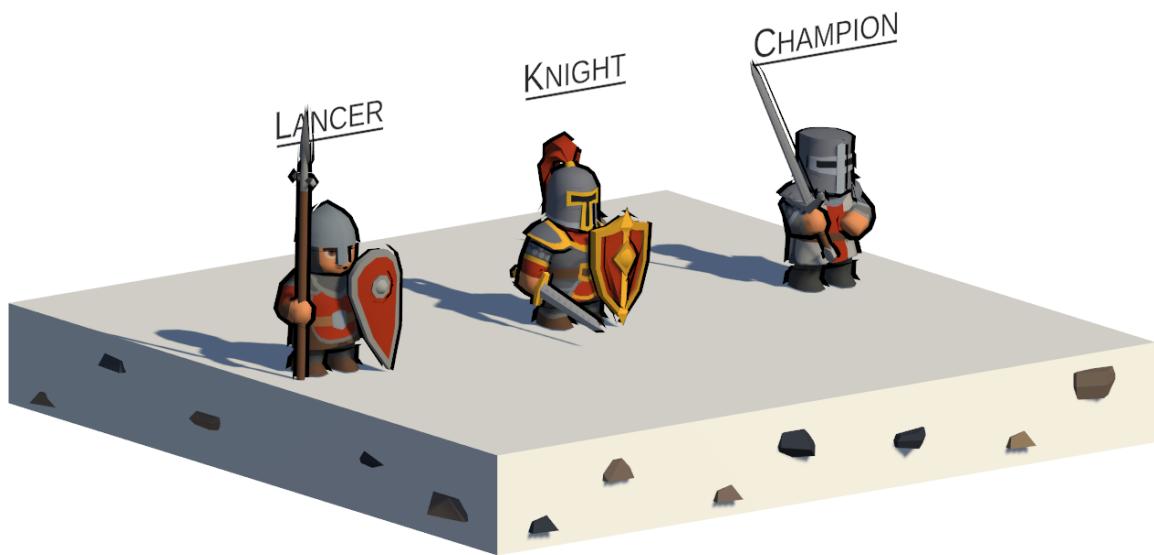


# YAARTS

## Infographie



Sol Rosca, Nathan Latino, Tristan Seuret (INF3b)

Mai 2020

# Table des matières

<b>1. Introduction</b>	<b>1</b>
<b>2. Contextualisation</b>	<b>2</b>
2.1. Cours de .NET	2
2.2. Cours d'infographie	3
2.3. Rapport	4
<b>3. Analyse</b>	<b>5</b>
3.1. Objectifs	5
3.1.1. Prototype utilisable	6
3.1.2. MVP	7
3.1.3. Utopique	7
3.2. Répartition des tâches	8
3.2.1. Sol Rosca (Architecture)	8
3.2.2. Nathan Latino (GUI)	8
3.2.3. Tristan Seuret (Fog of war et URP)	8
3.3. Travail collaboratif	9
<b>4. Conception</b>	<b>12</b>
4.1. Spécifications	13
4.1.1. Entités	13
4.1.1.1. Type Character	14
4.1.1.2. Type Structure	16
4.1.1.3. Type Collectible	17
4.1.2. Gestions des inputs	17
4.1.3. Pipeline de rendu Unity	18

<b>5. Réalisation</b>	<b>19</b>
<b>5.1. Architecture</b>	<b>19</b>
<b>5.1.1. Problématique</b>	<b>19</b>
<b>5.1.2. Approche héritage</b>	<b>21</b>
<b>5.1.3. Approche Components (composition)</b>	<b>22</b>
<b>5.1.3.1. Utilisation de Component dans le web</b>	<b>22</b>
<b>5.1.3.2. Utilisation de Component dans les jeux vidéos</b>	<b>23</b>
<b>5.1.3.3. Component, Strategy et Facade</b>	<b>25</b>
<b>5.1.3.4. Unity gameobject</b>	<b>26</b>
<b>5.1.4. Approche ECS</b>	<b>26</b>
<b>5.1.4.1. Component</b>	<b>26</b>
<b>5.1.4.2. Entity</b>	<b>27</b>
<b>5.1.4.3. System</b>	<b>27</b>
<b>5.1.4.4. Unity DOTS</b>	<b>27</b>
<b>5.1.5. Modèle développé</b>	<b>27</b>
<b>5.1.5.1. Première itération: Approche programmatique</b>	<b>28</b>
<b>5.1.5.2. Seconde itération: Approche prefabs</b>	<b>29</b>
<b>5.2. Mécanismes</b>	<b>30</b>
<b>5.2.1. Anatomie d'une Entity</b>	<b>30</b>
<b>5.2.2. Brouillard de guerre</b>	<b>32</b>
<b>5.2.2.1. Solution proposée</b>	<b>32</b>
<b>5.2.2.2. Modèle développé</b>	<b>33</b>
<b>5.2.2.2.1. Fog of War standard</b>	<b>33</b>
<b>5.2.2.2.2. Fog of War découvert</b>	<b>34</b>
<b>5.2.2.3. Integration</b>	<b>35</b>
<b>5.2.2.4. Avantages et inconvénients</b>	<b>36</b>
<b>5.2.2.5. Résultats et performances</b>	<b>36</b>
<b>5.2.2.6. Améliorations</b>	<b>37</b>
<b>5.2.2.7. Références</b>	<b>37</b>

<b>6. Interface Utilisateur</b>	<b>38</b>
6.1. Expérience et interface utilisateur	38
6.2. Solutions proposées	40
6.2.1. Unity UI (UGUI)	40
6.2.2. Next-Gen User Interface (NGUI)	40
6.2.3. User Interface Elements (UIElements)	41
6.2.4. Immediate Mode Graphical User Interface (IMGUI)	41
6.2.5. lien avec le/les Ch. d'infographie	42
6.2.6. Modèle développé	42
6.2.6.1. Gestion des actions	42
6.2.6.2. Minimap	43
6.2.7. Relation avec les 3 piliers de l'infographie	43
6.2.8. Architecture Unity	44
6.2.8.1. Présentation	44
6.2.8.2. Avantages et inconvénients	45
6.3. Résultats	45
<b>7. Assets</b>	<b>51</b>
7.1. Tools	51
7.1.1. Free	51
7.1.2. Payed	51
7.2. Assets	51
7.2.1. Payed	51
<b>8. Références</b>	<b>52</b>
8.1. Architecture	52
8.1.1. Bibliographie	52
8.1.2. Webographie	52
8.2. Fog of war	52
8.2.1. Webographie	52
8.3. GUI	53
8.3.1. Bibliographie	53
8.3.2. Webographie	53
<b>9. Annexes</b>	<b>54</b>

# 1. Introduction

---

YAARTS: Yet Again Another Real Time Strategy (Game). Est un prototype de jeu de stratégie en temps réel (Dune 2, Command & Conquer, Warcraft, Starcraft, ...) réalisé conjointement pour les projets de fin d'année des cours d'infographie et de .NET.

Le terme *jeu de stratégie en temps réel* (ou RTS: Real Time Strategy) 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 à beaucoup évolué mais principalement graphiquement. En effet, bien que la définition précise fasse l'objet de débats, les RTS sont traditionnellement définis par les verbes "**récolter**", "**construire**" et "**détruire**" en plus d'être des jeux où l'action se déroule en temps réel entre les différents participants. En partant de ces trois verbes, il est facilement déductible 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. Ces trois critères sont donc naturellement les objectifs du design de YAARTS.

## 2. Contextualisation

Ce projet est en essence inspiré par ce qui a été fait au semestre de printemps 2019 pour le projet P2 Java: YARTS (avec un seul "A", liens en annexe). Ce projet fut une premier tentative d'implémentation d'un RTS. Sans aucune expérience initiale avec la *technologie utilisée*<sup>1</sup> et encore moins avec les techniques d'infographie et fût l'occasion de cerner le problème et d'avoir une idée clair de quel sont les principaux obstacles et pièges à éviter.

YAARTS (avec deux "A") malgré sa dimension supplémentaire a pour but d'aller plus loin que le premier et de proposer un prototype de jeu de stratégie qui implémente toutes les fonctionnalités qui *définissent*<sup>2</sup> le genre le tout dans un style visuel agréable et cohérent.

### 2.1. Cours de .NET

Un jeu vidéo est un sport complet qui demande de nombreuses qualités sur lesquelles le cours de .NET a mit l'accent. En effet, dans un jeu vidéo, de nombreux composants hétéroclites coexistent et communiquent. Dans ces circonstances et à cette échelle, de nombreux problèmes qui dans le cadre de projets moins vastes n'ont pas le temps de surgir, ont ici tout le loisir de se dévoiler et de bloquer la progression. Pour palier à ça, il est impératif d'observer une certaine rigueur de bout en bout et c'est bien là un des points clé qui est revenu dans tous les travaux du cours de .NET.

Au vue des contraintes temporelles et de l'ambition dans les objectifs, il est nécessaire d'avoir une idée clair de ce que doit faire l'application ainsi que du niveau de finition souhaité. Le premier point permet de réfléchir à une architecture cohérente et le second à définir à partir de quel niveau il est possible de travailler sans plan car les conséquences de code de moins bonne qualité aura peu ou pas d'impacte n'ayant pas vocation à être étendu. Cette façon de faire permet d'avoir une base solide et robuste qui peut soutenir du code plus volatile et expérimental. De plus, cette façon de faire est cohérente dans le contexte d'un projet à cheval sur deux matières dont l'une qui est en pleine phase de découverte au niveau de ses concepts.

Le but de la partie C# de ce projet est donc de créer un moteur de jeu de stratégie en temps réel suffisamment robuste pour qu'il puisse servir de terrain d'expérimentation pour la partie infographie du projet.

---

1. LIBGDX: Un framework Java

2. Décris dans l'introduction

## 2.2. Cours d'infographie

Le projet décrit dans ce rapport étant un jeu vidéo, les liens avec le cours d'infographie sont nombreux. De façon générale, les notions du premier semestre sont la base de culture générale nécessaire à pouvoir naviguer sereinement dans les problèmes plus haut niveau que pose Unity. La matière du second semestre quant à elle est l'aperçu nécessaire pour pouvoir entamer un projet plus conséquent que quelques scripts qui se courrent après.

Même si la partie purement technique du pipeline OpenGL n'a pas été directement utilisée, les notions liées aux *shaders* et leurs déclinaisons *vertex* et *fragments* sont au cœur du développement de jeux vidéos. Malgré leur caractère bas niveau, ces notions reviennent dans de nombreuses discussions qui touchent des aspects plus haut niveau et dont il ne serait pas possible de saisir la pertinence ou les nuances sans avoir une idée du contexte dans lequel elles s'inscrivent.

Les notions de scène et de caméra sont fondamentales et reviennent systématiquement. Le fait de bien comprendre la notion de repère est un gain cognitif indispensable pour comprendre de nombreuses explications où les conversions sont faites implicitement en assumant que le lecteur jongle avec ces notions. De la même façon, une bonne compréhension des projection et de leur spécificités ainsi que la capacité de visuellement les différencier sont autant de place mentale libérée pour pouvoir résoudre d'autres problèmes qui assument que ces notions sont acquises.

Au travers du projet dont fait état le présent rapport, une bonne partie des notions du premier semestre ont été touchées mais au travers du prisme d'Unity qui apporte une vision considérablement plus haut niveau. Cette perspective nouvelle apporte des explications sous la forme de complément là où les questions n'était pas forcément évidentes à formuler dans le contexte bas niveau du premier semestre. Typiquement, de part leur simplicité à mettre en place dans Unity, les différents types d'éclairages, leurs attributs et leur relation avec la physique permet de gagner un niveau d'abstraction qui automatiquement fait assimiler un certain nombre de concepts plus basiques. De même, les outils que propose Unity pour visualiser et jouer avec les matériaux, les modèles et les textures permet d'enclencher les notions d'UV (Coordonnées de textures dans le cours), de normales, les déformations causées par les application de textures et de leur comportement vis-à-vis de la lumière.

Autrement dit, de façon plus pragmatique, le cours d'infographie donne une vision suffisamment globale de la matière pour permettre entre autres de ne pas perdre de temps à lire un article sur la gestion des ombres dans l'herbe via programmation de shader alors que ce qui est recherché est le type d'éclairage qui serait le plus pertinent pour une scène.

## 2.3. Rapport

Ce projet étant une proposition personnelle et ne faisant pas partie des projets distribués en cours, sa structure n'est pas la même. Pendant toute la réalisation du projet un effort a été fait pour être le plus organisé possible en vue de la rédaction anticipée d'un rapport final. Pour cette raison, nous jugeons plus pertinent de le structurer dans ses grandes lignes comme un rapport traditionnel (analyse, conception, réalisation) tout en tentant d'intégrer les consignes de segmentation du Pr. Gobron.

## 3. Analyse

Le périmètre d'un jeu vidéo étant très vaste et les développements prenant régulièrement des années pour des équipes professionnelles, il est nécessaire d'être réaliste dans les objectifs visés pour qu'ils restent cohérents avec le contexte académique du projet. Ainsi, ce chapitre a pour but d'expliquer la démarche suivie pour définir le projet ainsi que ses objectifs abstraits.

### 3.1. Objectifs

Le but est donc de créer un **prototype utilisable** de RTS et pour ce faire, d'abord sera écarté le gros de la notion d'adversaire qui sera réduite au strict minimum. Les objectifs sont donc essentiellement axés autour des trois verbes "**récolter**", "**construire**" et "**détruire**", introduits dans les premières lignes de ce rapport.

À ces trois verbes viennent s'ajouter les trois notions suivantes:

- **Actions semi autonome**: Cela implique que les entités du jeu sont capables d'une certaine autonomie. En effet, une fois un ordre donné par le joueur, l'entité doit être capable d'exécuter cet ordre en prenant en charge de gérer la diversité des obstacles qui le séparent de l'accomplissement de l'ordre. Pour ce faire, les entités doivent faire preuve d'une certaine intelligence, d'une capacité d'adaptation et doivent savoir quoi faire en cas d'impossibilité d'exécuter l'ordre.
- **Temps réel**: Par opposition à tour par tour, la notion de temps réel implique que le temps passe de façon continue et n'est pas échantillonné et partagé entre les joueurs comme pour un jeu de plateau ou un jeu de cartes traditionnel. Peu importe ce que fait le joueur, certaines actions se déroulent en arrière plan "en même temps". Cela implique qu'il est nécessaire d'avoir des mécanismes qui assurent que ces actions seront menées à bien sans forcément demander d'intervention implicite du joueur.
- **Exploration**: Ou autrement dit, le fait de dissimuler l'objet de l'exploration et implique donc la nécessité de la mise en place de mécanismes qui permettent de cacher des informations au joueur pour que ce dernier puisse les découvrir en exécutant certaines actions.

### 3.1.1. Prototype utilisable

Il est important de définir ce qui est entendu par "prototype utilisable". Dans le présent contexte, un prototype utilisable est une application interactive qui permet au testeur, guidé par les développeurs de se faire une idée de ce à quoi pourrait ressembler le produit dans un niveau de finition plus complet. Les limitations sont donc nombreuses et l'application ne fonctionne que dans un cadre restreint.

Autrement dit, le jeu n'est pas jouable au sens *fun* du terme mais il est testable et peut être le support d'une démonstration.

De plus, il est important de signaler que l'implémentation tourne autour des mécaniques essentielles et non pas autour des règles du jeu. Ça veut dire que le jeu n'est aucunement balancé et n'a pas de bût ou de conditions de victoire. Son style graphique et son "contexte historique" sont uniquement motivés par leur accessibilité et n'ont pas été le fruit d'une réflexion particulière.

### 3.1.2. MVP

- Une carte
- Une faction
- Entités:
  - Offensives (mêlée, distance)
  - Bâtiments (production)
  - Récolte (ressources)
- IA des entités:
  - Pathfinding
  - Collision avoidance (gestion des obstacles mobiles dans le pathfinding)
  - Système d'états
  - Gestion de la ligne de vue et de la portée
  - Gestion des actions
  - Gestion des interactions
  - Gestion de la durabilité
  - Système d'appartenance (à un joueur ou à l'ordinateur)
- Mécanisme de sélection des entités (simple et multiple)
- Gestion des ressources
- Gestion de la caméra
- Brouillard de guerre
- GUI
  - Une mini carte
  - Affichage des ressources
  - Affichage de la population
  - Panneau de sélection
  - Panneau des actions de la sélection
  - Curseur intelligent dont la forme dépend du contexte

### 3.1.3. Utopique

- IA qui joue l'adversaire
- Diversifications des cartes de jeu
- Génération procédurale de cartes
- Éléments de gameplay (technologies, amélioration des unités, variété dans les unités)
- GUI moins prototype
- Tout le reste

## 3.2. Répartition des tâches

Cette partie décrit la répartition des tâches au sein du projet.



Concernant la rédaction du rapport chaque membre s'est occupé des parties concernant son travail et Sol Rosca a pris en charge les parties communes.

### 3.2.1. Sol Rosca (Architecture)

- Architecture générale
- Architecture des entités
- Mécanisme de sélection
- IA des entités
- Unités offensives
- Gestion des ressources
- Unité de prod/récolte
- Bâtiment de production
- Communication moteur-gui

### 3.2.2. Nathan Latino (GUI)

- Menu principal
- Mini-carte
- Affichage des ressources
- Affichage de la population
- Panneau de sélection
- Panneau d'action de la sélection

### 3.2.3. Tristan Seuret (Fog of war et URP)

- Étude des solutions de versioning
- Étude des pipelines graphiques d'Unity (Standard et URP)
- Fog Of War
- Curseur intelligent dont la forme dépend du contexte si possible

### 3.3. Travail collaboratif

Après quelques recherches, il s'est avéré que l'utilisation d'un git tel que Github n'est pas adapté à Unity. En effet, Unity est un programme très graphique et l'utilisation d'un logiciel de versioning pour des fichiers textes n'est pas adaptée. De plus, Unity générant un nombre important de fichiers, les conflits de *merge* sont plus délicat à gérer qu'à l'accoutumée. Il existe néanmoins des méthodes et des configurations permettant de travailler collaborativement sur un projet Unity. Cependant, suite à ces recherches, Unity Collaborate fût découvert et s'est vite imposé comme une solution évidente.

Unity, conscient de la nécessité d'offrir un moyen de collaborer sur le même projet, a développé un service similaire à Git permettant la synchronisation et le versioning d'un projet. Ce dernier se nomme Unity Collaborate et est inclus dans le [Github Student Developer Pack](#) depuis le mois de Janvier 2020. Ce dernier offre plusieurs avantages notables par rapport à Git :

- Facilité d'initialisation L'utilisation d'un git aurait demandé la configuration de ce dernier, la configuration de Unity ainsi que la configuration de Git LFS (**Large File System**). De plus, il est directement intégré au sein de Unity et permet de facilement récupérer les derniers changements.
- Cloud build Lors de l'utilisation de Unity Teams (qui contient Unity Collaborate), il est possible de build le projet directement sur les serveurs Unity.



Figure 1 Cloud build interface

Comme il est possible de le remarquer sur cette image, une build demande en moyenne une vingtaine de minutes pour être créée. Non seulement les créées sur les serveurs Unity permet de s'affranchir d'un poids, il permet également de facilement distribuer les builds à l'entièreté de l'équipe et d'en garder une traçabilité.

- Modification de fichiers Comme expliqué précédemment, la modification concurrente d'une scène pose d'énormes problèmes de conflits lors de *merge*. Unity Collaborate, totalement intégré à Unity, permet d'afficher qu'un fichier est en cours de modifications par une autre membre de l'équipe et permet d'éviter ce genre de conflit.

The screenshot shows a 'In Progress' screen from Unity Collaborate. At the top, it displays the project name 'RTS' and a unique identifier. On the right, there's a 'Add Integration' button. Below this, a note explains what 'In Progress' means: 'Within the Editor, In-Progress displays when another user has unpublished changes to a scene or prefab. In some cases you may want to remove In-Progress for users who are no longer on the project or who have unpublished changes on duplicate copies of the project no longer in use.' A note below states that clearing the notification won't affect the project files. The main table lists modifications made by five users:

User	Timestamp	Project Path	Files In-Progress
Tristan Seuret	Fri, May 15, 2020 9:32 PM	E:/_COURS/Unity/RTS_Unity/RTS	Show 2 files <button>Clear</button>
sol rosca	Mon, Mar 30, 2020 7:23 PM	/home/sol/Code/Unity/RTS/RTS (1)	Show 1 files <button>Clear</button>
Latino Nathan	Tue, May 5, 2020 12:09 PM	F./Nathan/Documents/unity/projects/RTS	Show 1 files <button>Clear</button>
sol rosca	Sun, May 17, 2020 7:02 PM	C:/Code/Unity/RTS	Show 9 files <button>Clear</button>
sol rosca	Thu, Mar 5, 2020 12:06 AM	C:/Code/Unity/RTS - Copy	Show 2 files <button>Clear</button>

Figure 2 informations modification en cours d'un fichier

- Centralisation des fonctionnalités

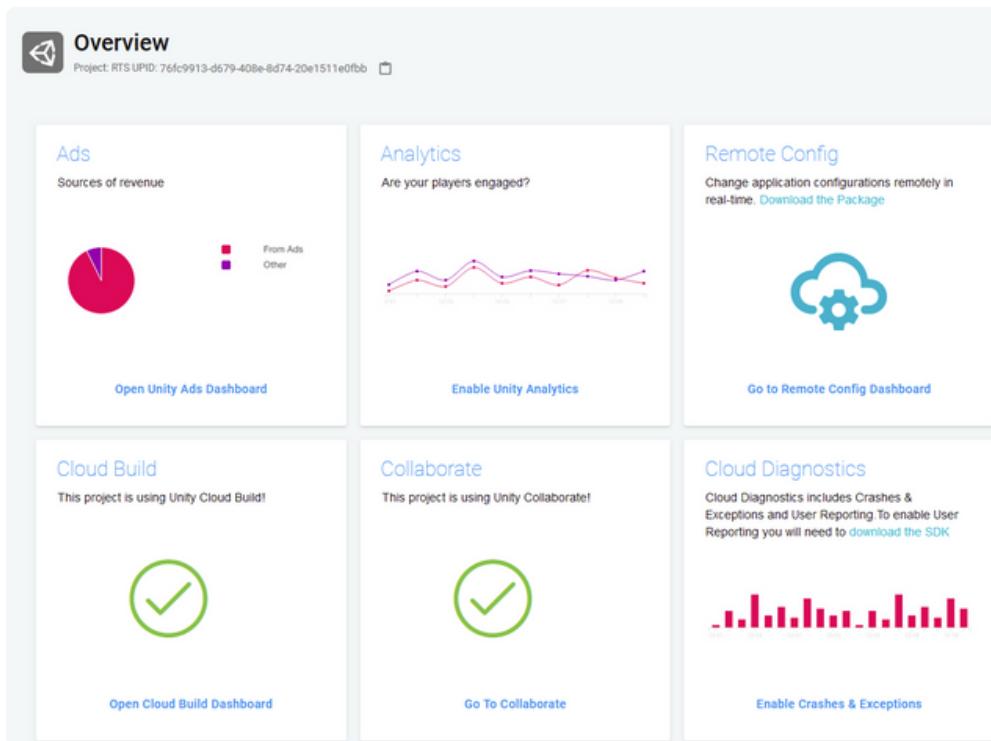


Figure 3 Dashboard Unity

De plus, l'utilisation de Unity Collaborate permet d'avoir un point central pour toutes les informations du projet. Que ce soit les retours utilisateurs, le versioning ou encore les builds, tout est accessible à un seul endroit.

Cependant, Unity Collaborate ne propose aucun système de branches comme il est disponible dans les autres logiciels de versioning.

De ce fait, afin d'éviter les conflits lors de merge ou autre désagrément, lorsqu'une personne souhaitait modifier la scène et tester sa solution, il faisait une copie de la scène localement. Ensuite, une fois la fonctionnalité aboutit, elle était réimplémenter dans la scène final en s'assurant que personne d'autre ne fasse de modification sur la scène.

## 4. Conception

Ce chapitre couvre la concrétisation des abstractions du précédent chapitre. Il énonce quelles sont les spécifications techniques du projet ainsi que les moyens techniques mis en œuvre pour les réaliser.

La consigne d'un rapport détaillé étant arrivé tard de nombreuses parties du logiciel n'ont pas été scrupuleusement documentées. Vu la charge de travail et le temps à disposition il n'aurait pas été possible de faire plus. Donc voici une liste non exhaustive des mécanismes implémentés qui ne seront pas couvert comme ils le devrait dans ce rapport:



- Logique d'affichage du rectangle de sélection
- Affichage de la sélection
- Barres de vie / progression
- Système de gestion des meshes
- Système de gestion des particules
- Système d'animation étendu
- Gestion des collision
- ...

- **Plateforme:** PC
- **Os:** Windows & Linux
- **Langage:** C#
- **Vue:** 3D
- **Moteur de rendu:** Unity3D
- **Camera:** Perspective
- **Contrôle:** Clavier + souris
- **Genre:** Stratégie
- **Sous-genre:** Temps réel
- **Type de joueurs:**
  - Humain (`Player`)
  - Ordinateur (`CPU`)
- **Contexte scénaristique:**
  - Médiéval
  - Toon
- **Économie:**
  - Récolte de ressources



Figure 4 Structure: Town Center

## 4.1. Spécifications

Dans cette partie seront développé et mis en termes techniques les objectifs du précédent chapitre en commençant par une fiche technique générale pour caractériser plus précisément le projet.

### 4.1.1. Entités

Les entités sont de trois types différents qui remplissent chacun un rôle spécifique:

- Character
- Structure
- Collectible

Ces trois types ont en commun quatre caractéristiques principales :

- Le fait d'avoir un **propriétaire**
- Le fait d'être **sélectionnable**
- Le fait de posséder un certain nombre de **points de "quelque chose"** (vie, durabilité, ressource, ...) qui quantifie une durée de vie
- Le fait de posséder **un état**.

Une entité appartient donc à un joueur, à l'ordinateur ou à la carte (gaïa) et une fois ses points épuisés, elle change d'état et passe dans un état où elle n'est plus sélectionnable (et par extension, n'est plus une entité).



Figure 5 Collectible: Gold



P.ex. les éléments de décor ou les projectiles ne sont pas des entités car elles ne possède pas ces 4 caractéristiques. Ce sont néanmoins des objets du jeu qui possède des attributs et servent des fonctions, ce ne sont juste pas des entités.

C'est les caractéristiques particulières d'une entité qui définissent son **type**. Aussi, un type peut posséder des **déclinaisons** dans le cas où sa façon d'interagir avec le monde, ses états ou sa façon de gérer ses points n'est pas générique. Finalement, la dernière variation au sein d'un type est appelée **classe** et ne diffère des autres membres de son type que par les valeurs de base de ses attributs logiques et/ou esthétiques. **Dans ce projet, une classe est la concrétisation d'un type.**

#### 4.1.1.1. Type Character

Un Character:

- **Est mobile :**
  - Pathfinding
  - Peut être nulle pour un certain temps
  - Mémoire
- **Est conscient du monde qui l'entour**
  - Zone de perception (Ligne de vue)
  - Gestion des collisions (directes & perception),
  - Mémoire



**Figure 6** Structure: Castle

- **Est capable d'interagir:**
  - Avec le sol (un point vide de la carte) et avec les autres entités (un point occupé par une autre entité)
  - En fonction du monde qui l'entour ou d'un ordre du propriétaire
- **Est produit par un bâtiment:**
  - En fonction de sa déclinaison
  - A un coût

Les personnages se déclinent de 3 façons en fonction de leur façon d'interagir avec le monde et chaque déclinaison possède au moins une classe:

- **Melee:** Knight, Champion, Lancer
- **Ranged:** Archer, Wizard
- **Worker:** Worker

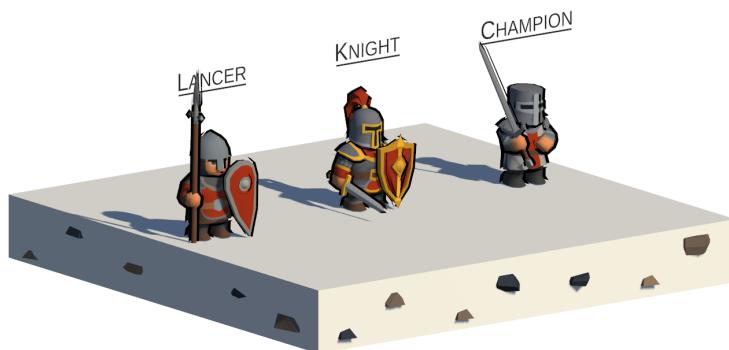


Figure 7 Character: Melee



Figure 8 Character: Ranged



Figure 9 Character: Worker & Collectible: Tree

#### 4.1.1.2. Type Structure

Une Structure:

- **Est statique:**
  - Placé par le joueur
- **Est conscient du monde qui l'entour:**
  - Zone de perception (Ligne de vue)
  - Gestion des collisions (directes & perception),
  - Mémoire
- **Possède des actions:**
  - En fonction de sa classe
  - Production d'Entités mobiles
- **Est produit par un ouvrier:**
  - A un coût

Il y a 4 classes de bâtiments, chacune spécifique à la déclinaison de personnages qu'elle produit:

- **TownCenter**: Worker
- **Caserne**: Knight, Lancer
- **Archery**: Archer
- **Castle**: Champion, Wizard

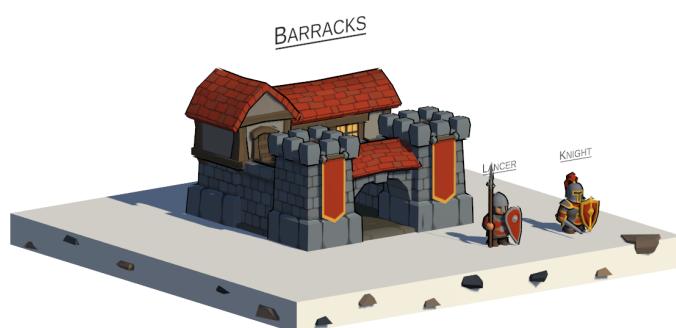


Figure 10 Structure: Barracks

#### 4.1.1.3. Type Collectible



Le type ressource est le nom de l'entité qui fournit ce type de ressource. Ainsi, l'entité Tree de type Ressource permet de récolter la ressource bois.

Un Collectible:

- **Est statique :**
  - Appartient à la carte (Gaïa)
  - Plusieurs instances
- **Possède un type de ressource :**
  - Quantité limitée
  - Récoltable par un ou plusieurs Worker

Il existe 2 déclinaisons de ressources qui gèrent différemment leur points et chaque déclinaison possède une classe:

- **Gold:** GoldMine
- **Wood:** Tree

#### 4.1.2. Gestions des inputs

- Clic gauche sur une entité permet d'afficher des informations la concernant.
- Clic gauche maintenu permet de faire un rectangle de sélection qui sélectionne plusieurs entités appartenant au joueur. Si dans la sélection se trouve des entités Structure et Character, la sélection ignore les Structure.
- Clic droit sur une entité sans sélection préalable ne fait rien.
- Clic droit sur une entité avec une sélection:
  - Si l'entité n'appartient pas au joueur, un ordre d'interaction est donné au entités de la sélection.
  - si l'entité appartient au joueur, elle se déplace pour la rejoindre
- Clics droit sur la carte avec une sélection:
  - la sélection s'y rend.
- Déplacement de la camera:
  - pression sur le bouton central de la souris
  - "WASD"

### 4.1.3. Pipeline de rendu Unity

Ce projet utilise URP (Universal Render Pipeline) afin de profiter des dernières optimisations fournies par Unity.

En effet, *URP* offre plus de contrôle sur le rendu de l'application via C#, est plus optimisé et a vocation à remplacer le *Built-in Render Pipeline* dans le futur.

Ainsi, il permet d'optimiser l'application en fonction du hardware et, comme il est possible de le voir sur cette [page](#), il permet plus de granularité concernant l'implémentation de fonctionnalités au sein du pipeline.

Ce pipeline a cependant été choisi pour une raison majeure : les performances. Unity a effectué une comparaison entre ses 2 pipeline et voici le résultat :

Stats (the lower the better)	Universal Render Pipeline	Built-in render pipeline
CPU Time (ms)	12.9	20.6
GPU Time (ms)	16.6	22.6
Draw calls*	978	1314
Batches**	103	976
Bandwidth (Load/Store MB)	50.6	117

\*Difference in draw calls is due to the fact that Universal Render Pipeline doesn't render additional draw calls for each additional light. \*\*Difference in batches is due to the fact that Universal Render Pipeline batches per shader while Built-in batches per material.

Figure 11 Comparaison des performances

De plus, URP a facilité l'ajout de *cell shading* dans le projet en suivant simplement ce tutoriel fourni par Unity : [Tutoriel Toon Outlines](#)

Cependant, aucune solution n'est parfaite. *URP* ne produit des ombres que pour les lumières directionnelles, ce qui a modifier certains points du projet. De plus, il ne permet pas encore de faire des projections et ajoute des limitations au niveau des caméras. Il a donc fallu trouver d'autres méthodes voir même des shaders entier permettant de palier à ce manque comme il sera expliqué plus tard dans ce rapport.

## 5. Réalisation

### 5.1. Architecture

Les premières heures du projet ont été le théâtre de nombreuses discussions sur l'architecture du projet. Cette partie décrit les options considérées ainsi que les choix faits et les raisons qui les motive.

#### 5.1.1. Problématique

Dans un projet où de nombreux composants doivent fonctionner de concert, il est primordial de garder un couplage faible entre les différentes parties du code pour qu'il reste maintenable et suffisamment robuste pour pouvoir non seulement supporter toutes les fonctionnalités planifiées mais également être étendu par la suite.

Dans le cadre d'un jeu vidéo cette problématique est au coeur des préoccupations. En effet, la technique du "c'est bon ça fonctionne, c'est que ça doit être une bonne solution" est la meilleure façon de se retrouver plus ou moins rapidement dans un impasse avec un code qu'il n'est pas possible d'étendre car la moindre modification casse un équilibre bancal.

Pour palier à ce problème il est non seulement indispensable d'avoir une bonne conception et une architecture adaptée, mais il faut aussi toujours s'assurer que le nouveau code s'intègre dans l'architecture sans modifier sa philosophie et les mécanismes en place.

*“What is **good** software architecture? For me, good design means that when I make a change, it's as if the entire program was crafted in anticipation of it. I can solve a task with just a few choice function calls that slot in perfectly, leaving not the slightest ripple on the placid surface of the code. [Bob Nystrom](#)”*

Le problème d'une série de solutions attachées ensemble pour donner le comportement souhaité à des objets n'est pas tant dans la qualité du code des solution mais dans le façon dont communiquent ces différents morceaux de code. Autrement dit, le couplage y est sûrement très important et le point critique où il n'est plus possible d'avancer approche à chaque ligne supplémentaire. De plus à ce moment là, chaque ajout est un casse tête non pas à cause de la nouvelle fonctionnalité mais pour faire en sorte de premièrement comprendre le code à utiliser et ensuite pour assurer de ne rien casser.

Pour ne pas tomber dans le piège du code rapide, le temps a été pris pour définir les principaux composants de l'application et de quelle façon ils communiquent ensemble ce qui a donné lieu à un premier jet de l'architecture:

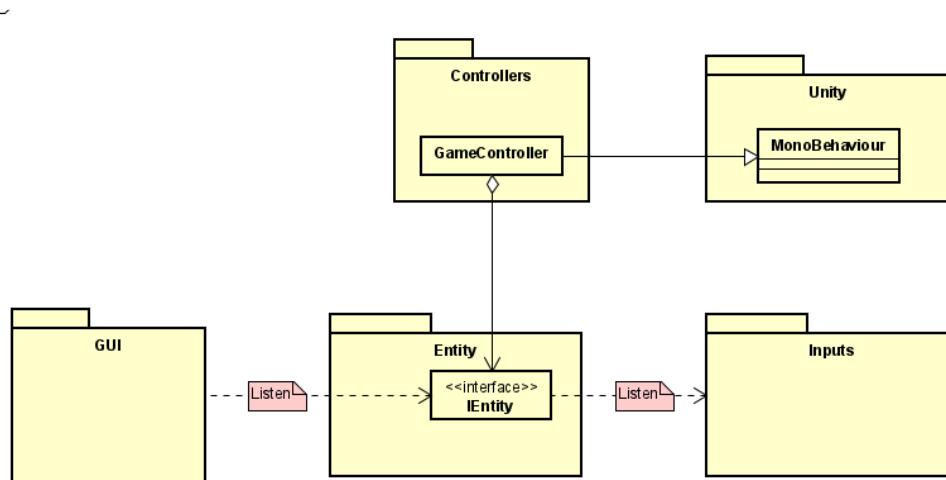


Figure 12 Premier jet

Ce premier jet fût utile pour séparer les tâches mais également pour identifier certains points chauds à étudier. Le principal étant la façon de gérer le module `Entity`. Plusieurs approches ont étées considérés et seront décrites dans les prochains points.

### 5.1.2. Approche héritage

Cette approche est naturelle, et pour peu que les relations et 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:

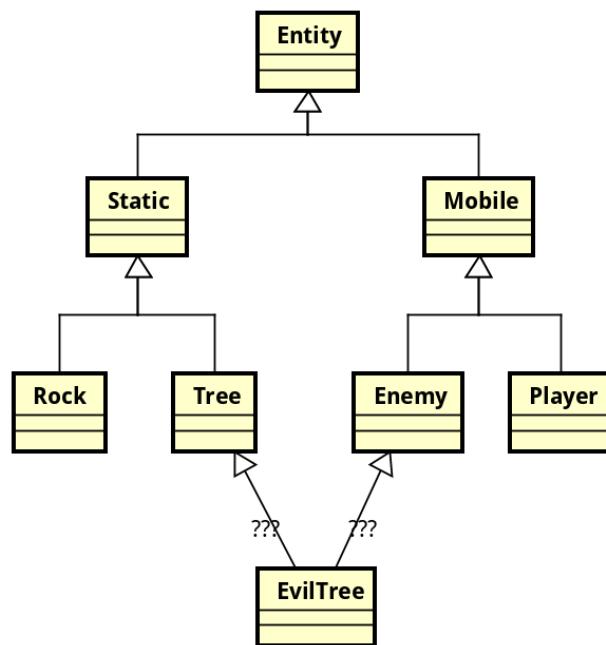


Figure 13 Le soucis avec l'héritage

Ce projet comportant justement un grand nombre d'entités aux comportements variés et l'expérience de YARTS (avec un seul "A") discalifie cette possibilité.

Pour compenser les défauts de cette approche il est nécessaire de trouver une autre façon que l'héritage pour la gestion des entités.

### 5.1.3. Approche Components (composition)

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, d'où le nom du pattern) de composants et chaque composant encapsule la logique qui le concerne ainsi qu'une référence vers l'entité qui le contient:

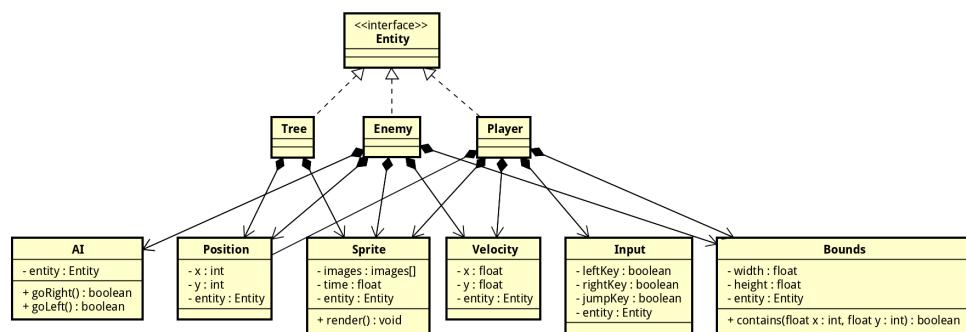


Figure 14 Pas si spaghetti que ça

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

1. Il est aisément de créer une nouvelle entité
2. Possibilité de dynamiquement ajouter ou retirer des composants (et donc de modifier le comportement) à la façon d'un pattern *Strategy*
3. Plus de performance de part une gestion plus optimisée du cache CPU

Ce pattern ne fait pas partie des patterns classiques du [Gangs of four](#) et même si il revient dans de nombreuses discussions sur Internet, il est difficile de trouver son origine. Il semble bien ancré dans le domaine du jeu vidéo et celui des interfaces graphiques ([plus particulièrement celles développées avec les outils du web](#)). Ce qu'il est intéressant de noter, c'est que même si dans les grandes lignes il s'agit du même pattern, c'est des besoins totalement différents qui justifie son emploi dans chacun de ces deux cas.

#### 5.1.3.1. Utilisation de Component dans le web

Côté web, c'est principalement pour le côté réutilisabilité des components qu'il est apprécié. En effet, les frameworks frontend [VueJS](#) et [React](#) l'emploient tous les deux. Dans ces deux frameworks, un component est un élément graphique "self contained" (auto-suffisant) qui contient logique, présentation et état dans un seul et même objet qui est utilisé sous forme de composition dans un objet responsable de leurs cycle de vie.

### 5.1.3.2. Utilisation de Component dans les jeux vidéos

Côté jeux vidéos, c'est pour le gain de performance qu'il offre vis-à-vis de l'approche héritage en accélérant les accès mémoire via optimisation de l'utilisation du cache du CPU.



"Random Access" veut dire que contrairement à un disque dur, l'accès à n'importe quelle donnée se fait à la même vitesse.

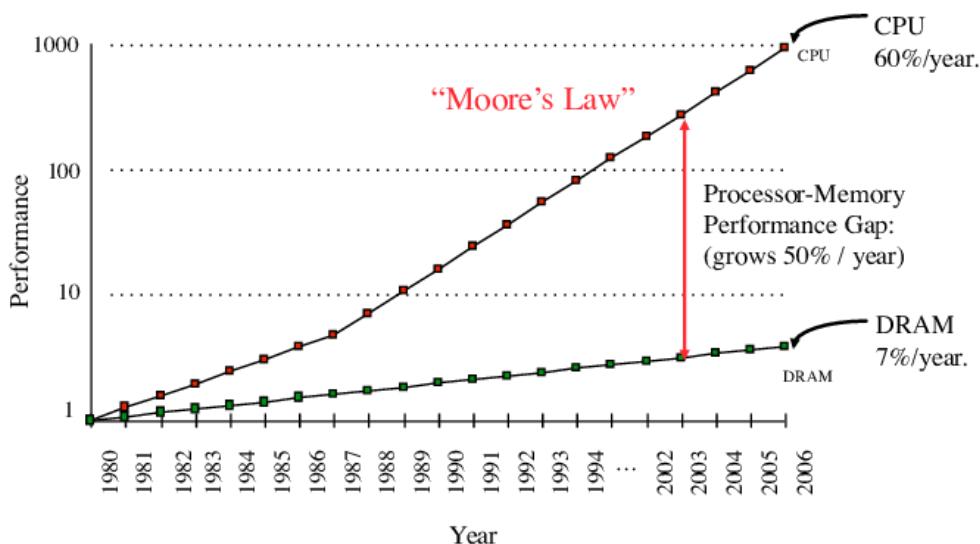


Figure 15 CPU vs RAM Evolution

Les CPU ont pendant des années profité d'un gain de puissance exponentiel mais en arrière plan de cette réalité se cache [le drame de la RAM dont la puissance ne croît que de façon linéaire](#). Le problème vient du fait que pour qu'un CPU puisse calculer, il doit avant tout sortir les données de la RAM pour la ranger dans ses registres. Et là, tout d'un coup la précédente figure révèle le drame qui se joue lors de chaque calcul de balle qui tombe sur une scène...

Autrement dit, avec les fréquences des CPU modernes le temps d'acquisition d'une donnée en RAM peut prendre [plusieurs centaines de cycle CPU](#). Pour compenser ce problème, les CPU disposent d'une mémoire intégrée dont l'accès est jusqu'à [deux ordres de magnitude](#) plus rapide que pour de la RAM. Cette mémoire est le *cache* du CPU et il en existe 3 différents sur les puces modernes: L3, L2 et L1 qui possède chacun un rapport taille/vitesse d'accès différent, L3 étant le plus lent mais le plus volumineux.

Ce n'est pas que le cache est plus rapide pour accéder à la mémoire, mais il applique une stratégie qui consiste à *cachet* toute la zone contiguë à un byte fetch en RAM par le processeur en pariant sur le fait que le prochain byte qui sera réclamé fera partie de cette zone. Si c'est le cas, le CPU pioche dans le cache, ce qui lui évite de devoir faire un accès RAM.

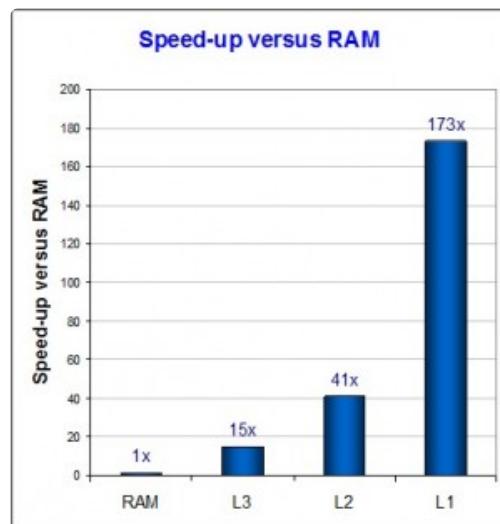


Figure 16 Cache vs RAM

Cette zone est appelée *cache line* et sa taille est de l'ordre de la centaine de bytes. Le fait d'éviter un accès RAM car le cache contient le prochain byte s'appelle *cache hit* et si ce n'est pas le cas, on parle de *cache miss*.

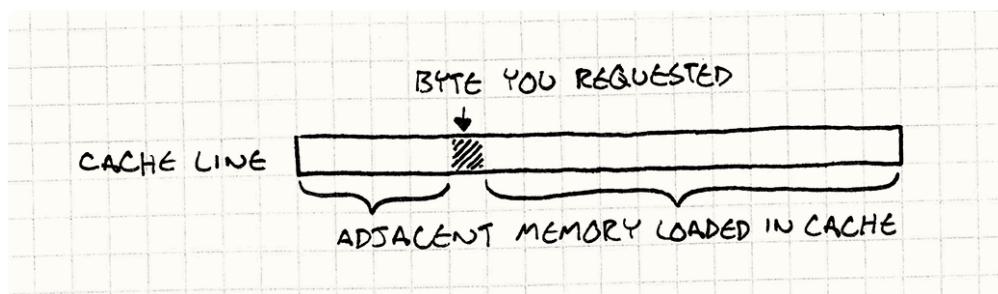


Figure 17 <https://gameprogrammingpatterns.com> Cache line

Lors d'un *miss*, et vu la figure 15, les performances du CPU s'effondrent car il se retrouve bloqué en attendant le prochain byte de donnée. L'impacte d'un *cache miss* est très facile à simuler programmatiquement:

```
1 | for i in myCollection:  
2 |     sleepFor500Cycles()  
3 |     i.doStuff()
```

Python

Il est facile de comprendre que dans ce cas l'impacte de la fonction `sleepFor500Cycles()` est dramatique sur le temps d'exécution. Pourtant c'est exactement le même impacte qu'a un *cache miss* sur les performances d'un programme. La conclusion logique est donc qu'il est critique d'optimiser un programme pour faire en sorte de minimiser les *cache miss*. C'est très précisément à ce moment que le pattern Component rentre en jeu.

Le pattern Component est un premier pas vers une façon plus efficace d'architecturer un programme. On parle de "[Data-Oriented Design](#)" (DOD) et cette façon d'architecturer a au centre des ses préoccupations le fait d'accélérer les accès mémoire en optimisant l'utilisation du cache et donc en assurant que **les données sont stockées en mémoire dans l'ordre de leur accès**.

Le sujet est vaste et passionnant mais sort du cadre de ce rapport pour plus d'informations sur ce sujet, le chapitre "Data locality" du livre de [Bob Nystrom](#) est une excellente introduction et pour avoir une vue complète de l'état du DOD.

### 5.1.3.3. Component, Strategy et Facade

Au niveau implémentation, ce pattern est très proche du classique *Strategy*. Ces deux patterns ont en commun le fait de déléguer une partie du comportement d'un objet à un autre objet.

Dans un pattern *Strategy*, la partie déléguée, la stratégie, sera généralement sans état et encapsule uniquement un algorithme. La Stratégie est instanciée par l'objet qui délègue et il est en charge de le remplacer par un autre objet qui implémente un autre algorithme en fonction de l'état de l'application.

Dans un pattern *Component*, le composant est également une objet à qui on délègue une partie de comportement. À la différence de la stratégie, le component en plus de se charger de la logique, contient également l'état associée à ce comportement. Cette particularité le rends moins générique et crée un certain couplage. Ce qui n'est pas forcément problématique car lorsqu'il est possible de s'en passer un component est simplement une stratégie qu'il est possible d'utiliser dans plusieurs conteneurs. Ces deux patterns sont donc complémentaires et la limite est souvent floue entre les deux.

À titre plus personnel, n'ayant pas trouvé d'article ou d'étude pour corroborer mon impression, je trouve qu'il possède aussi beaucoup du pattern Facade. De la même façon que Facade, Component sert de point d'entrée à un sous système complexe dont il abstrait l'utilisation.

#### 5.1.3.4. Unity gameobject

Au coeur des [GameObject](#) d'Unity se trouve une implémentation du pattern Component. Un clique sur un objet de la scène dévoile dans l'inspecteur sa composition. Chacun des éléments affiché alors dans l'inspecteur est un component qui représente un comportement et l'état qui va avec.

Dans le cadre de ce projet il semble donc assez naturel de profiter de l'implémentation existante. Mais avant de se précipiter, il reste encore une voie à explorer, celle de la performance sans compromis : ECS.

### 5.1.4. Approche ECS

Une confusion persiste sur le fait qu'Unity utilise une approche ECS. [Ce n'est pas le cas](#).

ECS se démarque des précédentes approche par le fait que ce n'est pas un pattern OOP. En effet, il n'utilise aucun mécanisme de ce paradigme. Son concept est basé sur les qualités d'optimisation de la mémoire qu'apporte la composition vis-à-vis de l'héritage mais poussées au maximum. Cette façon de faire dégage des performances exceptionnelles pour l'affichage de très nombreuses entités. Cette performance se paye par l'obligation d'adopter sans compromis une approche Data Oriented.

En ECS, les `Entity` sont les objets du jeu et sont **définis implicitement** par une collection de `Components`. Ces Components ne contiennent que des données et sont opérés en groupes fonctionnels par des `Systems`.

#### 5.1.4.1. Component

Un component est un simple conteneur. Une classe qui implémente un component a des attributs mais pas de méthode. Chaque component décrit un certain aspect d'une entité ainsi que ses paramètres. En soit, un component, n'est pas grand chose et c'est leur cumul qui est intéressant. 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

En 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 le 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 entité 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 il est possible d'images 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 coeur 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 Postion 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 vélocité 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. Unity DOTS

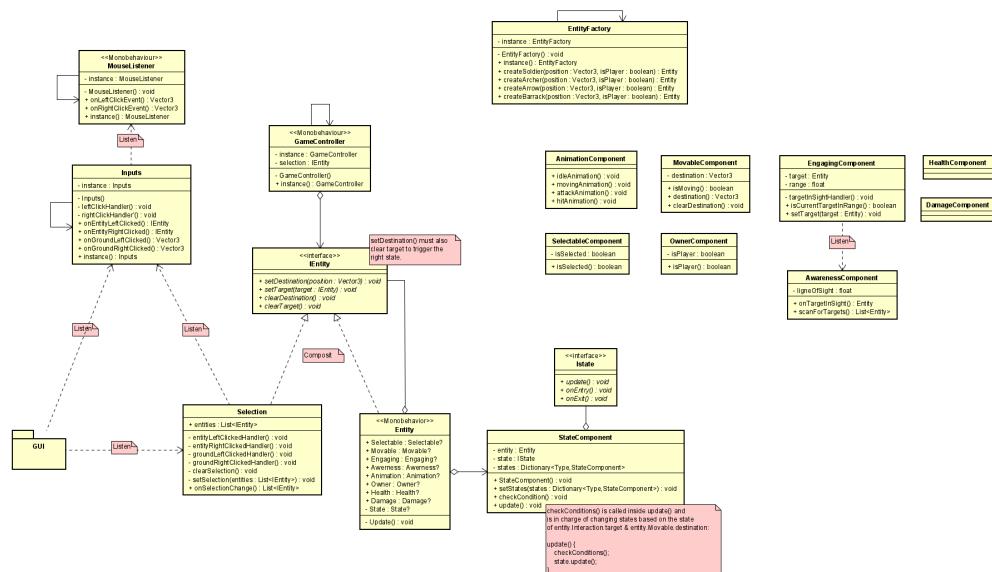
Unity DOTS est l'implémentation par unity d'une suite d'outils basé sur une approche DOD. Pour le moment ce package est hautement volatile et uniquement accessible en preview. Dans ce package ce trouve "Entities" qui est le framework ECS que propose unity.

### 5.1.5. Modèle développé

Le choix a été intialement fait d'utiliser une version programmatique de du pattern "Components". Après plusieurs discussion avec le Pr. Le Callennec, il a été décidé d'adapter l'architecture pour la faire utiliser les Gameobjects.

#### 5.1.5.1. Première itération: Approche programmatique

Le diagramme suivant présente l'architecture au moment de commencer à écrire du code:



**Figure 18** Première itération. Disponible en annexe

Les classes ne sont pas complètes et ne présentent que les membres importants pour que les membres du groupe aient une bonne idée de l'architecture générale et puisse travailler chacun de leur côté. À ce stade, il manque encore plusieurs composants à l'application mais ces composants utilisant ceux existant, ne nécessiteront pas de remaniement de l'architecture pour être intégrés. Une approche itérative a donc été adoptée pour permettre de commencer à présenter du contenu lors des réunions hebdomadaires avec le Pr. Le Callennec.

Cette première itération se base sur l'usage de factories et d'une classe `Scenario` (absente du diagramme précédent) pour instancier les éléments du jeu. Cette approche réduit la dépendance à l'interface d'Unity et ce qui rend le travail plus "traditionnel" pour des développeurs.

Il reste actuellement encore un vestige de cette approche dans les sources du projet, dans le dossier `Scripts\Game\Entities\OLD\` qui contient deux fichiers.

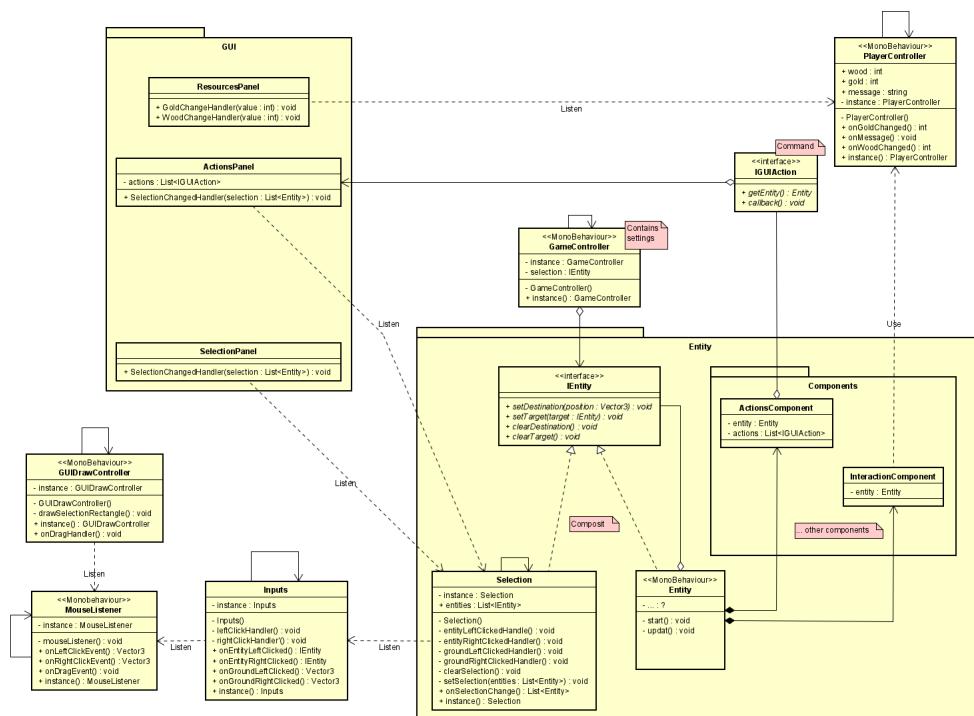
À ce stade, les composants de la classe `Entity` n'héritant pas de `MonoBehaviour`, il était de la responsabilité de la classe `EntityFactory` de composer les différents membres de la classe `entity` pour retourner ce qu'il lui était commandé par la classe `Scenario` qui elle était en charge de placer les entités à différents endroits de la carte.

#### 5.1.5.2. Seconde itération: Approche prefabs

À ce stade la grande majorité des mécanismes nécessaires sont mis en place. Après une discussion avec le pr. Le Callennec il a été décidé de tenter de remodeler légèrement la classe `Entity` pour lui permettre d'utiliser les avantages qu'apporte les prefabs d'Unity. Le principal avantage serait de pouvoir se passer de la classe `Scenario` et composer directement les scènes à l'aide de l'éditeur d'Unity.

Le premier changement fût le fait de faire hériter tous les components de MonoBehaviour afin de les rendre accessible depuis l'inspecteur d'Unity. Cette modification a eu comme conséquences de pouvoir se passer d'une grande partie des factories mais aussi de certaines stratégies. En effet, ces dernières pouvait maintenant se faire remplacer par des membres publiques du component qu'il serait nécessaire d'initialiser depuis l'interface d'Unity.

À ce stade l'architecture a pris sa forme finale qui est illustrée par le diagramme qui suit:



**Figure 19** Seconde itération: Disponible en annexe

Il n'est pas entièrement complèt et certaines choses diffèrent du code actuel mais tous les mécanismes importants y figurent.

## 5.2. Mécanismes

De nombreux mécanismes ont été mis en place au sein de ce projet et il est tout simplement impossible de tout détailler et documenter dans le temps imparti. Pour compenser, **en annexe de ce rapport se trouve une vidéo de présentation de l'application.**

### 5.2.1. Anatomie d'une Entity

Une entité est représentée par la classe `Entity` qui est le point d'entrée de sa logique. Cette classe héritant de `Monobehaviour`, un overload de la méthode `Update` permet d'injecter la logique dans le moteur d'Unity et les différents components qui sont attribués à son `GameObject` définissent son comportement:

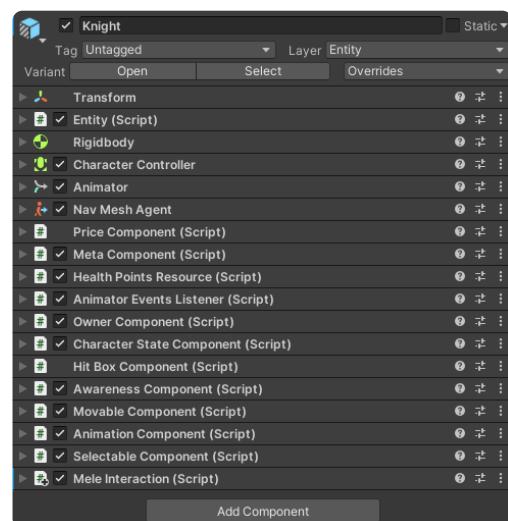


Figure 20 Unity inspector: Knight

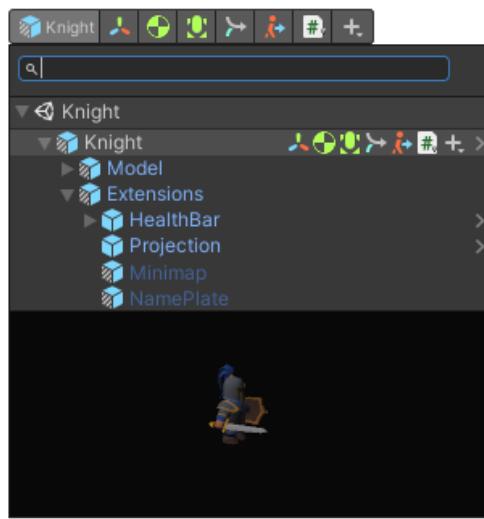


Figure 21 CAPTION

Une entité n'est pas un monolithe et est en réalité un conteneur. La figure précédente présente le `GameObject` à la racine de ce conteneur et la figure suivante décrit la hiérarchie d'une entité:

- Le GO **HealthBar** contient la présentation des components qui spécialisent le component `BaseResourceComponent` (Dans le cas du Knight `HealthPointsResource`).
- Le GO **Projection** contient le marqueur de selection utilisé par le component `SelectableComponent`.

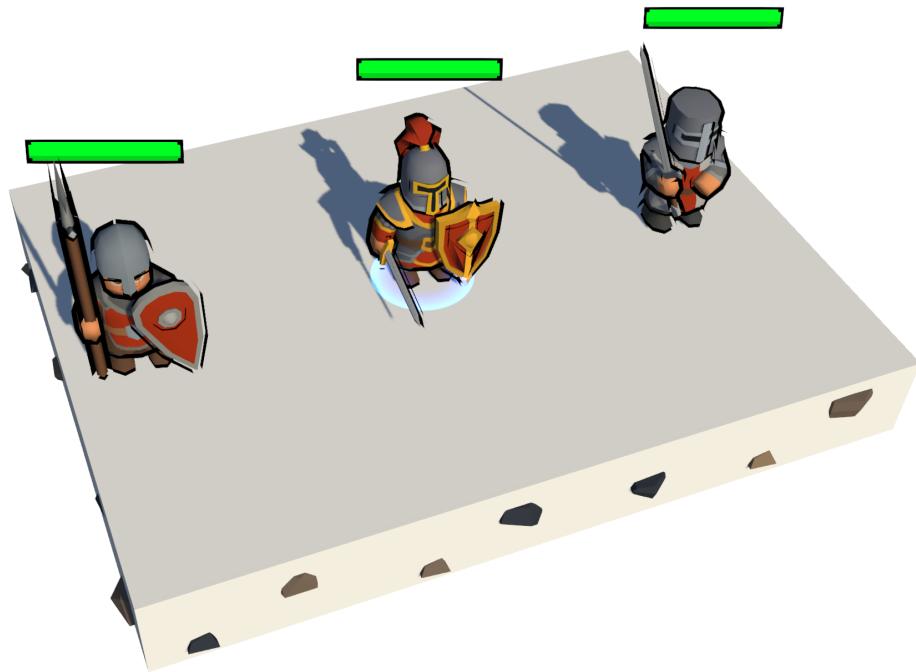


Figure 22 CAPTION

Le dossier `Resources/Prefabs/Entities`. Contient toutes les entités du jeu. À la racine de ce dossier se trouve la prefab la moins spécialisée qui est à l'origine de toutes les autres spécialisations. Cette façon de faire permet de ranger dans cette prefab toutes les extensions qui sont communes à toutes les entités.

## 5.2.2. Brouillard de guerre



Figure 23 CAPTION

Pour le projet de M. Seuret, ce dernier se concentrat majoritairement sur le *Fog of War* (brouillard de guerre). En effet, bien que paraissant très simple, il est complexe de créer un brouillard de guerre fluide ne consommant pas trop de ressources.

Ce point utilise des notions du chapitre 8 du livre "WebGL par la pratique", concernant les textures. En effet, que ce soit les ajouts des meshs à une texture où l'application de cette texture projeté sur l'entièreté du terrain, ce concept est dépendant de la possibilité de texturer l'objet.

Cette partie du projet est concentrée sur le rendu graphique. En effet, le FoW n'a aucun autre intérêt que de cacher de l'information au joueur. Il existe quand même un besoin pour la modélisation géométrique qui nous permet de créer les meshs simulant le champ de vision des unités. Cependant, cela reste très simple étant donné qu'il ne s'agit que de plans.

### 5.2.2.1. Solution proposée

Suite aux recherches effectuées ainsi que les différentes méthodes d'implémenter un FoW dans Unity, il a été décidé d'implémenter une méthode d'addition d'images.

Il existe cependant d'autres possibilités dont voici les désavantages :

- Texture transparente surplombant l'entièreté de la carte
  - Celle-ci doit prendre en compte l'angle de la caméra afin de supprimer les parties de la texture qui nous permettront de voir sur le terrain.
- Voxels supprimés à l'approche des unités
  - Extrêmement coûteux à initialiser
  - Demande le rajout de voxel lorsque ces derniers ne sont plus dans le champ de vision des unités afin de simuler les parties découvertes.
- Brouillard de Unity
  - Effet global
  - Effet coûteux pour le GPU

Ainsi, étant donné que tous les calculs sont fait sur des images, peu importe le nombre d'unités ou structures présentes, le processus ne prendra pas de temps supplémentaire (très peu, dû à la génération des meshs de chaque unité). Ceci était un point crucial car le projet a pour but de simuler aisément des batailles de plusieurs centaines d'unités.

### 5.2.2.2. Modèle développé

#### 5.2.2.2.1. Fog of War standard

Prenons tout d'abord le FoW basique, c'est-à-dire ce que les unités voient actuellement. Premièrement, il est nécessaire pour les unités de générer un mesh de leur simulant leur champ de vision théorique. Ainsi, ceci permettra de déterminer ce qu'elles peuvent voir.

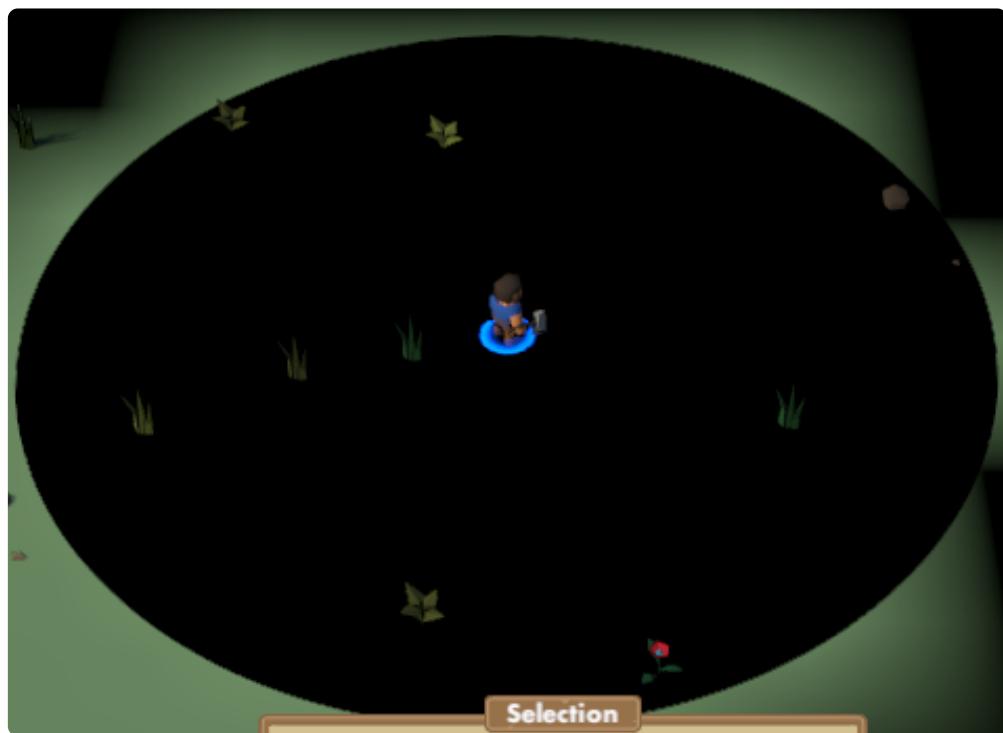


Figure 24 mesh des unités

Ces meshs sont ensuite capturés par une caméra - capturant son propre layer Unity - qui générera une *Render Texture*. Cette dernière sera ensuite projeté sur le terrain grâce à un objet de projection. Ce dernier ne pouvait malheureusement pas être utilisé dû au choix, en amont, d'utiliser le *Universal Render Pipeline* de Unity. Pour résoudre ce problème, nous avons utilisé le shader développé par Colin Leung: [UnityURPUnlitScreenSpaceDecalShader](#).

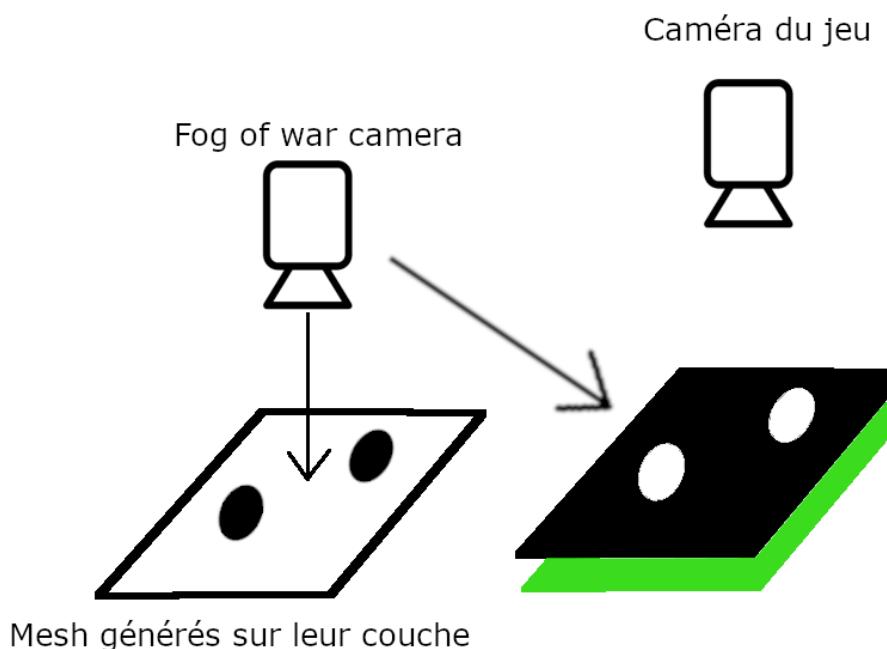


Figure 25 Explication FOW

Ceci nous permet finalement de rendre totalement noir l'entièreté du décor qui n'est pas directement vu par une des unités.

#### 5.2.2.2.2. Fog of War découvert

Concernant le FoW semi-transparent, c'est-à-dire les lieux découverts mais sur lesquels nous n'avons plus la vision, cela fut plus complexe.

En effet, le principe utilisé précédemment ne permet pas de garder une trace des positions déjà découvertes. Il est néanmoins possible d'adapter la solution.

Une des solutions est de ne pas *clear* (mettre à zéro) la texture dans laquelle la caméra va sauvegarder ses informations. Cependant, vu que nous utilisons le *Universal Render Pipeline*, cette solution n'est pas disponible. Il existe cependant la possibilité d'empiler (stack) le rendu de caméra et, pour les caméras empilées sur celle de base, de sauvegarder **par-dessus** les informations de la caméra de base. Il n'est malheureusement pas possible d'avoir le même comportement depuis une texture.

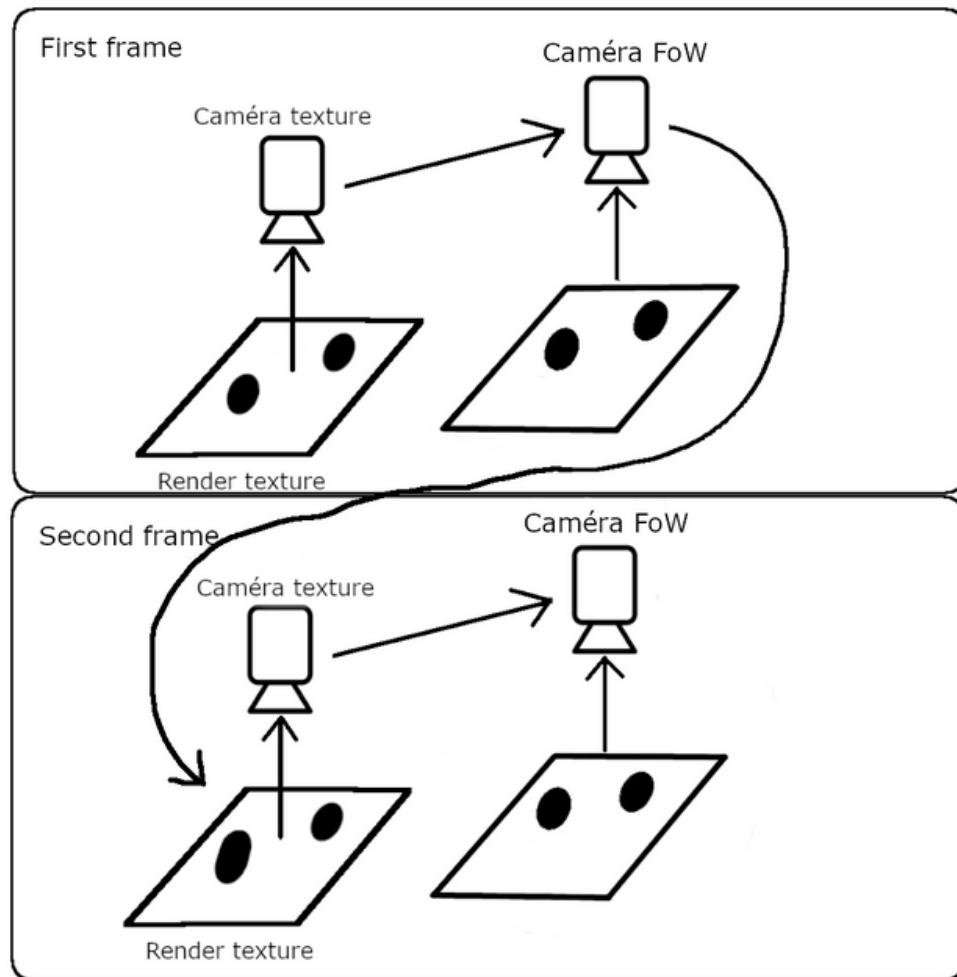


Figure 26 Explication FoW découvert

Ainsi, pour contourner les limitations de Unity, une render texture a été placée dans un layer propre à elle ainsi qu'une caméra. Le seul but de cette caméra est de filmer la texture afin de la récupérer pour la caméra qui sauvegardera par-dessus. Une fois la texture récupérée par la caméra de base, la caméra suivante sur la pile pourra sauvegarder ses informations et ainsi mettre à jour la texture. Cette dernière sera ensuite réappliquer sur l'objet filmé par la caméra de base et le processus recommence.

### 5.2.2.3. Integration

Comme expliqué précédemment, il est nécessaire d'utiliser un ensemble de 3 caméras :

- Caméra pour le FoW standard
- Caméra qui lit la texture
- Caméra qui applique le FoW par dessus la texture

Ensuite, il est nécessaire d'ajouter des objets représentant le FoW en lui même. Il s'agit de cube très fin sur lesquels le shader récupéré agira.

#### 5.2.2.4. Avantages et inconvénients

Concernant les avantages et inconvénients de cette implémentation, elle impose un coût presque fixe en performance, ce qui peut être un avantage, et un inconvénient dépendant du contexte. Un autre avantage est qu'il prend avantage de ce qui est offert par Unity et demande extrêmement peu de code pour notre rendu. Ici, il n'existe qu'un script permettant de générer les meshs des unités ou des bâtiments. Ainsi, elle est très simple à implémenter.

#### 5.2.2.5. Résultats et performances

Prenons une scène relativement simple, en faisant combattre une vingtaine d'unité de chaque camp :



Figure 27 fps combat

Comme nous pouvons le remarquer, nous avons une moyenne de ~21 fps pour cette scène de combat (chiffre vert).

Maintenant, en appliquant le FoW :

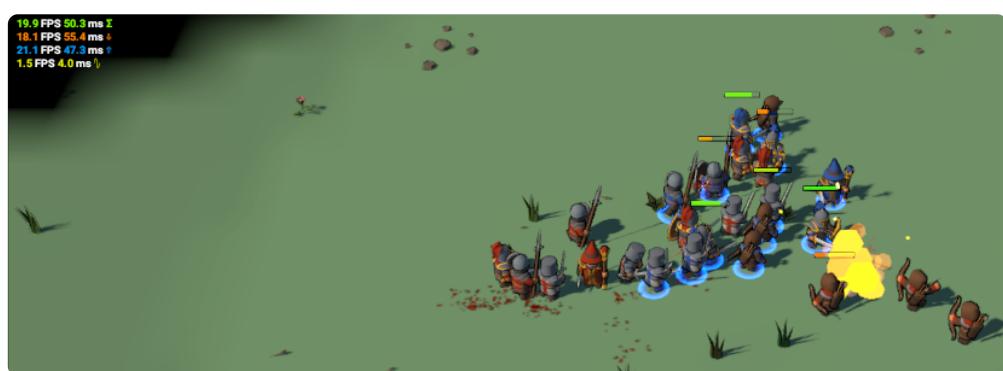


Figure 28 fps combat FoW

Cette fois, nous avons une moyenne de ~20 fps, soit une baisse de 5%. Cependant, il est important de noter que le nombre d'image par seconde est extrêmement volatile et dépend de beaucoup d'autres facteurs. De ce fait, le FoW a un impact quasi nul sur les performances.

Notre implémentation est donc plus que satisfaisante, car il n'impose qu'une perte très minime sur les performances du jeu, et ce avec un facteur de complexité très faible. Ceci nous permet de garder des performances raisonnable même avec un grand nombre d'unités, ou en tout cas que le FoW ne soit pas le point faible de ce projet.

### 5.2.2.6. Améliorations

Dans l'ensemble, l'implémentation actuelle remplit son travail et permet de ne pas imposer un coût trop élevé en performance. Cependant, afin d'être parfaitement fonctionnel, il nécessite encore quelques modifications afin d'ajouter la gestion logique de ce FoW. En effet, il serait nécessaire de n'afficher que les bâtiments découverts dans la zone de FoW découvert (semi-transparent). Les unités ne devraient plus être visibles ainsi que toutes modifications futures de l'adversaire.

### 5.2.2.7. Références

Malheureusement, il existe très peu de documents parlant de l'implémentation des Fog of War dans les jeux vidéos. De plus, ces derniers sont en général relativement vagues sur les techniques utilisés ou se reposent sur les possibilités offertes par les moteurs de jeu (Unity, Unreal Engine).

Il est néanmoins possible de trouver des explications de FoW plus poussé mais ces derniers se concentrent principalement sur la partie logique. Voici un exemple avec le "FoW" de [Valorant](#), nouveau jeu de Riot Games encore en bêta.

Ce système est déjà utilisé dans le jeu Counter-Strike : Global Offensive sorti en 2012 et existait déjà aussi tôt que 2005 dans des mods de son itération précédente, Counter-Strike : Source.

Cependant, ceci ne nous intéresse pas étant donné qu'il ne s'agit que de la partie logique et non pas la partie graphique d'un tel rendu.

## 6. Interface Utilisateur

Pour le projet d'infographie du jeu YAARTS, la mise en place d'une interface utilisateur est inévitable. Plusieurs techniques existent pour pouvoir créer une UI dans Unity. Le but de cette partie est de décrire ces différentes méthodes avec leurs avantages et inconvénients.

Une interface utilisateur est souvent sous-estimée dans un jeu. Pourtant, c'est un élément majeur. Elle aide à la compréhension et aux actions que le joueur peut exécuter. Dans un jeu de stratégie en temps réel, l'interface peut devenir lourde en informations. L'aperçu des informations doit être rapidement vérifiable et limpide pour le joueur.

Le domaine de l'UX/UI est vaste. Il est impossible de résumer l'entièreté de cette thématique. Néanmoins, la présentation de certains points importants liés aux jeux vidéos est possible.

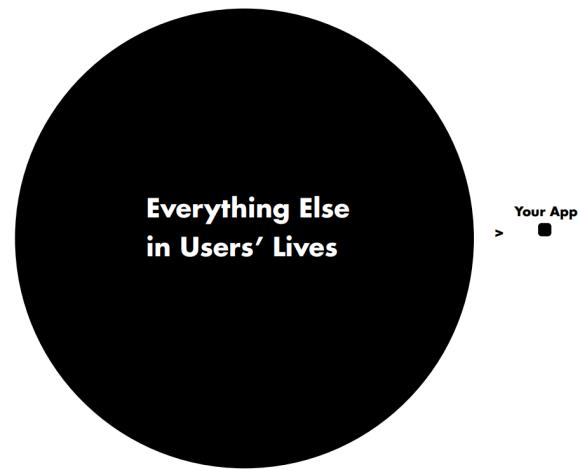
### 6.1. Expérience et interface utilisateur

Une première recherche se porte sur l'expérience utilisateur (UX) dans les jeux vidéos. Dans les studios de développement, les développeurs sont aidés par des designers. Cela permet d'avoir une cohésion dans le jeu. Dans ce travail, les connaissances sont basées uniquement sur les projets faits précédemment par des développeurs.

À la conception d'une UI, il est important de suivre des règles pour que l'expérience utilisateur ne soit pas détériorée.

Voici une liste non exhaustive de règles :

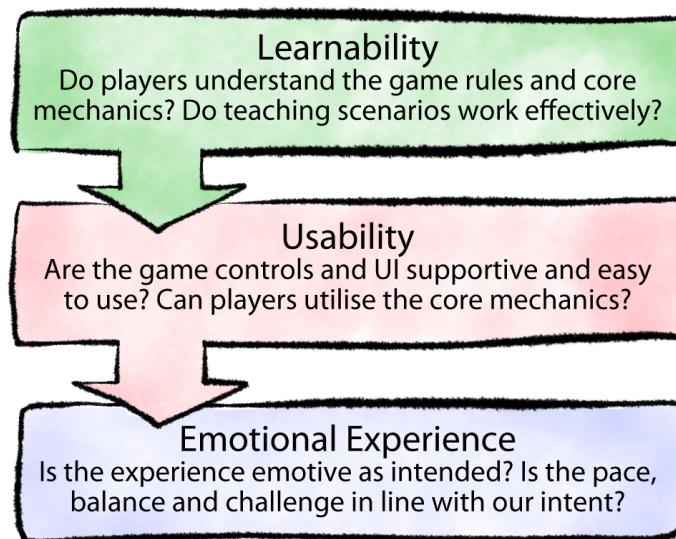
- S'adapter aux habitudes des utilisateurs, pas l'inverse
- Quand le team de développement devient trop proche de leur projet, elle perd en impartialité. Elle devient experte du jeu et peut oublier le ressenti d'une personne qu'il le découvre.
- Keep It Simple and Stupid (**KISS**)  
Cette règle est universelle, autant dans le code que dans l'interface. Plus l'UI est simple est, plus il est simple de la comprendre.



**Figure 29** UX Fundamentals for Non-UX Professionals, les habitudes par rapport à l'application

- Connaître l'audience visée Avant de créer une interface, la connaissance de l'audience est un facteur clé. Dans le cas d'un jeu vidéo, on l'adapte à l'âge, région de l'endroit de la diffusion du jeu. Beaucoup de jeux ont été modifiés pour correspondre aux moeurs de la région (ex. la différence des morts-vivants de World of Warcraft en Asie et en Amérique).

Pour avoir un bon feedback des utilisateurs, il est important de le séparer en plusieurs groupes. Voici un schéma qui représente ces différents groupes et leur priorité :



**Figure 30** Medium, what is game UX, conception steps

L'apprentissage du jeu est la première étape. Si les joueurs ne comprennent le jeu, ils ne resteront pas longtemps dessus. L'utilisabilité doit fournir à l'utilisateur un moyen simple de jouer au jeu (peu importe la difficulté de celui-ci). Le joueur a compris quels sont les buts à atteindre (apprentissage), mais si les contrôles ne sont pas conventionnels ou clairs, cela crée une frustration. La dernière étape est l'expérience émotionnelle de l'utilisateur. L'équilibre du jeu est respecté (Game design).

Pour conclure cette partie, l'expérience utilisateur est un sujet très vaste et il est facile de s'y perdre. Pourtant, c'est un point crucial pour trouver un public au jeu. Cela passe par l'interface.

## 6.2. Solutions proposées

Il existe plusieurs technologies pour faire une interface en Unity. Dans ce projet, NGUI a été essayé sur le menu principal. L'interface en jeu est en UGUI.

### 6.2.1. Unity UI (UGUI)

Unity UI (UGUI) est actuellement le toolkit par défaut d'Unity pour développer une interface. Il utilise un canevas pour faciliter la création. Malheureusement, l'amélioration de UGUI n'était pas une priorité des développeurs. À terme, il va être remplacé par UIElements. Le système est un goulot d'étranglement de performance. Moins qu'au départ avec de différentes améliorations, mais il reste quand même une perte de performance à ce niveau.

L'architecture à utiliser est tout aussi problématique. Il existe plusieurs moyens de *bind* des événements, mais il est difficile de séparer correctement l'UI avec les données.

### 6.2.2. Next-Gen User Interface (NGUI)

Le Next-Gen User Interface est un plug-in pour Unity. Il permet de facilement créer une interface grâce à l'utilisation de widgets et la gestion dans l'éditeur d'Unity. Ce plug-in a été développé pour mieux gérer la création d'une interface dans Unity. Avant la version 4.6 d'Unity en janvier 2015, le système d'interface n'était pas optimisé et rendait sa création complexe et fastidieuse. NGUI a permis de faciliter l'élaboration d'une UI grâce notamment aux widgets et à l'utilisation de l'éditeur. Il était aussi plus performant au niveau de la réactivité.

Le plug-in était sous licence, mais il est possible de créer une bibliothèque de widgets si l'on possède cette licence. Beaucoup d'autres plug-ins continuent d'exister avec comme base NGUI. Aujourd'hui, avec l'évolution du UGUI, NGUI a moins d'intérêt. Il permet toujours de faire des interfaces rapidement, mais n'offre plus l'optimisation dont il jouissait à l'époque.

D'ailleurs, les développeurs du plug-in ont été recrutés chez Unity pour le développement de la gestion de l'interface.

### 6.2.3. User Interface Elements (UIElements)

UIElements est le nouveau système de GUI d'Unity. Ce système est voué à remplacer Unity UI. Il permet de créer des interfaces avec les technologies web. La création passe par des fichiers de style (*stylesheets*), une gestion des événements revus et une persistance des données. UIElements utilise format UXML pour fournir une structure à l'interface. La séparation entre le *frontend* et *backend* très utilisé en web permet une meilleure répartition entre le design et le code métier. Il permet aux développeurs et designers de mieux collaborer.

Ce système offre beaucoup de nouvelles fonctionnalités par rapport à l'UGUI :

- **The Visual Tree:** contiens tous les éléments visuels dans une fenêtre. L'arbre visuel est un objet graphique fait de nœuds légers appelés *visual elements*
- **The Layout Engine :** positionne les éléments visuels en fonction de la propriété du layout et du style.
- **The UXML format:** définit la structure de l'interface utilisateur
- **Styles and Unity style sheets (USS):** définis les propriétés de style qui définissent les dimensions et l'apparence des éléments visuels.
- **The Event system :** communique les interactions des utilisateurs aux éléments visuels.
- **Built-in controls:** Contextualisation des menus de contrôle
- **Binding:** relie une propriété au contrôle visuel qui modifie la valeur de la propriété.
- **Supporting IMGUI:** support pour le debug d'application
- **ViewData persistence:** persistance des données d'état spécifiques à l'interface utilisateur. UIElements est encore très jeune. Il faut attendre encore un moment pour qu'il gagne en maturité et fournit une base solide pour la création d'une interface.

### 6.2.4. Immediate Mode Graphical User Interface (IMGUI)

Ce système est surtout utilisé pour le debug d'une application. Elle n'est pas vouée à être fournie aux utilisateurs. IMGUI est utilisé dans l'affichage d'outils de debug, la création d'inspecteurs pour des scripts ou pour la création de nouveaux fenêtres et outils pour l'environnement d'Unity.

## 6.2.5. lien avec le/les Ch. d'infographie

Cette partie se concentre sur les interactions (chapitre 5). Il est simple de justifier le rapprochement, l'interface utilisateur offre des informations sur l'état du jeu et les actions du joueur permettent d'interagir avec l'environnement de celui-ci. Que ce soit dans le jeu ou par le joueur, les événements sont asynchrones et demandent d'être traités à tout moment. L'interaction passe par plus biais. Le clavier, la souris est un moyen de communication avec le jeu, mais aussi les composants comme les boutons, sliders et plein d'autres.

## 6.2.6. Modèle développé

Le modèle développé se rapproche du jeu StarCraft 2.



Figure 31 StarCraft 2, jeu développé par Blizzard

Le positionnement et la gestion des événements est très proche de cet autre RTS. Cela permet de ne pas modifier les habitudes des joueurs.

### 6.2.6.1. Gestion des actions

Pour pouvoir gérer des actions avec l'interface, une séparation est faite sur plusieurs éléments. Voici un diagramme simplifié de la gestion d'événements :

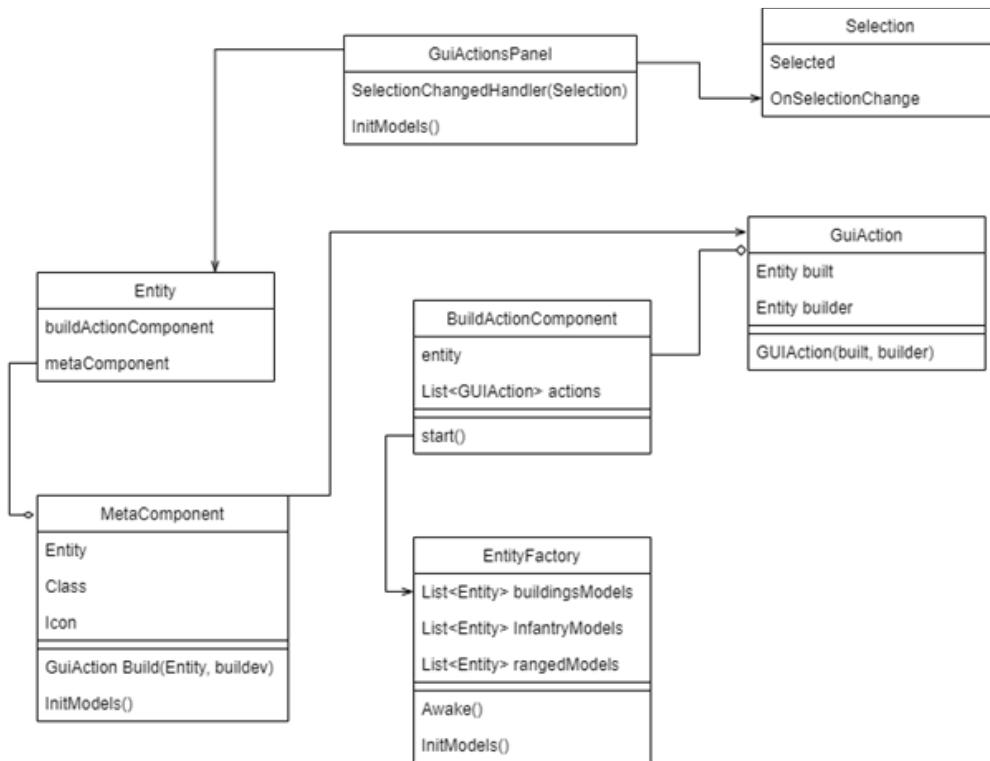


Figure 32 diagramme de gestion des actions

Si la sélection d'unité change, le GuiActionsPanel est notifié. S'il y a un bâtiment ou un travailleur, la liste des actions valides remplit le panel d'action. A la création d'une entité on lui donne une liste d'actions qu'il peut exécuter. Ce système permet de modifier librement une action et de l'ajouter à n'importe quelle entité

#### 6.2.6.2. Minimap

Pour la minimap, une camera est positionnée à une hauteur suffisante pour avoir l'effet orthographique. Dans le projet actuel à cause de la pipeline, il n'est pas possible d'utiliser la caméra orthographique fournie par Unity. La position de toutes les entités visibles est récupérée et rendue sur l'image les points de couleurs. Le Fog of War est aussi affiché.

#### 6.2.7. Relation avec les 3 piliers de l'infographie

Une interface graphique peut avoir une relation directe avec les trois piliers de l'infographie. La modélisation d'objets est possible pour les présenter aux joueurs dans l'interface. Une UI à sa propre "scène" elle est souvent isolée au reste du jeu et à donc son propre rendu. L'animation permet de rendre l'interface plus ludique et fournir des informations sur l'état du jeu et avoir un retour des actions de l'utilisateur.

## 6.2.8. Architecture Unity

### 6.2.8.1. Présentation

Les composants utilisés dans le GUI sont sur le canevas. Nous utilisons le système UGUI.

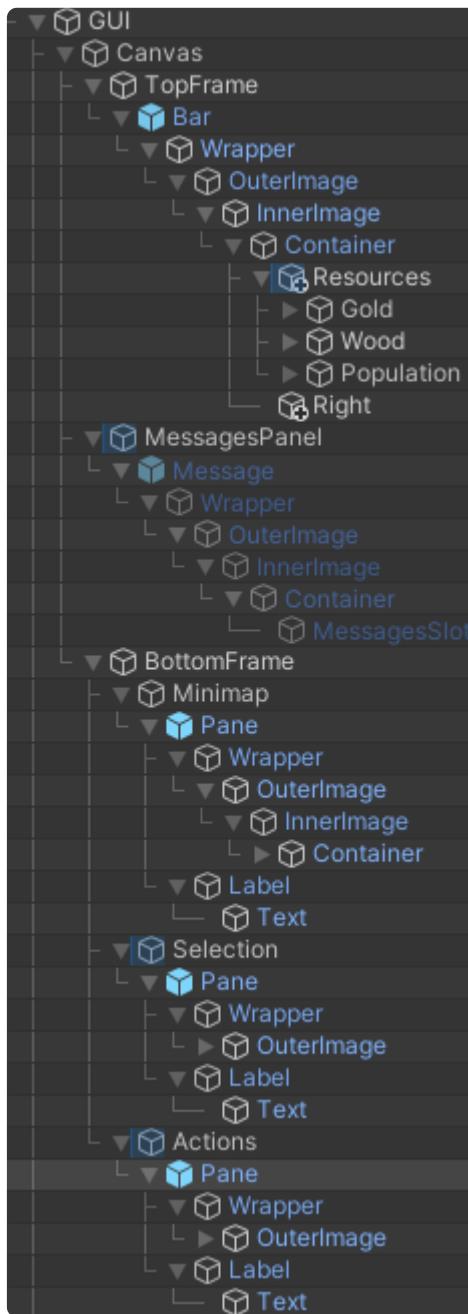


Figure 33 Hierarchie GUI

Chaque élément est dans un **gameobject** qui donne la position de l'élément pour avoir une meilleure visibilité et permet de modifier toute une partie du la GUI. Le changement de résolution est géré pour que l'UI s'adapte correctement.

#### 6.2.8.2. Avantages et inconvénients

L'architecture de l'application et le pattern observer permet de rapidement mettre en place l'affichage des données et de séparer le rendu des données. Il manque des fonctionnalités de personnalisation dans l'éditeur.

### 6.3. Résultats

Pour les résultats voici l'affichage général obtenu en jeu :

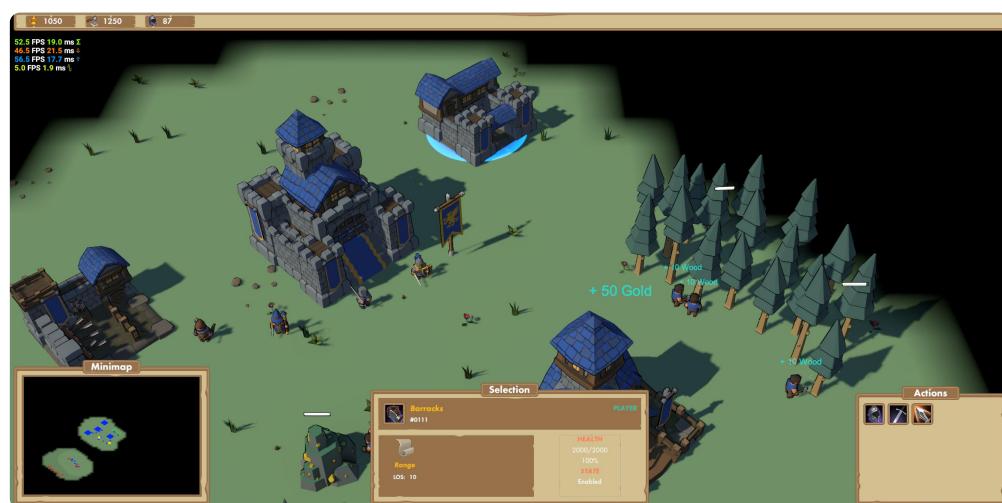


Figure 34 Jeu YAARTS avec l'interface

Le menu de sélection qui change par rapport à la sélection. S'il y a qu'une unité, on peut voir ses attributs.



Figure 35 Affiche tous les éléments sélectionnés



Figure 36 Sélection d'un bâtiment



Figure 37 Sélection d'une unité

La minimap qui permet de visualiser rapidement l'état de la carte avec les unités ennemis en rouge et les unités alliées en bleu. Les carrés jaunes sont les minerais d'or et en vert les arbres.



Figure 38 Minimap avec la position des éléments

Une barre en haut affiche les ressources. Les trois ressources du jeu sont l'or, le bois et le nombre de populations qui appartient au joueur. La population dans un jeu de stratégie permet de limiter le nombre d'entités dans l'armée d'un joueur.



Figure 39 Barre des ressources

Le cercle bleu sous les entités (une ou plusieurs) donne un visuel sur la sélection en plus du panel au milieu en bas de sélection. Les barres de vies des unités ou ressources affichent respectivement leurs points de vie ou le nombre de ressources restantes. Quand une construction est en cours la barre de vie augmente progressivement par rapport à l'avancement du bâtiment.



Figure 40 Sélection des unités entourées en dessous

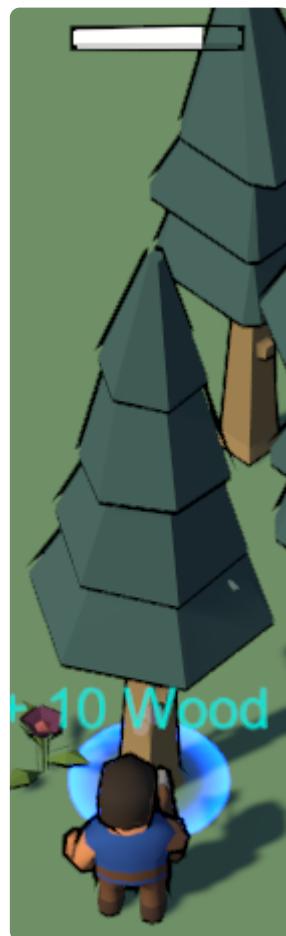


Figure 41 Ressources restantes



Figure 42 Point de vie d'une unité



**Figure 43** Point de vie pendant la construction

Un menu d'actions permet au joueur d'interagir pour la création d'un bâtiment si un travailleur est sélectionné et le recrutement d'une unité si c'est un bâtiment qui est choisi.



Figure 44 Menu de création de bâtiment



Figure 45 Menu de création d'unité

L'UI va encore évoluer. Il faut encore plusieurs itérations avant d'arriver à une interface ergonomique et 100% fonctionnelle.

## 7. Assets

---

### 7.1. Tools

#### 7.1.1. Free

- Debug Drawing
- LiteFPSCounter
- ProBuilder
- PolyBrush

#### 7.1.2. Payed

- Peek
- Console Pro
- QHierarchy

### 7.2. Assets

#### 7.2.1. Payed

- ToonyTinyPeople
- Polygonal Arsenal

## 8. Références

### 8.1. Architecture

#### 8.1.1. Bibliographie

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Elements of reusable Object-Oriented Software", Addison-Wesley, 1994
- Alexander Shvets, "Dive into Design Patterns", Goodreads Author, 2018
- Robert Nystrom, "Game Programming Patterns", gb, 2011
- Mast Smith, "Unity 2018 Cookbook", Packt, 2018

#### 8.1.2. Webographie

- Nate Eborn, "Component design pattern", Medium, 2017
- Tony Albrecht, "Pitfalls of OOP", Sony Computer Entertainment Eu R&D division
- Rams3, "ECS Deep Dive", Rams3's Blog 2019
- Eizenhorn, "Unity ECS & Job System in production", Medium, 2018
- Herb Sutter, "Modern C++: What you need to know", Channel9.msdn, 2014
- Ray wenderlich, "Intro de component based architecture", Raywenderlich.com, 2013
- Mouad Bahi, "High Performance by Exploiting Information Locality through Reverse Computing", Researchgate, 2011
- Sandra Alvarez, "Rts group movement", github page, 2017
- Habrador, "Learn how to optimize your Unity project", Habrador website
- Andrew Sirota, "Unity optimization tips: Mobile & Desktop", 2020 Makaka Games
- Yanko Oliveira, "A UI System Architecture and Workflow for Unity", Gamasutra
- Nick Bucher, "Introducing Design Patterns and Best Practices in Unity", University of Alabama
- Kieran Lord, "How I Code in Unity or How I've Been Using Unity in Stupid Ways for 5 Years", 2014

### 8.2. Fog of war

#### 8.2.1. Webographie

Chamberlain, Paul. "Demolishing Wallhacks with VALORANT's Fog of War." Riot Games Tech, Paul Chamberlain, 14 Apr. 2020, technology.riotgames.com/news/demolishing-wallhacks-valorants-fog-war.

### 8.3. GUI

### 8.3.1. Bibliographie

- Edward Stull, « UX Fundamentals for Non-UX Professionals: User Experience Principles for Managers, Writers, Designers, and Developers », Paperback, 2018.
  - Diana MacDonald, « Practical UI Patterns for Design Systems. Fast-Track Interaction Design for a Seamless User Experience », Apress, 2019.
  - Dr. Ashley Godbold, Mastering UI Development with Unity, 2018.
  - Steve Schoger, Adam Wathan, « Refactoring UI », 2018.

### 8.3.2. Webographie

- What is Games UX : [medium.com/@player\\_research/what-is-games-user-experience-ux-and-how-does-it-help-ea35ceaa9f05](https://medium.com/@player_research/what-is-games-user-experience-ux-and-how-does-it-help-ea35ceaa9f05)
  - RTS UI : [lets-talk-rts-user-interface.waywardstrategy.com/2015/05/04/lets-talk-rts-user-interface-part-1-interview-with-dave-pottinger](http://lets-talk-rts-user-interface.waywardstrategy.com/2015/05/04/lets-talk-rts-user-interface-part-1-interview-with-dave-pottinger)
  - Building a Better RTS : [polarorbit.net/2015/03/building-a-better-rts-part-2](http://polarorbit.net/2015/03/building-a-better-rts-part-2)
  - Strategy Game Battle UI : [medium.com/@treeform/strategy-game-battle-ui-3b313ffd3769](https://medium.com/@treeform/strategy-game-battle-ui-3b313ffd3769)
  - discussion about UGUI and the future : [forum.unity.com/threads/what-is-the-future-of-ugui.573481/](https://forum.unity.com/threads/what-is-the-future-of-ugui.573481/)

## 9. Annexes

---

- YAARTS\_Demo.mkv
- Journal de bord: Sol
- Journal de bord: Nathan
- Journal de bord: Tristan