps n'est pas un facteur limitant. Par exemple, aux échecs, un joueur aura beau être capable de déplacer très rapidement son fou, ça n'aura aucun impact sur l'issue de la partie.

L'avènement de l'informatique et la puissance de calcul des ordinateurs a permis de faire évoluer la diversité dans les jeux de stratégie en y ajoutant la notion de temps réel. Dans ce genre de jeu, les joueurs ne jouent plus chacun à leur tour mais jouent maintenant de façon concrète. Jouer vite est une qualité qui peut se mesurer en nombre d'actions par seconde et même si ce n'est pas forcément le joueur le plus rapide qui gagne automatiquement la partie, la capacité d'exécuter rapidement un grand nombre d'actions peut donner des avantages décisifs et en fonction de la situation, cela devient un critère tout aussi important que la capacité d'élaborer une stratégie solide ou celle de s'adapter rapidement.

1. Yet Another Real Time Strategy (game)

Pour illustrer ça, imaginons que les échecs se jouent en temps réel, les joueurs ne doivent plus attendre leur tour et peuvent jouer autant qu'ils le veulent. Les règles sont toujours les mêmes, et les pièces bougent de la même façon mais maintenant, un joueur doit non seulement réfléchir plus rapidement à comment adapter sa stratégie mais il doit également faire preuve de dextérité pour bouger rapidement ses pièces. En effet dans cette variante on peut imaginer un joueur rapide qui dans l'absolu est capable de déplacer trois pièces en une seconde et un autre moins rapide qui n'est capable d'en bouger que deux dans le même interval de temps. En conclusion, un bon joueur de cette variante imaginaire des échecs ne sera pas forcément le plus fin stratège mais celui qui combine le mieux stratégie, capacité d'adaptation, vitesse d'exécution et dextérité.

Ce genre de jeux de stratégie sont ce qu'on appelle communément des jeux de stratégie en temps réel (STR en français et RTS pour Real Time Strategy Game en anglais).

Le terme "jeu de stratégie en temps réel" est utilisé pour la première fois en 1992 pour désigner le genre du jeu *Dune II* basé sur le roman éponyme de Frank Herbert. Depuis, le genre a beaucoup évolué mais principalement graphiquement. En effet, bien que la définition précise fasse l'objet de débats, les jeux de stratégie en temps réel sont traditionnellement définis par les termes "**récolter**", "**construire**" et "**détruire**" en plus d'être des jeux où l'action se déroule en temps réel entre les différents participants.

En partant des trois termes "récolter", "construire" et "détruire", on peut déduire que les intentions des joueurs sont axés autour du fait de "gérer des ressources", "développer une base" et "créer des unités" pour combattre l'adversaire. L'action se déroulant en temps réel, le joueur ne dispose que d'un temps limité pour gérer ses ressources et ses bases et contrôler ses unités, ce qui introduit les notions de rapidité en plus de la dimension stratégique comme expliqué précédemment et explique que le contrôle à la souris et au clavier soit généralement privilégié.

2.2. Java

Java est un langage haut niveau, orienté objet, avec un typage statique fort dont la syntaxe est proche du C++. Il est multi-plateformes et guidé par le principe du WORA (Write once, Run Anywhere).

2.3. LibGDX

LibGDX est un framework Java de développement de jeux vidéos. Portable et peu opinionné, il laisse une très grande liberté aux développeurs sur les outils à utiliser.

3. Planification

3.1. Objectifs

Dans le chapitre concernant les jeux de stratégie en temps réel ressortent les 3 mots clés qui définissent le genre: **récolter, construire et détruire**. Il est donc nécessaire que le MVP comporte ces trois éléments de gameplay. En plus de ces trois éléments les objectifs suivants sont considérés comme ceux à atteindre:

- Une carte ("plateau de jeu") sur lequel se passe l'action
- Menu principale
- Une interface utilisateur qui contient les éléments suivants:
 - Une mini carte qui donne une vue global au joueur de l'action
 - Affichage des ressources
 - Affichage de la population
 - Un panneau qui contient les actions spéciales qui affiche les actions que peut effectuer l'unité sélectionnée
 - Un "curseur intelligent" qui en fonction du type d'entité sélectionné, du type de clic effectué et de l'endroit de ce dernier effectue l'action adéquate.

Cette première itération du jeu se veut être un proof concept. L'ajout d'un adversaire, IA ou un autre joueur à travers le réseau n'est pas un objectif réaliste et n'est considéré que dans les objectifs secondaires au même titre que les points suivants:

- Brouillard de guerre
- IA qui gère un certain nombre d'actions sans l'intervention du joueur
- Multijoueur: Adversaire IA
- MultiJoueur: Adversaire humain
- Donner une âme au jeu:
 - Aspect esthétique
 - Contexte du jeu
 - Bruitages et musiques
- Rendre le jeu fun à jouer via:
 - Mécanisme de brouillard de guerre
 - Réflexion sur les unités du jeu
 - Ajout de nouveaux bâtiments

3.2. Organisation et logistique

3.2.1. Trello

Trello (trello.com) est une application permettant de gérer des projets et tâches en équipe. Très pratique pour la planification ainsi que l'organisation, cette application est basée sur la méthode "Kanban".

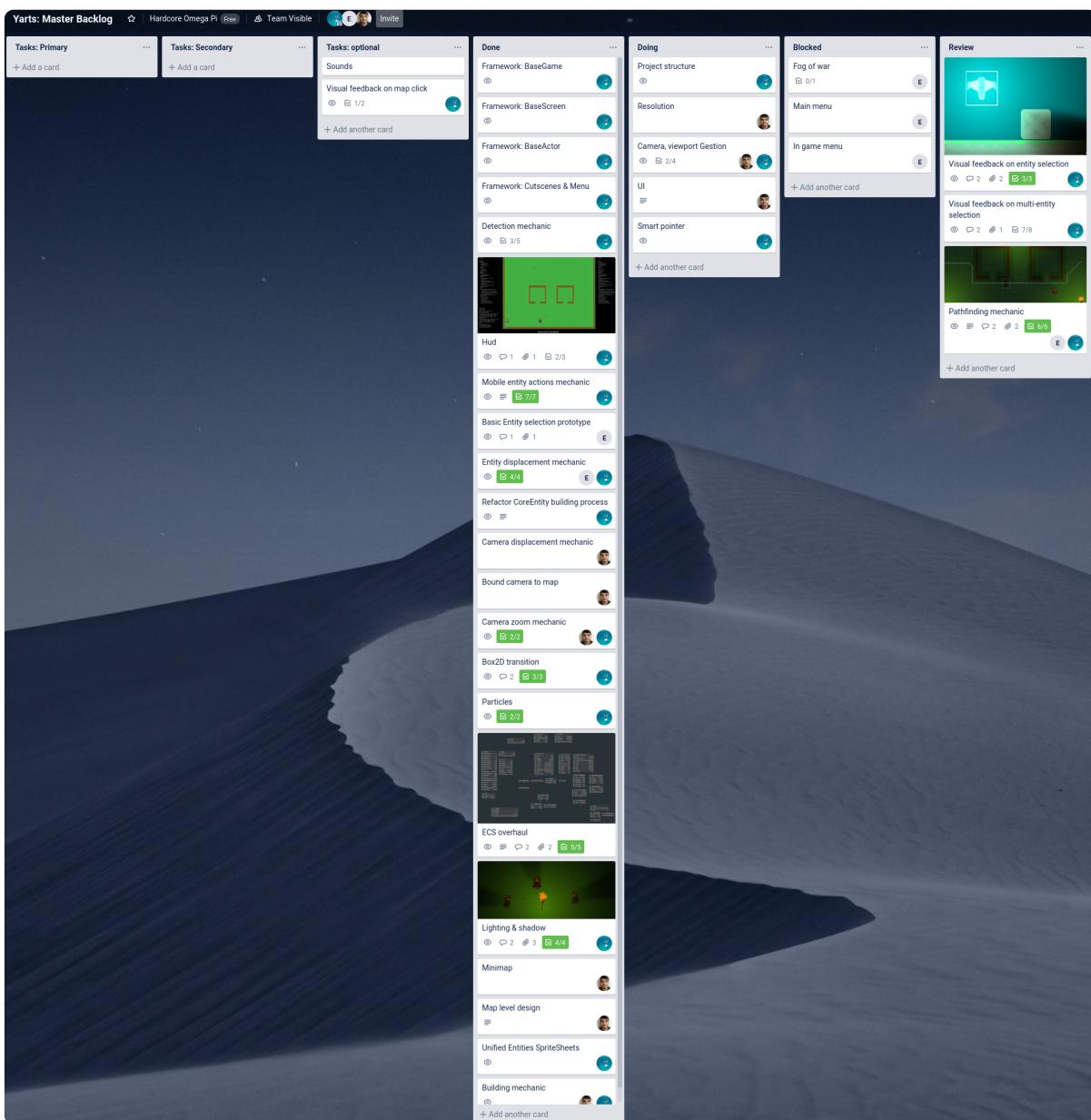


Figure 1: Trello

Chaque carte contient un espace organisationnel supplémentaire comprenant:

- Des liens utiles trouvés par les membres
- Des captures d'écran de l'avancement ainsi que divers médias
- Une todo découpant la tâche de la carte en sous-tâches élémentaires
- Une section commentaire

The screenshot shows a task board card with the following sections:

- Pathfinding mechanic**: A title with a gear icon.
- in list Review**: A link to the list.
- MEMBERS**: A section showing three user icons: E, a blue circle, and a green circle.
- Description**: A tab with a list of items:
 - Detection of obstacle, Recalculate of shot path .
 - <https://hackmd.io/RD5Qa7AjTBexRg0o4KFV4g>
- Usefull links:**
 - <https://www.redblobgames.com/>
 - <https://happyCoding.io/tutorials/libgdx/pathfinding>
 - <https://github.com/libgdx/gdx-ai/wiki/Steering-Behaviors>
 - <https://github.com/libgdx/gdx-ai/wiki>
 - <http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>
- Attachments**: A section showing two attachments:
 - iJloyl.png**: Added 5 May at 15:22. Options: Comment, Delete, Edit, Remove Cover.
 - lbEtQEn.png**: Added 4 May at 15:09. Options: Comment, Delete, Edit, Make Cover.
- Add an attachment**: A button to add more attachments.
- Implement pathfinding**: A checked item in a todo list.

100%	Hide completed items	Delete
<input checked="" type="checkbox"/> Visual repr. of the path		
<input checked="" type="checkbox"/> Obstacle-Detection		
<input checked="" type="checkbox"/> Path-computing		
<input checked="" type="checkbox"/> Move-from-isolated-env-to-main-project		
<input checked="" type="checkbox"/> Adapt-obstacle-detection		
<input checked="" type="checkbox"/> Include-decoration-as-obstacles		
<input checked="" type="checkbox"/> Fix-heuristic-function-dynamic-picker		
- Add an item**: A button to add new items to the todo list.

Figure 2: tâches

3.2.2. TeamViewer

TeamViewer (www.teamviewer.com) est un logiciel qui permet d'accéder à distance à une machine. Principalement utilisé pour du partage d'écran, cet outil s'est révélé précieux pour pouvoir faire de l'Extreme Programming.

3.3. Répartition des tâches

La répartition des tâche au sein des membres est la suivante:

- **Nathan Latino:** Interface utilisateur et graphisme
- **Edwin Claude:** Path finding, brouillard de guerre, menu principal
- **Sol Rosca:** Architecture, Framework et rédaction du rapport

3.4. Planning

Comme expliqué dans le chapitre sur l'initiation du projet, ce planning ne reflète pas la réalité des proportions du temps passé sur les divers points. Le travail sur le projet ne commence vraiment qu'à partir de la semaine du 22 avril.

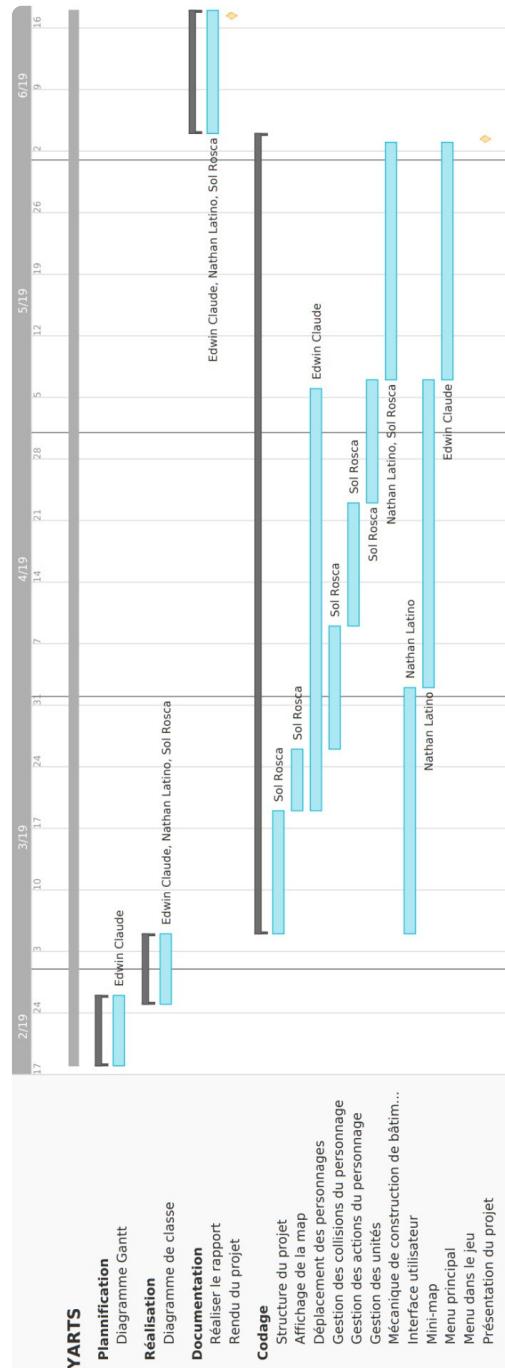


Figure 3: Planning initial

4. Conception

4.1. Techniques de programmation

4.1.1. Extreme programming

Plus particulièrement la partie "binôme" de cette pratique. Cette technique a l'avantage de pouvoir se pratiquer physiquement sur la même machine mais aussi à distances avec des outils comme TeamViewer.

- "Puisque la revue de code est une bonne pratique, elle sera faite en permanence"
- "Puisque la conception est importante, elle sera faite tout au long du projet (refactoring) "
- "Puisque la simplicité permet d'avancer plus vite, nous choisirons toujours la solution la plus simple "
- "Puisque la compréhension est importante, nous définirons et ferons évoluer ensemble des métaphores"

4.1.2. Développement itératif

Cette pratique s'est mise en place d'elle-même, elle n'est probablement pas le reflet de la pratique "officielle" et donc, voici notre interprétation:

Cette technique se base sur trois itérations successives pour implémenter une feature:

- **Une première itération naïve:** Ne prend pas en compte la qualité et l'optimisation du code mais se concentre sur la faisabilité.
- **Une seconde itération qui simplifie:** Nétoie le code et tatonne plusieurs patterns/architectures pour tenter de trouver les avantages et inconvénients.
- **Une dernière itération "présentable"** Consiste en une réécriture du code pour coller à un pattern/architecture qui est le fruit des leçons tirés des précédentes itérations.

4.2. Git flow

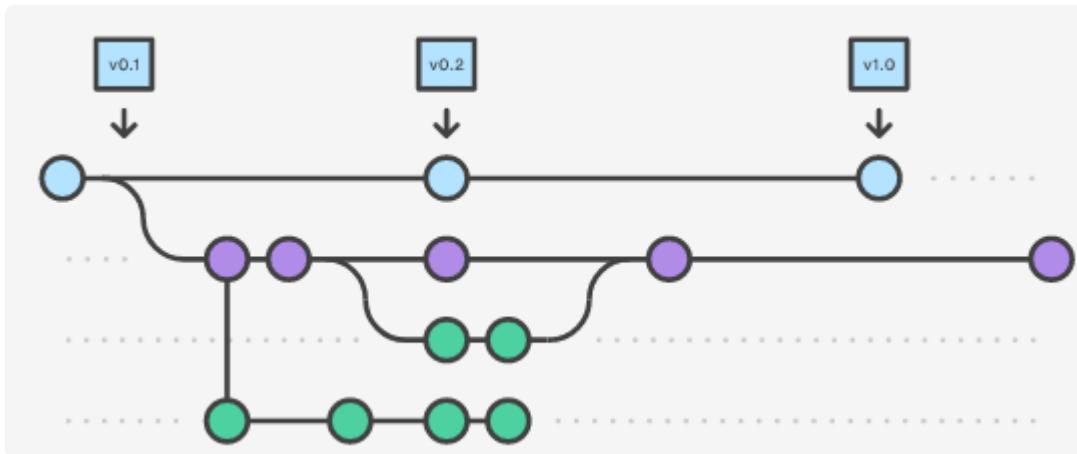


Figure 4: Git Flow

Le flow utilisé était "Feature Branche". Chaque nouvelle feature a sa propre branche et est merge sur la branche de développement (dev) à la compléction de son objectif. La branche dev n'est merge avec le master qu'une fois qu'un des objectif principal est atteint. Contrairement à l'illustration précédente, ce projet n'avait pas de versioning des milestones.

4.3. Spécifications

4.3.1. Jeu

- **Plateforme:** Desktop
- **Os:** Windows & Linux
- **Langage:** Java
- **Framework:** LibGDX
- **Genre:** Stratégie
- **Sous-genre:** Temps réel
- **Perspective:** 2.5D (Topdown)
- **Camera:** projection orthogonale
- **Controle:** Clavier + souris
- **Joueurs:**
 - 1 joueur humain (proof concept)
 - Dans un premier temps une faction adverse sans intelligence sera utilisé pour tester les capacités offensives.
- **Contexte scénaristique:** Aucun (le jeu sera dans un premier temps générique)
- **Population:**
 - Limitée
 - Nécessité de construire des "maisons" pour augmenter la limite
- **Économie:**
 - Récolte de ressources
 - Sert à finnancer la production de bâtiments et d'unités
- **Ressources:**
 - En quantité limitée
 - Un type de ressource unique
 - Recoltable sur la carte sur des points spécifiques

4.3.2. Carte

La carte est l'air de jeu. C'est une aggrégation de cellules sur un plan orthonormé d'une certaine taille. C'est sur la carte que se passe l'action du jeu.

- **Cellule:**
- Est repérée par une position dans un repère orthonormé
- Une cellule est un conteneur
- Une cellule peut contenir un unique élément parmi plusieurs types d'entités:
 - Du vide
 - Un obstacle naturel
 - Une ressource
 - Une entité appartenant à un joueur
- Un joueur peut influencer sur une cellule avec des:
 - Entités statiques
 - Entités mobiles

4.3.3. Entités

- Naturelles
- **Interactives**
 - Un élément qui peut être récolté (une ressource)
 - Une ressource possède une quantité définie non rechargeable de points de cette ressource que les joueurs doivent récolter.
- **Décor**
 - Un élément figé, il n'est pas destructible ou collectible
 - C'est principalement un obstacle, une zone où la construction et le déplacement sont impossible
- Crées par le joueur
- **Statique**
 - bâtiments
 - Appartient à un joueur
 - Possède un nombre de points de vie
 - Peuvent être de deux types:
 - De productions: production d'unités
 - Utilitaires: Augmente la population maximum
- **Mobile**
 - Unités
 - Appartient à un joueur
 - Possède un nombre de points de vie
 - Se déplacent
 - Peuvent être de deux types:
 - Utilitaire:
 - Récolte des ressources
 - Construit des bâtiments
 - Offensif:
 - Peut attaquer

4.3.4. Récolte

La ressource récoltable est le nerf de la guerre. Elle se trouve en quantité limité sur la carte dans des cellules contigues dont l'affichage reflète cette état.

Ces cellules possède un certain nombre de points de ressource et sont épuisables. Un click sur la cellule permet d'avoir des information sur sa quantité de ressouce.

Ces ceullues peuvent être exploitées par une **unité utilitaire** qui peut transporter un nombre finit de ressource. Chaque unité de temps t une resource est transférée de la cellule à l'unité utilitaire. Une fois plein, l'unité utilitaire retourne automatiquement au bâtiment principal (base) et les ressources qu'elle contient sont transférées au pool de ressource du joueur.

Pour initier ce mécanisme, le joueur doit selectionner une ou plusieurs unitées utilitaires et cliquer droit sur une cellule contenant des ressources.

Ce mécanisme se poursuit tant que le joueur ne selectionne pas une des unité utilitaire à la tache et ne la déplace sur une cellule sans ressource.

Une fois la ressource épuisée, la cellule devient une cellule vide (sa texture change en conséquent).

4.3.5. Inputs

- Selections:
- click gauche sur une entité permet d'afficher des informations la concernant.
- click gauche maintenu permet de faire un cadre de selection qui selectionne plusieurs entités mobiles crées par le joueur.
- click droit sur une entité sans selection préalable ne fait rien.
- click droit sur une entité avec une selection:
 - si l'entité possède des points de vie et n'est pas de l'équipe du joueur, donne l'ordre d'attaquer.
 - si l'entité est amie, elle s'y rend.
 - si l'entité est un élément de décor, ne fait rien.
- click drouasoit sur une cellule vide avec une selection:
 - la selection s'y rend.
- Déplacement de la camera:
- avec les touches fléchées du clavier (NSEW + diagonales)

4.3.6. Déroulement d'une partie

Au début d'une partie, le joueur se retrouve au commandement d'un bâtiment principale ainsi que une petite troupe (n à définir) d'unitées utilitaires. Un certain nombre de crédit (ressource) lui sont alloués. Le bâtiment principal permet de produire de nouvelles unitées utilitaires qui elles mêmes peuvent construire des bâtiments de production d'unitées offensives ou des bâtiments utilitaires pour augmenter la population. Le bâtiment principal offre une certaine limite de population qu'il est nécessaire de faire augmenter au fur et à mesure de la production d'unitées. Cette augmentation de la population se fait par la construction de nouveau bâtiments utilitaires ("maisons").

Pour assurer sa pérennité, il est nécessaire que le joueur investisse des unitées utilitaires dans la récolte de ressources qu'il investira dans de nouvelles unitées utilitaires ou des bâtiments de production d'unitées offensives pour au final amasser une armée suffisante pour détruire le joueur adverse.

4.4. Conventions

- Le code suit la convention K&R.
- Les noms du code sont écrits en anglais.
- Les distances se mesurent en pixels et l'origine est en bas à gauche

4.5. Premier diagramme de classes

Le diagramme 1 (page suivante) est ici pour la forme, il fait partie des délivrables préliminaires lors de la phase de conception et ne tient pas compte de LibDGX. Il illustre les premières réflexions sur ce qui semblait être des points importants. Aucune implémentation de ce diagramme n'a été faite.

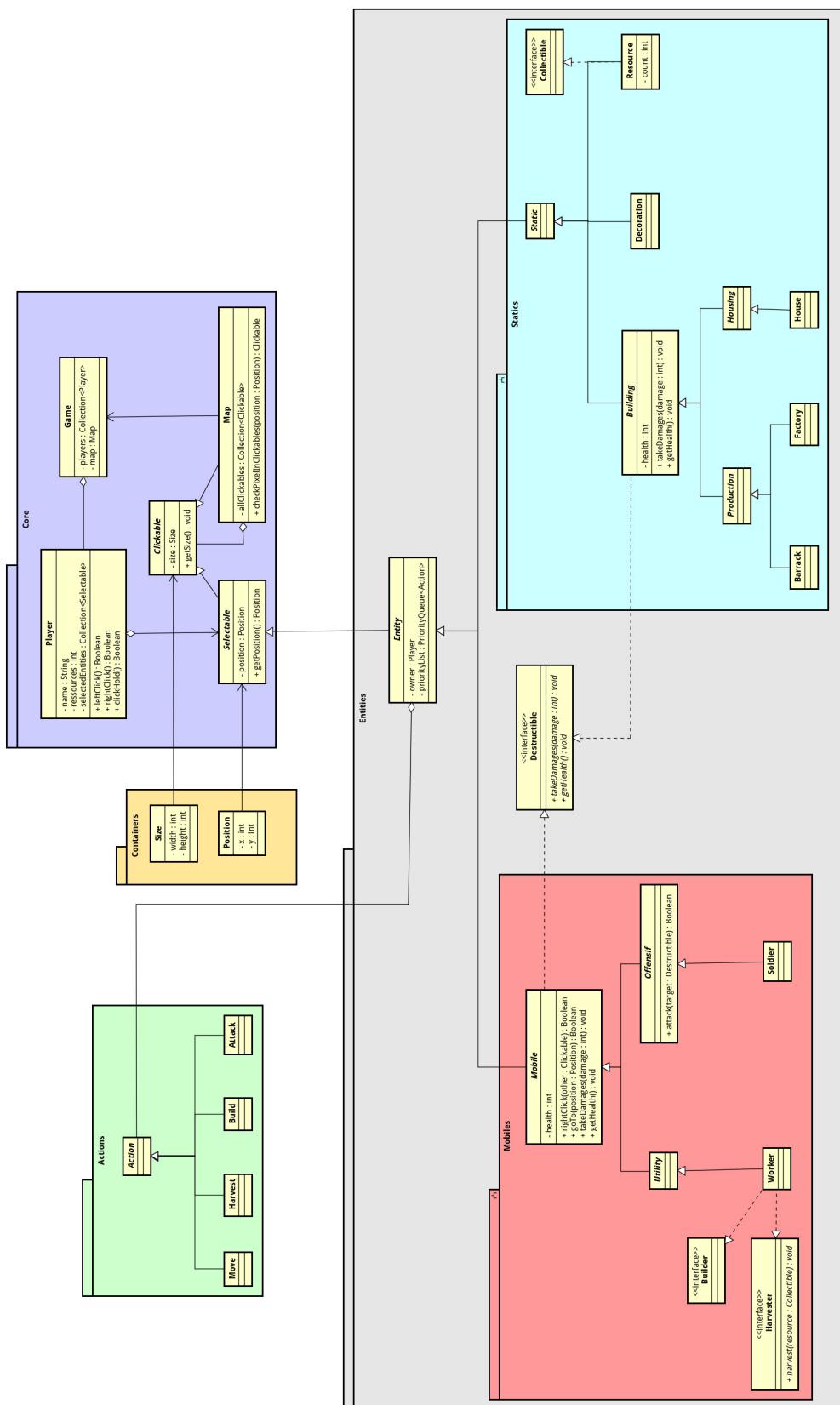


Figure 5: Premier Diagramme de classes

5. Réalisation

5.1. Architecture

L'architecture de ce projet a vu de nombreux remaniements complets avant de prendre sa forme actuelle. Ceci est dû à plusieurs choses:

- Dans un premier temps la méconnaissance de LibGDX
- Dans un second temps la compréhension plus profonde de subtilités de LibGDX
- Une simple volonté d'expérimenter
- Une nécessité imposée par les spécificités intrinsèques du genre "Stratégie en temps réel" qui cumule d'un côté un nombre conséquent de modules qui fonctionnent et communiquent ensemble et de l'autre une grande variété dans les types d'entités.

La complexité structurelle du dernier point fait qu'il existe **une sorte de moment critique** dans l'implémentation du framework. Ce moment est la transition entre l'implémentation des mécanismes de base (affichage, selection, déplacement, collision, ...) et la spécialisation de ces mécanismes.

À bas niveau, la cause principale du problème est un **trop grand couplage** entre les classes qui composent ces mécanismes. La conséquence immédiate est le manque de robustesse et la difficulté de maintenir le programme.

À plus haut niveau, c'est la spécialisation des classes qui forment la **hiérarchie des entités** qui pose problème. La conséquence est une éternelle envie de refactor du code pour tenter une optimisation utopique à chaque ajout de nouvelle entité.

Pour résoudre ces problèmes, il a été décidé de découper le projet en deux parties. Une partie **framework** dont le but est de permettre à son utilisateur de travailler à un niveau d'abstraction élevé et une partie **YARTS** qui est la concrétisation du projet.

Les prochains points de ce chapitre illustrent et détaillent les différentes approches expérimentées dans leur ordre chronologique pour pouvoir proposer un framework robuste et modulable.

5.1.1. Vue d'ensemble

Avant de parler des différentes approches il est important de comprendre la structure générale du programme et sa relation avec LibGDX.

Le diagramme qui suit est une représentation haut niveau de l'application et est le fruit des enseignements du livre de Lee Stemkoski décrit dans le chapitre introductif de ce document.

Chaque couche de ce diagramme est un élément indépendant des autres et cette représentation ne décrit pas l'architecture de chaque couche mais se contente de délimiter trois niveaux d'abstraction:

- Le plus bas niveau: **LibGDX**
- Le bas niveau: **Framework**
- Le haut niveau: **Client (YARTS)**

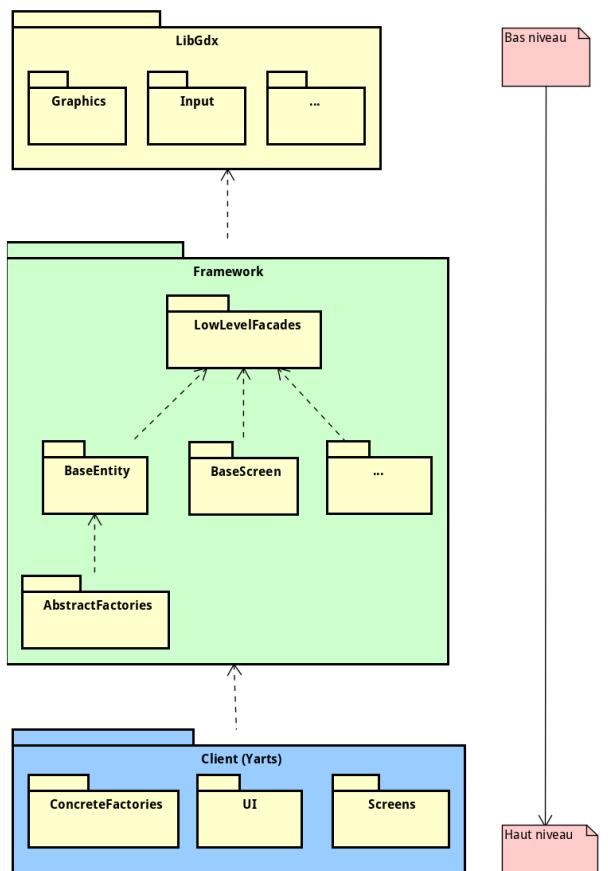


Figure 6: Overview

L'idée est de ségréguer les différentes classes dans une plage allant des "détails de bas niveau" aux "concepts de haut niveau". Si la structure est réalisée correctement, le code **Client** devrait pouvoir réaliser à l'aide des outils exposés par le **Framework** un jeu complet sans jamais avoir à appeler directement **LibGDX**.

Ce dernier point mérite une explication: Le but du **Framework** est de permettre de créer facilement (à l'aide d'abstractions importantes des outils de **LibGDX**) un jeu. Mais attention, il est important d'être conscient que **qui dit facilités, dit limitations vis à vis de ce qui pourrait être fait avec LibGDX directement**.

Aussi, il est important de comprendre que le **Framework** ne devrait pas être spécialisé dans un type de jeu en particulier. Il devrait pouvoir facilier la création de tout types de jeu 2D.

Le code de l'application se trouve réparti entre **Framework** et **Client** et les descriptions qui suivent peuvent être considérées comme les spécifications de l'architecture (ce qu'elle doit permettre d'accomplir). En effet, même si l'architecture du **Framework** à changé de nombreuses fois, la vision globale elle est resté la même:

5.1.1.1. Framework

Le niveau **Framework** est le cœur de l'application et est lui même découpé en plusieurs niveaux d'abstraction:

- **Bas niveau:** Ici se trouvent des *façades* spécialisées qui cachent la complexité liée à **LibGDX** et regroupent les fonctionnalités par catégories. Elles sont également en charge des diverses normalisations de valeurs qui leurs sont spécifiques à l'aide de classes *adapter*. En effet, **LibGDX** n'utilise pas les mêmes systèmes de coordonnées dans tous ses modules. Par exemple, la physique utilise des unités SI et l'origine se trouve en bas à gauche, la camera utilise des pixels et son origine est en haut à gauche. Les acteurs et les scènes utilisent des pixels avec l'origine en bas à gauche... Il a été décidé qu'au sein du **Framework** tout se calculerait en pixels avec l'origine en bas à gauche et c'est les classes de ce niveau qui sont en charge de la gestion de cette abstraction. Les classes de ce niveau sont en principe les seules à communiquer directement avec **LibGDX**. Elles sont le *bridge* entre **LibGDX** et le **Framework**
- **Niveau intermédiaire:** Le cœur du **Framework**. Tous les mécanismes de base essentiels à la création d'un jeu se trouve dans cette couche du **Framework**
- **Haut niveau:** Ici se trouve les plus grosses abstractions du **Framework**. Ce sont ces classes qui sont exposées au client et se sont normalement les seules qu'il doit utiliser. Ces classes sont de deux sortes:
 - Des classes abstraites dont la concrétisation sert de "cadre de travail" au client. Chacune expose des méthodes spécifiques à un aspect concret d'un jeu que le client peut soit redéfinir pour y injecter sa logique soit query pour récupérer des informations (comme des événements par exemple).
 - Des *abstract factory* qui permettent de composer ses propres entités concrètes.

5.1.1.2. Client

Comme vu précédemment, le **Client** utilise le framework pour créer des classes de convénience de plus haut niveau et injecte sa logique dans les classes "cadre de travail" du **Framework**.

Dans le cas de YARTS, le client crée des entités qui possèdent des points de vie et qui se déplacent à une certaine vitesse. Il ne s'encombre pas avec tous les mécanismes sous-jacents comme la gestion des points de vie ou la physique qui régule la vitesse de déplacement. Il peut influencer ces mécanismes dans une certaine mesure mais il est possible d'imaginer que les réglages par défaut lui conviennent et que tout ce qu'il doit faire pour afficher un soldat est d'appeler une factory, lui passer une sprite sheet qui contient les animations en spécifiant la taille d'une cellule et le tour est joué. Ce soldat ainsi créé sera est selectionnable, se déplace (en affichant l'animation correspondant à son état et son orientation) à l'endroit où le joueur clique droit. Si un obstacle entrave la route du soldat, il l'évite. Si le soldat rencontre une entité ennemie, il l'a poursuit et l'attaque automatiquement dès qu'il la rattrape. Tous ces mécanismes s'exécutent sans que le code client n'ai eu à implémenter quoi que ce soit.

Autrement dit, le **Framework** offre la possibilité au client de réfléchir en terme haut niveau de "jeu de stratégie en temps réel" et non en terme bas niveau comme des angles, des distances, ou de gestion de textures, ... Les préoccupations du client sont de l'ordre de l'équilibrage des points de vie des unités par rapport au dégats qu'elles font, leur vitesse de déplacement et l'apparence qu'elles ont.

Le mécanisme qui permet au client d'injecter de la logique dans les classes qu'il dérive lui permet de définir des comportements ou des actions plus complexes comme de donner la possibilité à son soldat d'utiliser une arme spéciale qui fait plus de dégats mais qui nécessite un long temps de chargement en cliquant sur une icône qui apparaîtrait automatiquement dans l'interface quand cette unité est sélectionnée. Ou même de lui permettre de construire des bâtiments qui tout comme l'arme spéciale apparaîtraient sous forme d'icônes dans l'interface.

Dans le même ordre d'idées, le **Client** peut créer un mécanisme qui lui permet de créer une entité aux allures de bâtiment à laquelle il assignerait une vitesse de déplacement nulle et qui en réalité cacherait une *factory* de soldats. Dans le jeu, une fois construit, lors de la sélection de ce bâtiment, une icône cliquable dans l'interface permet d'instancier des soldats qui apparaissent à côté du bâtiment. Tous ces soldats ont le même comportement que le soldat original.

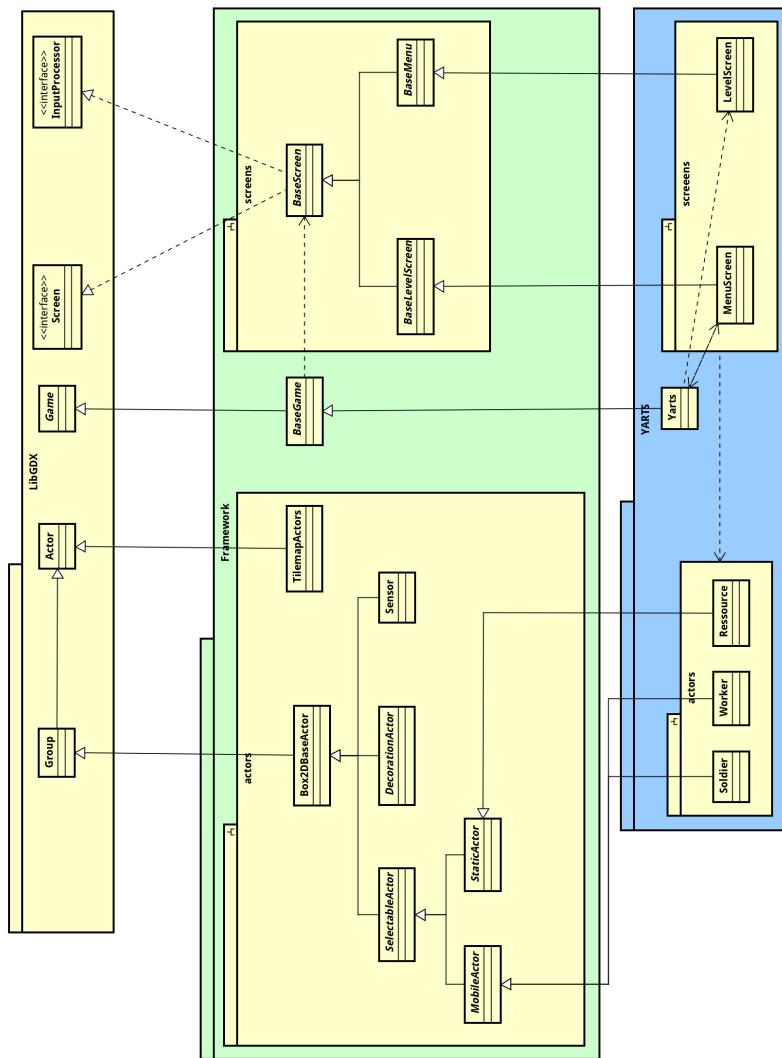
5.1.1.3. Avantages

- **Robustesse:**
 - Des changements dans le code à un niveau particulier n'impacte que modérément les autres niveaux voir pas du tout.
 - La ségrégation des classes facilite le debug en simplifiant l'isolation des comportements et donc l'identification de la source du bug.
 - Facilite le refactoring / modification.
- **Garde la complexité sous contrôle:**
 - Favorise des implémentations qui suivent le principe de responsabilité unique ce qui résulte en un code plus clair.

De nombreuses façons permettent d'arriver à ces comportement et la première tentative fut naturellement axée sur un usage important de l'OOP.

5.1.2. Approche OOP

Le diagramme 2 (page suivante) représente une vue simplifiée de la première approche. Dans le package "actors" contenu dans "Framework", on voit une ébauche de l'arbre d'héritage des entités. Cet arbre n'est pas complet et dans la pratique, plusieurs variantes modélisées sur papier ont été tentées. Dans le package "YARTS" on peut voir un second package "actors" qui contient des concrétisations issues de l'arbre d'héritage.

**Figure 7: Approche OOP**

Cette approche est naturelle, et pour peu que l'architecture des relations et de la communication entre les classes qui se chargent des mécanismes de base implémente les bon patterns, le résultat est fonctionnel et relativement robuste. Le problème vient principalement du coté de l'arbre d'héritage des entités.

En effet, dans cette implémentation, chaque entité est un objet. Le système d'instanciation est donc basé sur des classes et permet à une entité d'en étendre une autre tout en jouissant de comportements polymorphiques.

Poussée au maximum, cette façon de faire conduit à une grande hiérarchie de classes rigide où la difficulté de définir la place d'une nouvelle entité est proportionnelle à la taille de la hiérarchie. Le diagramme suivant illustre ce problème:

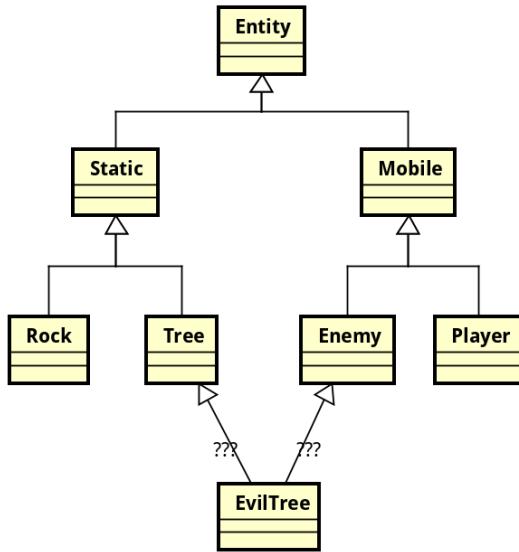


Figure 8: hiérarchie de classes

Une solution à ce problème serait de ne pas baser la gestion des entités sur l'héritage mais sur de la composition d'objets.

5.1.3. Approche OOP + composition (ECS naïf)

Dans cette approche, la construction d'entités ne se fait plus via héritage mais via composition tout en tirant profit du polymorphisme en définissant une interface `Entity`. Une entité devient une aggrégation (techniquement une composition) de composants et chaque composant encapsule la logique qui le concerne ainsi qu'une référence vers l'entité qui le contient:

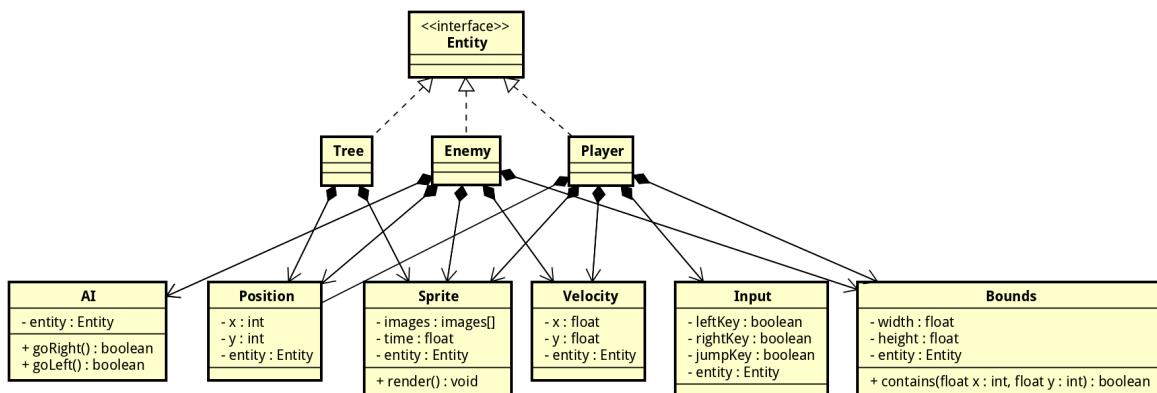


Figure 9: ECS Naïf

Malgré une tête de spaghetti et une complexité apparente dans la modélisation, un système de gestion des entités basé sur ce système possède les avantages suivants:

1. Il est ais  d'ajouter une nouvelle entit .
2. Possibilit  de dynamiquement ajouter ou retirer des composants (et donc de modifier le comportement).
3. Pour un grand nombre d'entit s (ce qui est le cas pour un jeu de strat gie en temps r el) plus de performances.

Par contre, cette fa on de faire est d'une part plus verbeuse dans le code et d'autre part d place le probl me de la complexit  dans les composants. En effet, certains comportements peuvent  tre sensiblement diff rents en fonction de l'entit  qui l'utilise. Aussi, le code qui g re la logique a tendance   se charger de batteries de conditionnels   mesure que le nombre d'entit s grandit.

Dans YARTS, cette technique a  t  employ e sans n cessiter une r ecriture compl te du code. Il n'existe donc pas de diagramme d crivant cette architecture.

5.1.4. Approche ECS

ECS veut dire Entity Component System et cette approche se base sur la pr c dente et pousse son concept   l'extr me. En bref, les `Entity` sont les objets du jeu et sont d finies implicitement par une collection de `Components`. Ces `Components` ne contiennent que des donn es et sont op r s en groupes fonctionnels par des `Systems`.

5.1.4.1. Component

Un component est un simple conteneur. Une classe qui impl mente un component a des attributs mais pas de m thode. Chaque component d crit un certain aspect d'une entit  ainsi que ses param tres. En soit, un component, n'est pas grand chose et c'est leur cumul qui est int ressant. Voici un exemple de composants:

- `Position(x, y)`
- `Velocity(x, y)`
- `Physics(body)`
- `Sprite(images, animations)`
- `Health(value)`
- `Damages(value)`

5.1.4.2. Entity

Dans le cadre de l'ECS, une entité est un concepte, mais peut être vu comme un objet du jeu. Par exemple, un rocher, une maison ou un soldat. Fondamentalement elle n'est définie que par les composants qui la constitue et un ID. Il est possible d'ajouter ou de retirer des composants pendant l'exécution, ce qui se traduit en une façon fondamentalement différente d'aborder les choses. En effet, dans une vision ECS des choses, on peut imaginer qu'une de nos entités est un mage qui peut geler les soldats adversaires. Ces soldats sont eux mêmes des entités et si ils sont touchés par le sort de glace du mage, il suffit de leur retirer leur component Velocity pour les clouer sur place. À partir des composants précédents on peut imaginer les Entités suivantes:

- Rock(Position, Sprite)
- Ball(Position, Velocity, Physics, Sprite)
- Wizard(Position, Velocity, Sprite, Health, Damages)

5.1.4.3. System

Les systèmes sont le cœur de la logique de l'ECS. Un système opère sur une combinaison de composants spécifique. Par exemple, Le système MovementSystem peut opérer sur les entités composées des components Position et Velocity et contient toute la logique qui permet de déplacer des entités. Chaque système, et dans l'ordre d'instanciation de tous les systèmes sera mis à jour idéalement 60 fois par seconde. Voici quelques définitions de systèmes:

- MovementSystem(Position, Velocity) : Applique une vitesse à l'entité qui possède Position
- GravitySystem(Velocity) : Applique une accélération à l'entité qui possède Velocity
- RenderSystem(Position, Sprite) : Affiche les entités qui possède Position et Sprite

5.1.4.4. Implémentation

Le diagramme 3 est particulier dans le sens où il montre une représentation logique des Entités. Cette hiérarchie n'existe pas en tant que tel dans le code. Seul les composants existent. Par contre dans le code, la classe *EntityFactory* se sert de cette représentation pour limiter le répétition de code.

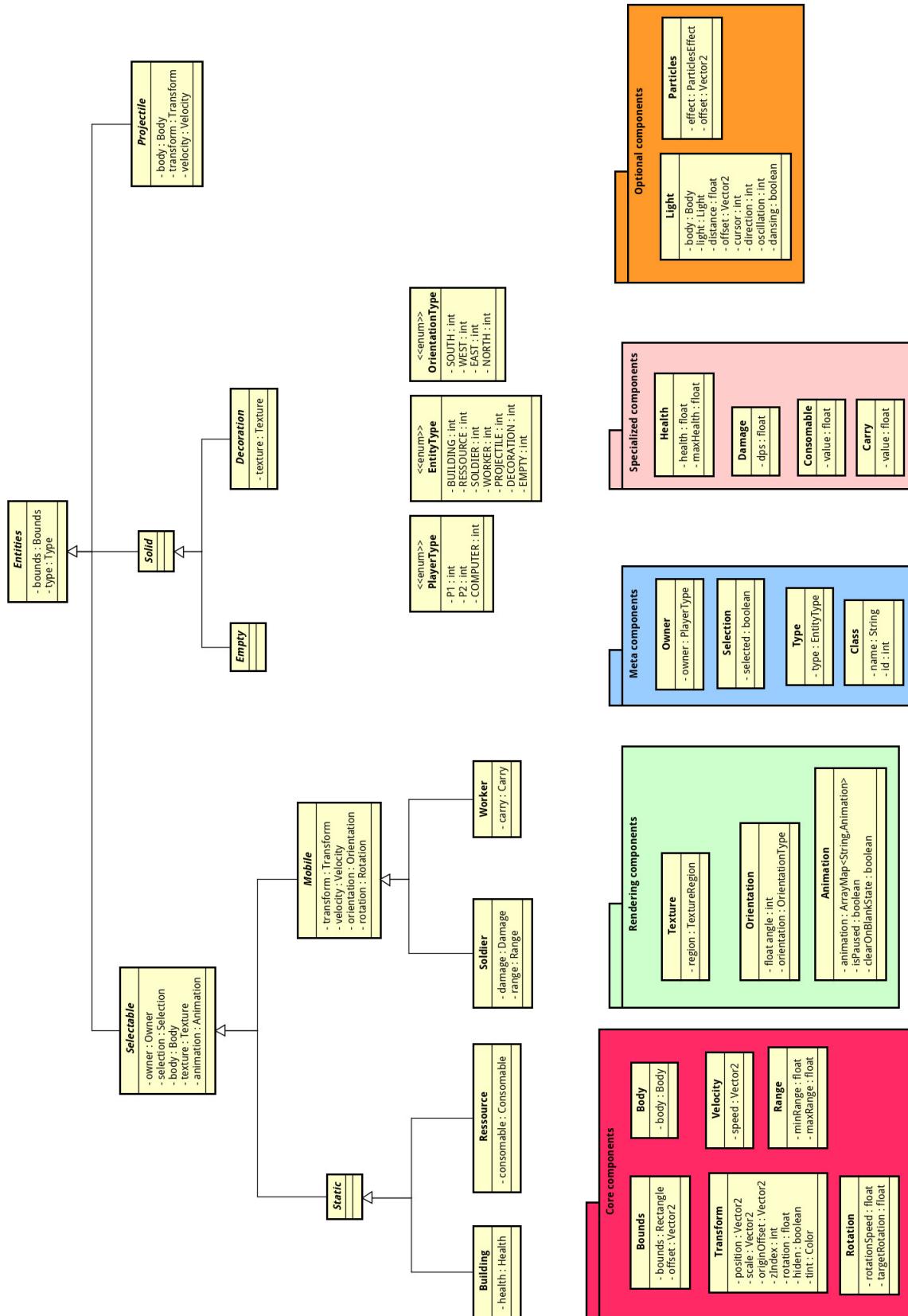


Figure 10: Simplification de l'architecture finale

Cette représentation a servi à l'implémentation initiale du système ECS de YARTS. Depuis, certains Components ont été modifiés et d'autres ont été ajoutés.

5.1.5. Conclusion sur l'architecture

Pour le choix de l'implémentation à retenir, les trois critères ont été évalués:

- Performance
- Complexité du code
- Maintenabilité

5.1.5.1. Performance

Un test interne à YARTS permet d'instancier un grand nombre d'entités sur la carte. Une entité ayant approximativement les mêmes fonctionnalités a été créé dans chaque implémentation. Ces fonctionnalités sont: Texture et Animation, Physique et Detection de collision. Pour ce test, tous les effets de lumière et d'ombre sont retirés. Le tableau suivant représente les résultats. Les ordonnées représentent le nombre d'entités. Les Abscisses, le type d'implémentation. Au centre se trouve les valeurs obtenues exprimées en images par seconde (fps).

	OOP	OOP + Comp	ECS
50	60 fps	60 fps	60 fps
200	32 fps	15 fps	60 fps
500	9 fps	1 fps	60 fps
1000	1 fps	1 fps	53 fps
10000	crash	crash	2 fps

Il est remarquable que les performances de la variante OOP + Composition soit si faible en comparaison de la version OOP. Il est probable qu'un problème dans l'implémentation fausse les résultats. Malgré ça, il est improbable que même avec un code optimal l'issue du test soit différent. L'ECS est sensiblement plus performant que les deux autres implémentations et cette information colle avec ce qu'on trouve sur le net.

5.1.5.2. Complexité du code

Ce test est relativement subjectif et se base sur l'impression de complexité dans le codes des 3 implémentations à fonctionnalités égales.

Les deux premières variantes sont approximativement de la même complexité. Dans la version OOP, la complexité se trouve dans les dérivées de la classe mère Entity. Dans la version OOP + Composition, elle se trouve dans les Components. De plus, il est important de noter que les mêmes problèmes apparaissent approximativement au même stade de développement.

L'ECS a une philosophie totalement différente dans son fonctionnement et il n'est pas évident de raisonner en mode ECS lors de l'implémentation des systèmes. Pour donner un ordre d'idée c'est un peu la même difficulté de réflexion que les premiers pas en programmation concrète. De plus l'utilisation de l'ECS a empêché l'utilisation d'autres systèmes de LibGDX et a donc nécessité l'implémentation d'un équivalent. Dans ce contexte, il est à noter qu'après quelques refactor, la complexité générale du code est restée plus faible que dans les deux autres implémentations. Avec la version ECS, au stade de développement actuel, on remarque que les problèmes des variantes OOP n'apparaissent pas. Le code est relativement robuste et il est très simple d'ajouter de nouveaux mécanismes.

5.1.5.3. Maintenabilité

Comme expliqué précédemment, les versions OOP, passé un certain stade sont difficiles à maintenir. Il est probable qu'une meilleur architecture les concernant puisse résoudre ou améliorer la situation. La variante ECS quant à elle, est très flexible. Il est vraiment aisés d'ajouter de nouveaux composants/systèmes.

De ces tests il ressort que l'ECS est l'implémentation qui est la plus performante, lisible et maintenable. Ces résultats sont en concordance avec ce qui se trouve sur le net dans le cadre d'un jeu de stratégie en temps réel. C'est donc cette implémentation qui est retenue pour coder YARTS.

La version actuelle du code implémente l'approche ECS à l'aide du module Ashley de LibGDX. Ce module augmente les performances vis à vis de l'implémentation manuelle en offrant des conteneurs spécialisés et optimisés pour le système.

5.2. Mécanismes

De nombreux mécanismes ont dû être implémentés pour avoir un jeu fonctionnel. Certains, comme le pathfinding ou la gestion des collisions sont fort documentés sur le net et des classes existent dans LibGDX qui en facilitent l'implémentation. D'autres, au contraire sont peu documentés. En règle générale, les mécanismes spécifiques aux jeux de stratégie en temps réel sont peu représentés. Cela dit, il est toujours possible de trouver des éléments de réponse dans l'un ou l'autre fil de discussion.

Chaque mécanisme de ce programme, petit ou grand a nécessité un temps considérable de réflexion, de recherche et de raffinement pour d'une part se l'approprier et d'autre part faire en sorte qu'il s'intègre au mieux dans le framework tout en restant le plus générique possible. De la même façon, les décrire dans ces pages nécessite tout autant de minutie et se trouve être un exercice particulièrement complexe mais surtout trop long. Il ne sera donc présenté ici qu'une sélection de mécanismes et l'impasse sera faite sur les tests effectués sur ces derniers.

5.2.1. Debugging et configuration

Pour une application de ce genre, il est crucial de pouvoir afficher des informations utiles au développement pendant l'exécution. Pour se faire, des mécanismes parallèles de debugging ont été mis en place. Il sont de deux types, logging (console) et render (dans le jeu). Le *singleton Conf* contient toutes les variables qui permettent d'activer ou de désactiver ces mécanismes de debug. Cette classe centralise également toutes les valeurs qui touchent à la configuration du jeu (résolution, full screen, constantes de conversion d'unités, ...). Tous les autres parties du framework utilise ce fichier ce qui permet de ne pas avoir de "valeurs magiques" et d'augmenter la lisibilité du code.

Au lancement du programme, s'affiche dans la console un petit résumé de quelques grandeurs utiles ainsi qu'un résumé des touches qui active (toggle) les mécanismes de debug:

```

1 Config info:
2 -----
3 Resolution 1920 x 1080
4 World width: 60.0 x 33.75
5 Gravity: (0.0, -9.8)
6 PPM: 32.0
7
8 Map info:
9 -----
10 A tile: 16.0 x 16.0
11 Tiles: 200 x 200
12 Map (pixel): 3200.0 x 3200.0
13 Map (meters): 100.0 x 100.0
14
15 Debug info:
16 -----
17 F1: toggle state notifications
18 F2: toggle inputs events
19 F4: toggle gui
20 F7: toggle health
21 F8: pathfinding strategy
22 F9: toggle pathfinding marks
23 F10: Cycle range
24 F11: toggle lights
25 F12: toggle physics debug

```

Bash

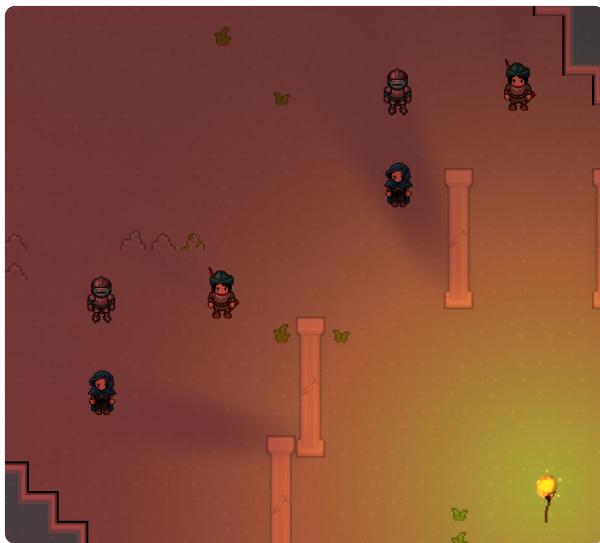


Figure 11: Lights on



Figure 12: Lights off

"toggle physics debug" affiche les zones de collision des différents entités ainsi que leur vecteur de déplacement de ces dernières:



Figure 13: Normal vue



Figure 14: Physics vue

"toggle pathfinding" permet d'afficher un visuel du trajet qu'emprunte une entité.

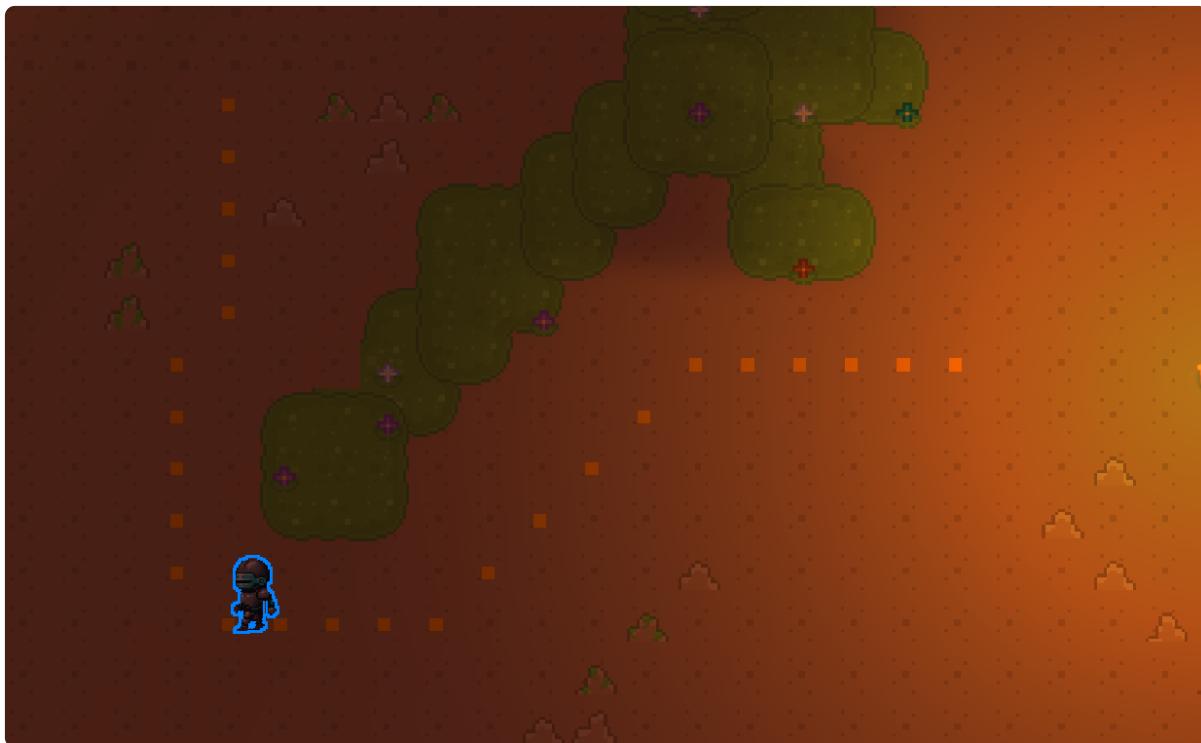


Figure 15: Path finding

Les barres de vie ne s'affichent que si une entité n'est pas à 100% de sa vie. "toggle healthbar" permet d'activer ou de désactiver ces barres.

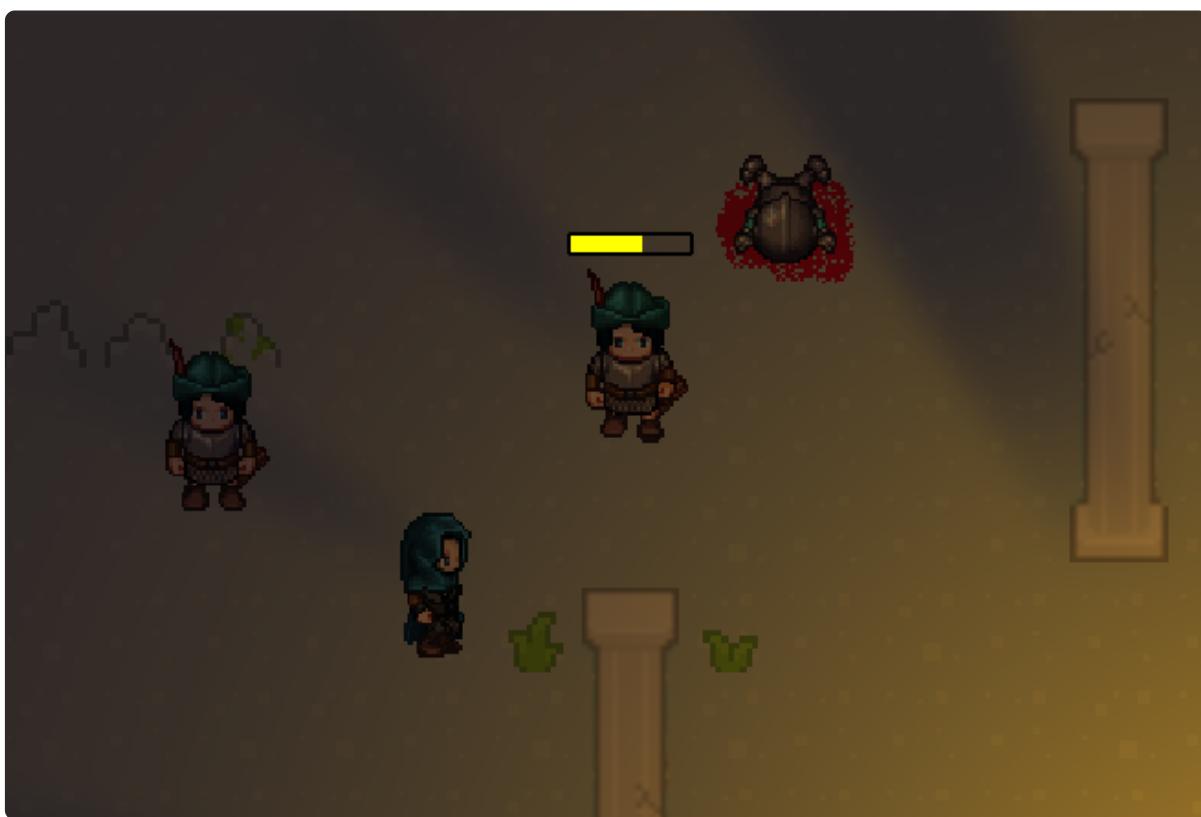


Figure 16: Health

D'autres affichages de debug plus spécifiques à l'un ou l'autre systèmes seront présentés dans les pages qui suivent.

5.2.2. Champ de vue, portée et état

Dans YARTS, les entités ont conscience du monde qui les entoure et réagissent à ce qu'elles voient. Chaque entité possède un champ de vue et une portée qu'il est possible de voir dans le jeu en pressant plusieurs fois sur F10 :

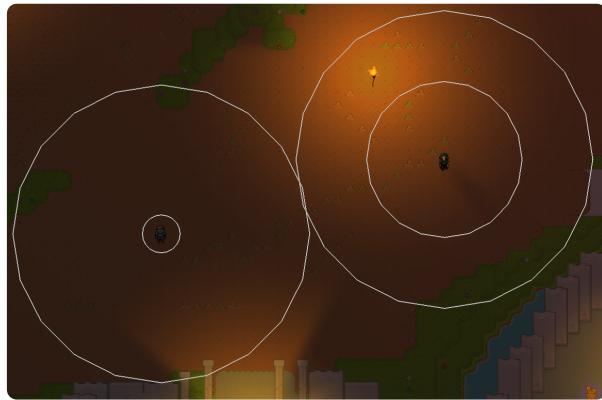


Figure 17: Champ de vue et portée

Au centre de chaque cercle se trouve une entité. À gauche un Soldat et à droite un Mage. Le cercle extérieur représente le champ de vue de l'entité et le cercle intérieur est spécifique au type d'entité. Le Soldat est de type Cac (Corp à corp) et son petit cercle correspond à sa portée d'attaque. Le Mage est une entité de type Distance et sa portée d'attaque est équivalente à son champ de vue alors que le cercle intérieur représente la distance minimum à laquelle il peut attaquer.

Ces deux entités n'appartiennent pas au même joueur, et sont donc automatiquement agressives l'une face à l'autre. Cela veut dire que si par exemple le Mage rentre dans la portée du Soldat, ce dernier va l'engager sans que le joueur n'ai quoi que ce soit à faire.

Pour le moment les deux entités ne se voient pas et elle se trouve dans l'état `Passive`. En effet dans Yarts, le comportement des entités est régit par une machine d'état. Les changements d'états sont influencés par deux choses:

1. Le monde qui entour l'entité
2. Les inputs de l'utilisateur

Le diagramme 4 (page suivante) présente la première version de la machine d'état qui régit le comportement des entités. Dans la version actuelle du jeu, deux états supplémentaires complètent ce modèle (`Disengaging` et `Dead`).

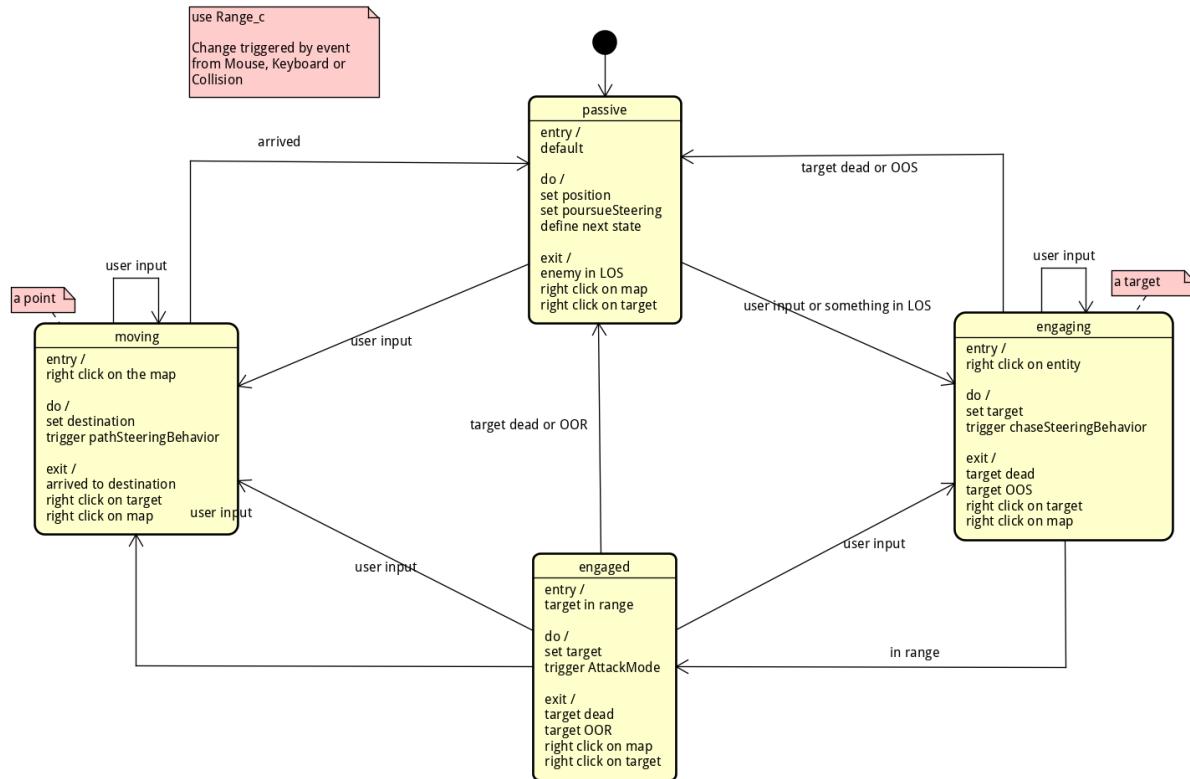


Figure 18: Machine d'état des comportements

Si maintenant, le joueur sélectionne le mage et le déplace sur le bord de la portée du Soldat, le mage passera à l'état "Moving" et ce changement sera log dans la console:

[STATE] P1 WIZARD3 MOVING state towards (41.999996,19.799997)

Dès qu'il rentre dans le champ de vue du Soldat, ce dernier passe à l'état "Engaging" qui est l'état qui correspond au fait qu'il a vu un ennemi mais qu'il est trop loin pour l'attaquer. Le Soldat se met donc automatiquement à la poursuite du Mage et la console va log la ligne suivante:

[STATE] P2 WARRIOR5 ENGAGING state Against WIZARD3



Figure 19: Entité engage

Les cercles reflètent l'état de l'entité ("Engaging = jaune")

Une fois que le mage finit son déplacement il passe revient à l'état "Passive" mais comme le Mage voit maintenant aussi le Soldat il change directement d'état et passe en "Engaged". En effet comme le mage est de type Distance, il n'a pas besoin de se déplacer pour attaquer le Soldat et passe directement à l'état "Engaged":

```
[STATE] P1 WIZARD3 ~PASSIVE~ state
[STATE] P1 WIZARD3 -ENGAGED- state Against WARRIOR5
```

Dans cet état, le mage attaque le Soldat avec des boules de feu dont la fréquence est gérée par une autre machine d'état (qui ne sera pas développée dans ce rapport). La capture suivante illustre cet instant: Le Soldat se déplace dans la direction du Mage ("Engaging") et le Mage lui attaque le Soldat ("Engaged"):

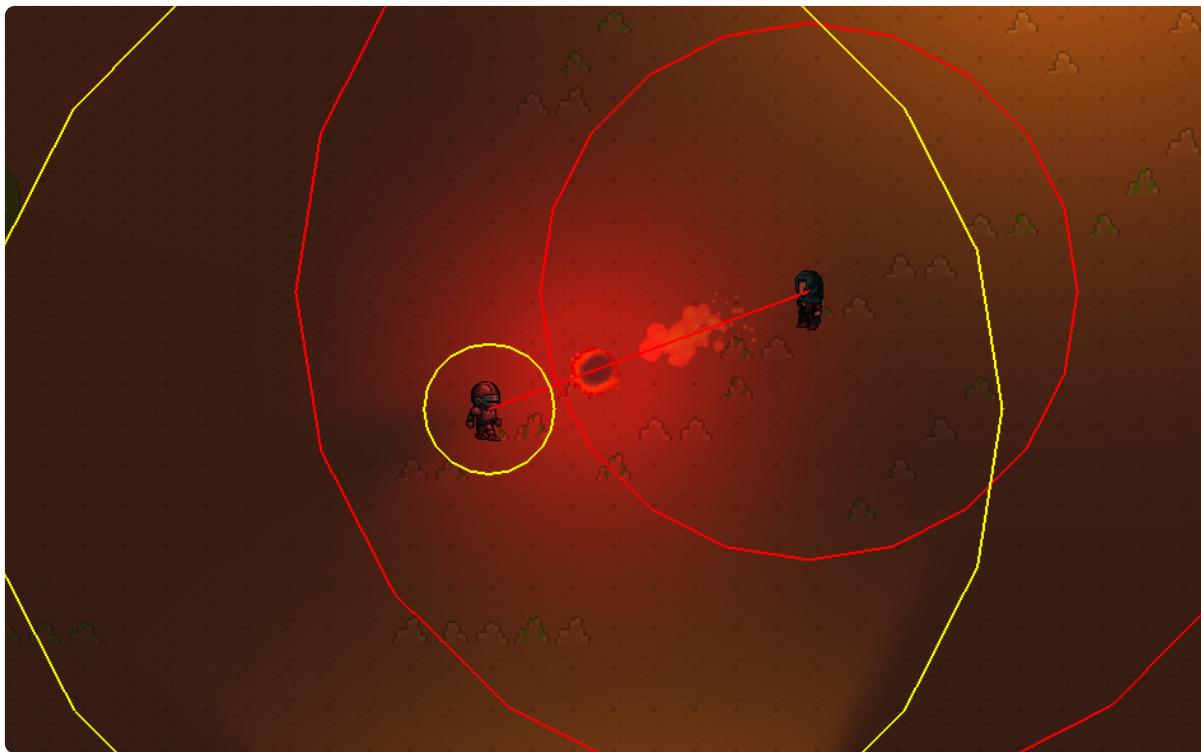


Figure 20: Entité à portée

Soldat: "Engaging" (jaune) Mage: "Engaged" (rouge)

Le Mage ne peut pas se déplacer pendant qu'il incante une boule de feu, et fatallement le Soldat va finir par rentrer dans le petit cercle du Mage qui correspond à sa portée minimum. À cet instant, le Mage change son état en "Disengaging" ce qui correspond à un état de fuite. Il se déplace donc automatiquement dans la direction opposée de celle du Soldat:

```
[STATE] P1 WIZARD3 DISENGAGE state Against WARRIOR5
```



Figure 21: Perte de la portée

Noter l'apparition de la barre de vie suite au dégâts faits par la boule de feu

Comme le mage se déplace légèrement plus vite que le Soldat (Il ne porte pas d'armure) au moment où la portée est à nouveau suffisante, il repassera à l'état "Passive" et l'instant d'après à l'état "Engaged" et lance une nouvelle boulle de feu sur le Soldat. Ainsi de suite jusqu'à ce que le Soldat le coince dans et coin et ne déverse toute sa rage sur lui.

[STATE] P2 WARRIOR5 -ENGAGED- state Against WIZARD3



Figure 22: Entitee à portée

Et finalement après quelques baffes de plus:

[STATE] P1 WIZARD3 DEAD state

Les mécanismes qui viennent d'être décrits forment l'IA des entités du jeu. Le fonctionnement de ces mécanismes fait intervenir de nombreux systèmes et leur conception est au coeur de la problématique décrite dans l'introduction du chapitre qui décrit l'architecture.

Le prochain point couvre l'échange de messages et les notifications entre les différents systèmes.

5.2.3. Notifications

Pour réduire le couplage au sein du framework il a rapidement été nécessaire de trouver un moyen efficace pour gérer la communication entre les différents systèmes. Le diagramme 5 présente une version simplifiée de ce mécanisme. Ce diagramme montre l'utilisation du pattern *observer* qui s'occupe de dispatcher les evenements générés d'un coté par LibGDX (inputs) et de l'autre par les systèmes.... aux systèmes.

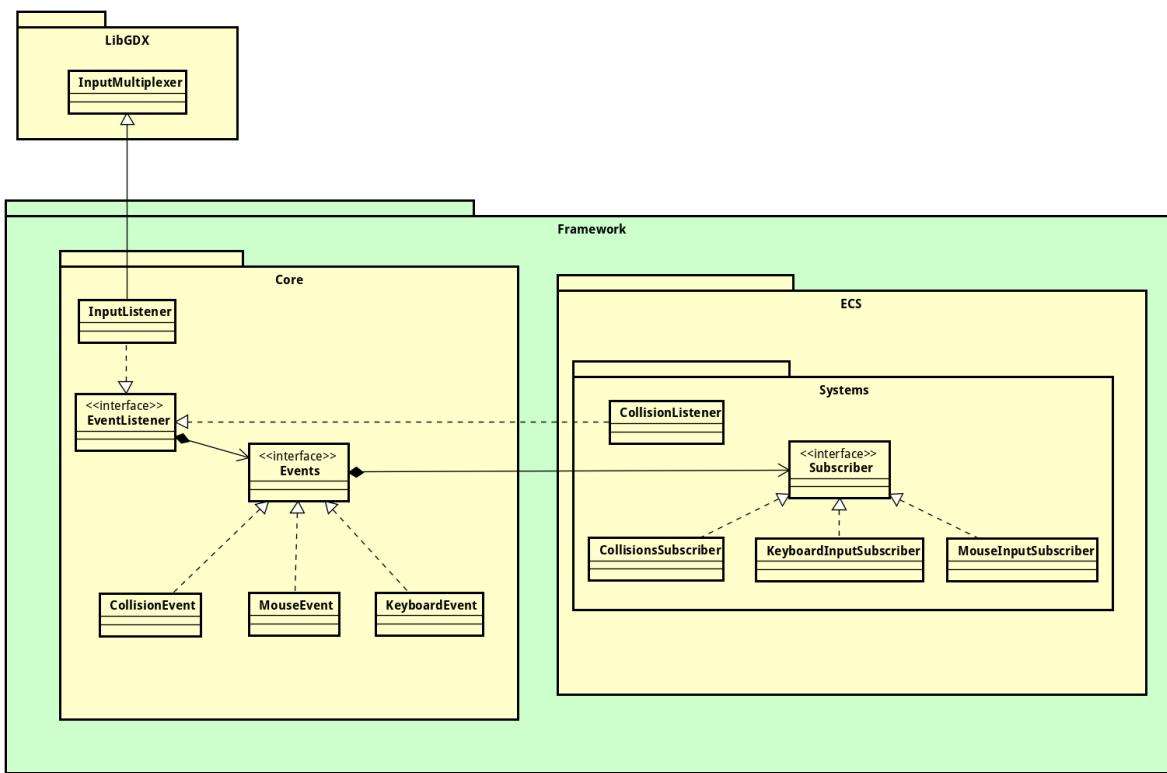


Figure 23: Implémentation du patern observer

Afin de renforcer le principe de responsabilité unique, les événements communiquent via des classes qui implémentent le pattern *commande* ce qui déplace le code lié à la communication des systèmes à une série de classes (qui pour plus de clarté ne sont pas visibles sur le diagramme) qui portent le suffixe "Handler".

5.2.4. Pathfinding

L'Algorithme utilisé est A*, une extension de l'algorithme de Dijkstra. Il permet de calculer le chemin le plus court entre deux points.

En interne, le mécanisme de pathfinding se représente la map sous la forme d'un graph qui représente une grille de points. Il est possible de voir le chemin généré par l'algorithme avec la touche `f9`.

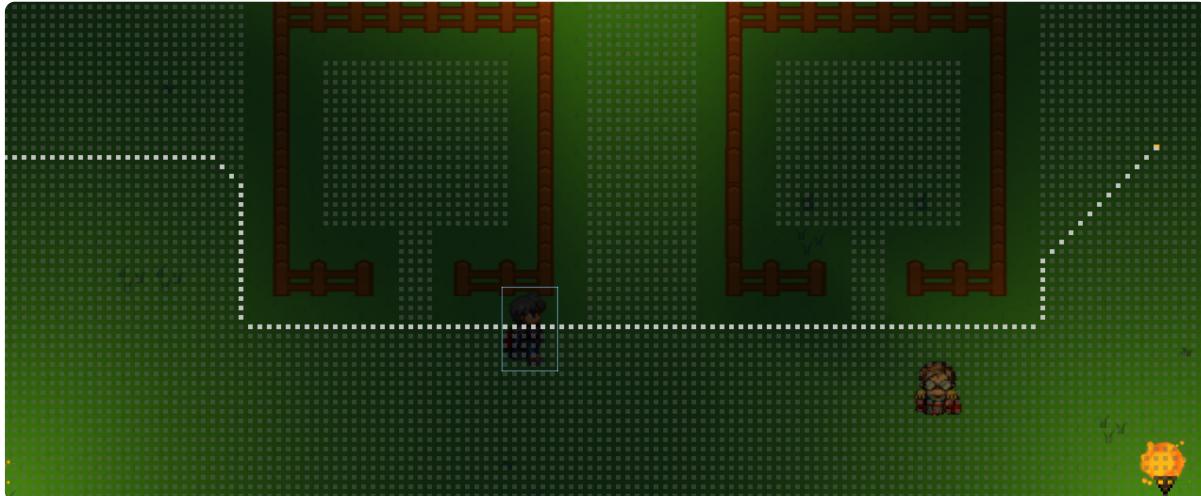


Figure 24: Première itération du pathfinding

Pathfinding dans une ancienne version du jeu, mais le principe est le même.

L'implémentation de l'algorithme ayant des problèmes, son utilisation n'est pas complètement fiable. L'utilisation d'un pattern *strategy* permet de changer entre l'algorithme de déplacement de base et A* en pressant la touche `f7` en jeu.

5.2.5. Brouillard de guerre

Le brouillard de guerre permet de cacher une partie de la carte, il sert à dissimuler les actions d'un autre joueur ou de l'ordinateur.



Brouillard de guerre

Figure 25: Brouillard de guerre

Illustrer le brouillard de guerre Capture d'écran du jeu Dune 2 qui

La première tentative d'implémentation du brouillard de guerre utilisait les facteurs d'OpenGL pour faire du mélange de couleurs (blending) cette façon de faire fonctionnait très bien dans un environnement isolé mais n'a pas pu être adaptée au framework. Le problème vient de la façon dont est gérée la carte dans le framework.

Après plusieurs autres tentatives, le brouillard de guerre était presque utilisable. Par manque de temps, le choix a été fait de mettre cette fonctionnalité en attente.



Figure 26: Implémentation fonctionnelle dans le framework mais hautement instable

5.3. Design

Le design permet d'informer les utilisateurs du champ d'application d'un logiciel. Par exemple, un programme de gestion est moins attrayant visuellement. Il doit proposer à l'utilisateur des fonctionnalités qui permettent de traiter les données rapidement. Grâce à la disposition de ses éléments, on peut savoir rapidement à quel catégorie appartient le logiciel et qu'elle est son champ d'application.

Un jeu vidéo doit être attrayant pour trouver son public. Il est nécessaire de captiver les joueurs dans le jeu. C'est pour cela que le logiciel doit avoir beaucoup d'animations et d'événements indépendants qui permettent aux utilisateurs d'être toujours actif.

Le design permet aussi de donner une personnalité au jeu. Le choix du thème abordé est importante pour avoir une certaine harmonie.

5.3.1. Map

La map représente la surface jouable et il existe différentes de la représenter. Une des premières idées pour pouvoir faire des profondeurs de champ est d'utiliser la 3D. La modélisation 3D est lourd en développement et demande une certaine technicité. La vue 2D isométrique se rapproche d'un simple plan vu de dessus mais avec une gestion d'image et de la grille différente de la 2D "classique". On appelle aussi ce type de vue 2.5D ou isométrique. Pour mieux visualiser la différence voici deux exemples :

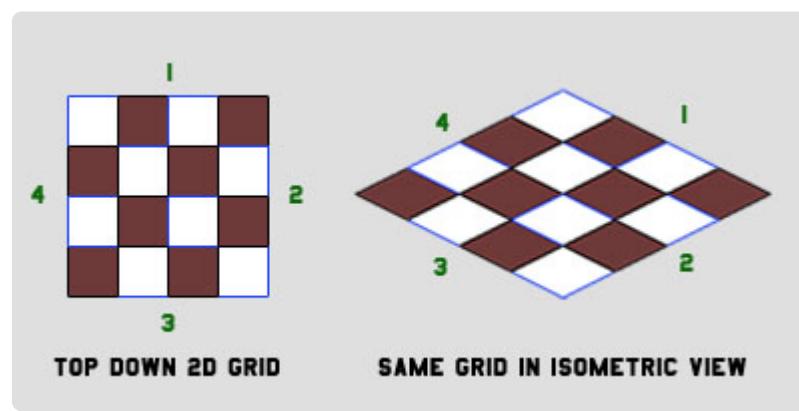


Figure 27: Perspective

Dans YARTS, Le choix s'est porté sur la 2D (2.5D) pour réduire la complexité générale. En effet, le changement de coordonnées de la vue isométrique ajoute une couche de complexité qui n'est pas nécessaire pour ce projet.

La carte de YARTS est une vue classique 2D (2.5D). L'utilisation d'une coordonnée de hauteur zsert de critère de tri des éléments pour gérer leur ordre d'affichage et permet de donner un effet de perspective qui est implémenté avec une astuce toute simple: L'inverse de la coordonnée y d'un objet sur la map 2D est équivalent à la coordonnée z.

Les images suivantes illustrent la gestion de la perspective dans le jeu. Si un personnage est plus haut sur l'axe des ordonnées, il se trouve derrière un objet situé plus bas sur l'axe.



Figure 28: Z-index



Figure 29: Z-index 2

Pour la création de la map, [Tiled](#) est utilisé. C'est un logiciel qui permet de générer des fichiers utilisable directement dans le framework. Tiled permet de définir des emplacements référencés par une string qu'il est possible de les récupérer dans le framework pour y instancier des objets particuliers:



Figure 30: Dans tiled



Figure 31: Résultat en jeu (lights off)

5.3.2. Sprites

Un sprite est une image d'une taille standard en pixel (32x32, 64x64,...). Une collection de sprites se nomme "tileset" dans le cas de sprites qui contiennent des éléments de décore. YARTS utilise un tileset de offert par kenney.nl:

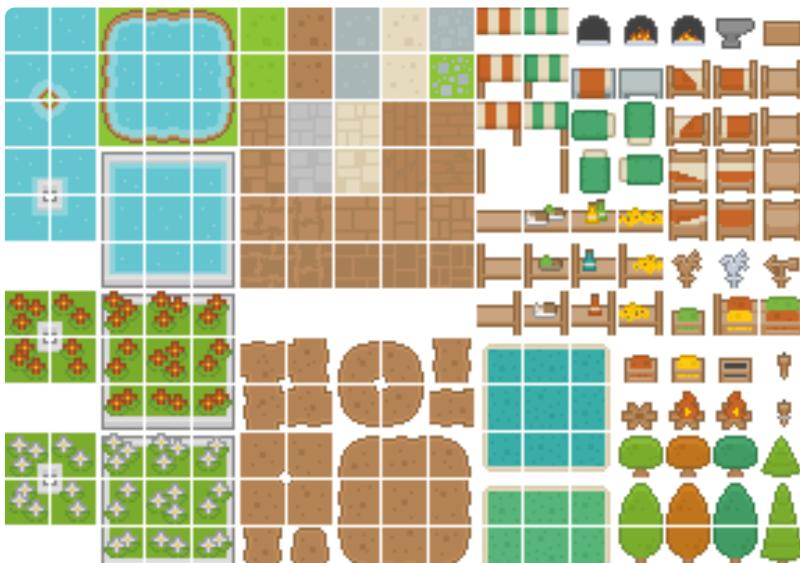


Figure 32: Tileset

5.3.3. Animations



Figure 33: Animations

Pour les sprites d'animations on parle de spritesheet, chaque image (frame) d'une animation est une région (une "case") de cette spritesheet. L'image sur la droite est la spritesheet utilisée pour l'entité Archer.

L'animation des personnages se fait grâce aux différents systèmes de gestion des textures. `OrientationSystem` permet de déterminer quel direction doit utiliser `AnimationSystem` pour que le personnage s'affiche dans la bonne orientation au moment d'un déplacement. `AnimationSystem` détermine ensuite quel image set dans le component `TextureComponent`. `RenderingSystem` lui ne se charge que de dessiner l'image qu'il trouve dans `TextureComponent`.

Pour le projet, la création des personnages s'est fait grâce à un générateur de spritesheet [Universal LPC Generator](#). Les images sont ensuite traité sur [Gimp](#)

5.3.4. Gestion des ressources

L'instanciation d'image peut rapidement devenir lourd en mémoire. La classe `TexturesManager` implémente *flyweight* et *singleton*. Elle permet de n'avoir qu'une unique instance de chaque image ou tileset.

5.3.5. Interface utilisateur

L'interface utilisateur créée pour le projet doit pouvoir être dynamique et se mettre à jour régulièrement par rapport aux événements du jeu. Il permet d'avoir une vue d'ensemble de la partie et de donner des actions possibles à l'utilisateur.

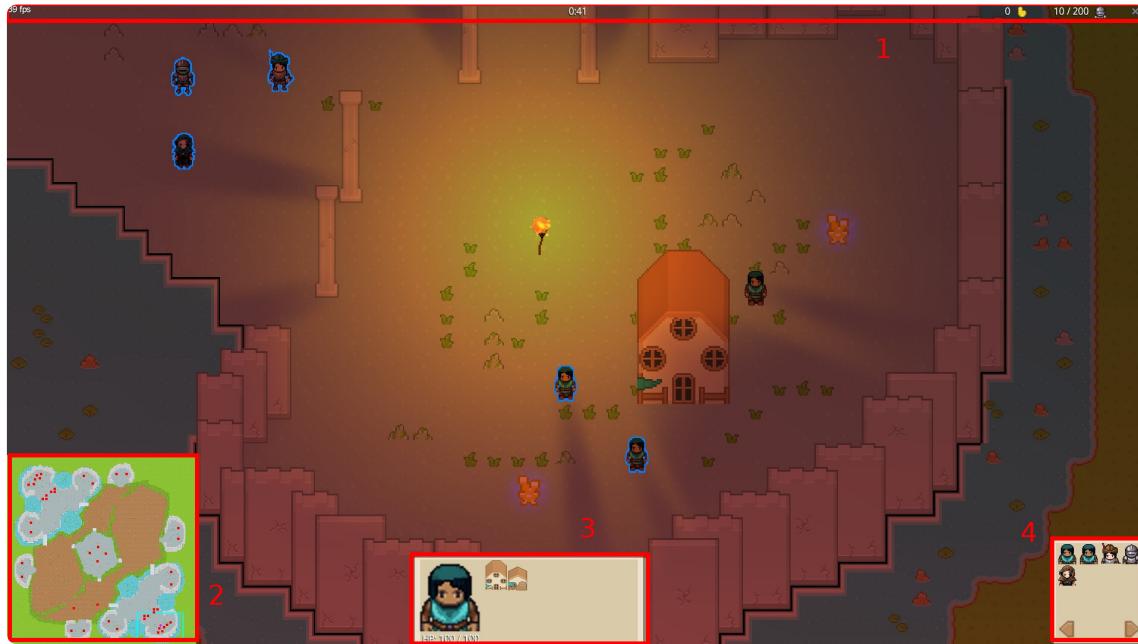


Figure 34: UI

1. Information sur la partie : image par seconde, temps de jeu, ressources, nombre d'unité disponible
2. Mini-map
3. unité principale sélectionnée, boutons d'action
4. liste d'unités sélectionnées

5.3.5.1. Mini-map

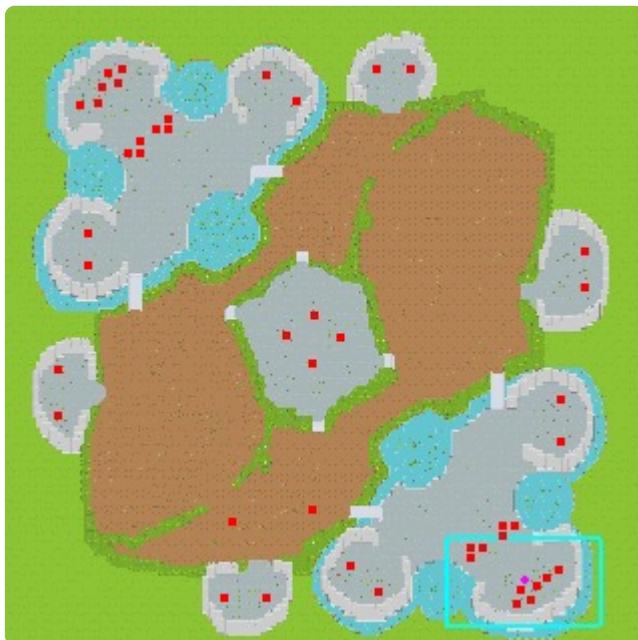


Figure 35: Minimap

Une mini-map permet d'avoir une vision réduite de la map avec uniquement les informations cruciales.

Les points rouges montrent où sont situés les unités sur la map. Le rectangle turquoise simule la vision de l'utilisateur qu'il a sur la map.

Pour pouvoir mettre la position des unités sur la mini-map, il faut appliquer des transformations pour les mettre à l'échelle. En plus de cette modification, un certain nombre d'autres transformations doivent être faites (alignement à la taille de la mini-map, la résolution de l'écran, changement de l'origine,...). C'est la classe `MiniMap` qui traite la gestion et la mise à l'échelle des coordonnées des objets à afficher.

5.3.5.2. Panel actions et sélections

La gestion des panels d'actions et de sélections se fait dans des systèmes. le système `ButtonTableSystem` et `SelectionTableSystem` utilisent une classe `Vignette` qui permet de créer des boutons qui sont liées à une entités et/ou à une action. A chaque fois qu'une nouvelle entité est sélectionnée, ses actions possibles sont mises à jour par le système `ButtonTableSystem`.

`SelectionTableSystem` récupère toute les entités sélectionnées et affiche les vignettes correspondantes.

A gauche, l'unité sélectionnée et à droite, les différentes actions possibles avec cette entité (construction d'une base et d'une caserne). Diverses informations peuvent également y figurer comme dans le cas présent, les points de vie.

6. Récapitulatif

La dernière itération du framework comporte un total de 122 classes dont 59 appartiennent au package ECS (34 components, 22 systèmes et 3 utilitaires). Yarts contient 11 classes supplémentaires.

Les 8 patterns suivants ont été utilisées:

- **Flyweight:** `TexturesManager`
- **Singleton:** Classes suffixées de `Manager`
- **Facade:** classes suffixées de `Base` ainsi que `Map`, `EcsHelpers` et `Mapper`
- **Adapter:** classes internes chargées de la conversion d'unités
- **Factory:** classes suffixées de "factory" dans le package "yarts"
- **Abstract factory:** classes suffixées de "factory" dans le package "framework"
- **Observer:** classes suffixées de "listener"
- **Command:** classes suffixées de "event"

Un nombre totalement incalculable d'heures a été passé à réfléchir, discuter, chercher, lire, relire, et coder.

6.1. Objectifs

L'objectif principal était de pouvoir mettre en place les fondations d'un jeu de stratégie en temps réel avec ses mécaniques de base. Les objectifs suivants sont atteints:

- Afficher une air de jeu
- Sélectionner des unités
- Déplacement des unités
- Avoir un système de combat
- Créer de nouvelles unités
- Créer de nouveaux bâtiments
- Système économique basé sur des ressources à collecter
- Interface utilisateur:
 - Affichage d'une mini carte
 - Affichage des ressources
 - Affichage de la population
 - Panel d'actions spéciales des unités
 - Curseur intelligent (mais pas esthétique pour le moment)

Les objectifs principaux suivant n'ont pas été atteints:

- Menu principale
- Déplacement amélioré (Pathfinding)

En revanche les objectifs secondaires suivants ont été atteints:

- Traiter l'aspect esthétique du jeu
- Contexte de jeu

La gestion de déplacement des unités n'a pas eu un bon résultat avec l'utilisation actuel d'algorithme de pathfinding. Il n'est pas toujours très précis et ne couvre pas toutes les possibilités du terrain. Toutefois, les entités ont un algorithme "fallback" pour se déplacer et ne resteront jamais figées.

L'affichage du brouillard de guerre ne fonctionne pas. Il y a actuellement un début de développement mais il ne permet pas d'être afficher correctement avec le reste du jeu.

le menu principal n'a pas pu être mise en place. Le manque de temps et le développement d'autres fonctionnalités plus a empêché sa création.

La création de musique et bruitage est un plus non négligeable dans un jeu. Nous avons les outils et la connaissance nécessaire pour les créer et les implémenter mais malheureusement pas suffisamment de temps.

6.2. Bugs

La liste est longue et n'est pas particulièrement documentée. La nature changeante et instable de l'architecture ne favorise pas la rédaction d'issues.

Les mécanismes les moins stables sont désactivés dans la version présentée et cette dernière jouit donc d'une certaine stabilité. Cette version est celle sur laquelle a été faite la vidéo de présentation fournie avec le présent document.

6.3. Améliorations

Le framework ainsi que le jeu sont au début de leur développement. Beaucoup d'améliorations sont possibles dont voici une liste non-exhaustive:

- Brouillard de guerre
- Pathfinding
- Ajout des conditions de victoire / défaite
- Menu principal
- Ajout d'options (résolutions, son, qualité des graphismes,...)
- Ajout d'un mode multijoueur (en ligne)
- Ajout de raccourcis clavier
- Ajout d'un adversaire IA
- Création de scénario (campagne)
- Nouvelles unités/bâtiments
- Équilibrage du jeu

7. Conclusion

Faire un jeu vidéo est une activité passionnante. C'est un *sport*² complet qui couvre de nombreux domaines de la formation d'un ingénieur en développement logiciel. Les erreurs furent nombreuses et les leçons légion.

Il est difficile d'être satisfait du résultat. Oui, c'est beau et ça à l'air bien mais en tant qu'auteurs du programme il est bien difficile d'ignorer tout ce qui pourrait encore être fait.

Pour nous, les objectifs sont atteints et même si certaines features ne sont pas fonctionnelles, il est important de noter que la partie Yarts du projet a été réalisée en un temps très court. Le gros du développement s'est passé sur le framework. Une fois celui-ci dans un certain état d'avancement il a été remarquablement facile d'implémenter de nombreuses fonctionnalités. Par exemple, le système de construction de bâtiments ainsi que la production d'unités tout entiers furent implémentés dans les heures qui ont précédé la présentation du projet le 3 juin.

1. Explication de la référence

Relativement tôt dans le développement du framework cette "caractéristique" fut pressentie. La capacité qu'il aurait, une fois au point, de rapidement permettre de développer un jeu.

On ne devrait pas parler de pressentir des choses dans notre domaine d'industrie, les choses devraient être robustes et planifiées et non laissées au hasard des intuitions. Disons qu'on a eu de la chance, et acceptons que "expérience" est le nom que l'on donne à ce genre d'écart de la voie toute tracée et que leur somme fera de nous des développeurs plus aptes à mieux planifier.

8. Ressources

8.1. papier

- Lee Stemkoski - Beginning Java Game Development with LibGDX
- Patrick Hoey - Mastering LibGDX Game Development

8.2. internet

8.2.1. LibGDX

- Centralisation de tutos et articles <https://github.com>
- Game from scratch <https://www.gamefromscratch.com>

8.2.2. Développement de jeux en général

- Game programming patterns : <http://www.gameprogrammingpatterns.com/>
- Erreurs classiques (fr) : <http://conquerirlemonde.com/blog/index-des-articles/>
- Gamasutra : <https://www.gamasutra.com/>
- Redblobgames (algorithmes en détail) : <https://www.redblobgames.com/>

8.2.3. Etat de l'art

- The Design of StarCraft II
- StarCraft II: Building On The Beta
- 1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond
- Successful Playtesting In Swords & Soldiers
- Postmortem: Ronimo Games' Swords & Soldiers
- The End of RTS? A Command & Conquer 4 Interview

8.2.4. ECS

- Introductif: <https://www.gamedev.net/>
- Overwatch gameplay architecture and ECS: <https://www.youtube.com>
- ECS in C++ : <https://www.gamasutra.com/>
- ECS by Mark Jordan : <https://medium.com>
- LibGDX and Ashley : <https://www.gamedevelopment.blog>

8.2.5. Steering behaviors

- Introductif : <http://www.simoncoenen.com>
- Introductif : <https://gamedevelopment.tutsplus.com>
- Collision avoidance : <https://gamedevelopment.tutsplus.com>
- Autonomous movement : <http://fightingkitten.webcindario.com>

8.2.6. Pathfinding

- Introductif : <https://happyCoding.io/tutorials/libgdx>
- Heuristic: <https://theory.stanford.edu>

8.2.7. Brouillard de guerre

- Introductif: <https://www.programcreek.com>

8.2.8. Level Design

- Simplex Noise : gamedevelopment.blog
- Noise génération : redblobgames.com
- Perlin noise : stackoverflow.com
- Random noise : lodev.org
- Map from noise: redblobgames.com
- Random map generation: gamedevelopment.blog
- Map generator: github.com
- More on Map generator: gpdev.net

8.2.9. Ui Design

- Basic : [gamedevelopment.blog](#)
- Scene2D : [gamedevelopment.blog](#)
- Scene2D-c : [gamedevelopment.blog](#)
- Vis-ui : [github.com](#)
- Ui-skins : [github.com](#)
- Test-ui : [github.com](#)

8.2.10. Repo des mini projets faits pour se familiariser avec LibGDX

- Starfish-Collector : https://github.com/RoscaS/libGDX_Starfish-Collector
- Adventure-Game : https://github.com/RoscaS/libGDX_Adventure-Game
- Space-Rocks : https://github.com/RoscaS/libGDX_Space-Rocks
- MarioBros : https://github.com/RoscaS/libGDX_MarioBros
- Jigsaw-Puzzle : https://github.com/RoscaS/libGDX_Jigsaw-Puzzle
- Jumping-Jack : https://github.com/RoscaS/libGDX_Jumping-Jack
- Ashley : https://github.com/RoscaS/libGDX_Ashley

9. Annexes

-
1. Presentation-YARTS.mp4
 2. yarts.jar