Rapport TER

Zink Tom - Saperès Clément - Nigh Kai - Bonetti Timothée

april 2025

Table des matières

1 Introduction

L'objectif de ce projet est de réaliser un jeu vidéo massivement multijoueur sur navigateur. Celui ci se joue de deux façons bien distinctes :

- *Mode "Péon"* : Le joueur contrôle un personnage et s'occupe des actions élémentaires telles que récupérer des ressources, construire des bâtiments, etc.. Ces cations seronts fait a traves des mini-jeux simples.
- *Mode "Seigneur"* : Le joueur ne contrôle pas de personnage directement, mais a une vision plus globale et donne des ordres aux péons en fonction des besoins du royaume.

Il y aurait plusieurs royaumes gérés par des seigneurs différents, en concurrence pour devenir le royaume le plus puissant. L'idée est inspirée de la "pixel war", l'événement temporaire sur r/place de Reddit où tous les utilisateurs du site pouvaient dessiner sur une page blanche. Ce principe très simple avait créé des dynamiques de groupes autour de communautés d'Internet et de streamers notamment qui voulaient leur place sur ce tableau géant, et nous avions assisté à des guerres d'alliances et des trahisons. Là où le but de la pixel war était purement esthétique, notre jeu serait plus concurrentiel. L'objectif de notre projet est de recréer ces dynamiques de groupes dans les royaumes.
// Image Pixelwar (commentée) ca prend du temps a compiler
Ce projet peut se suffire a lui même mais il est fait en collaboration avec un autre groupe de TER qui a crée un Backend scallable pour accueillir des applications comme la notre. Nous allons donc lier nos projets pour déployer notre frontend sur leur backend.
Les défis que nous avons identifiés avant le debut du developement sont les suivants :

- Générer une carte procéduralement avec ressources.
- Choisir la bonne technologie pour faire tourner le projet sur un navigateur.
- Créer un moteur de jeu simple.
- Comment créer une interface utilisateur.
- Créer les minijeux.
- Créer et équilibrer les 2 façons de jouer différents.
- Gérer la communication avec le Backend.
- Créer (ou trouver) des assets.

trouver une problématique

2 Conception et recherches

2.1 Game design

2.2 Planification

Partage des tâches Au départ de ce projet nous avions trois premiers objectifs, créer un outil de génération de carte en c++, poser les base du moteur de jeu en typescript et préciser le design du jeu que nous voulions créer. Nous nous sommes donc partagés les tâches, Tom Zink s'occuperait du moteur, Clément Saperes du game design, Kai Night et Timothée Bonetti de la génération de la carte. Une fois les bases du moteur de jeu crées nous devions mettre en commun et créer les 2 gameplay, les interfaces, l'interactivité. Cela a été plus compliqué que prévu, au vu de notre emplois du temps mais aussi de difficultés rencontrés : Mauvaise connaissances des technologies (typescript, TREE js), l'absence de librairie satisfaisante pour créer des interface utilisateur sur navigateur et autres soucis.
Nous explorerons plus en détails les difficultés rencontrés dans d'autres sections de ce compte rendu.

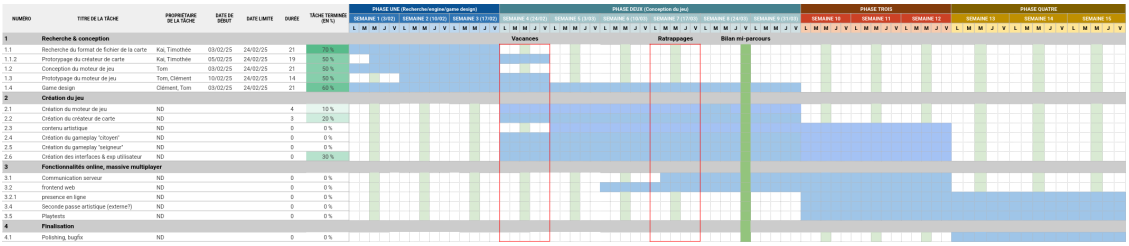


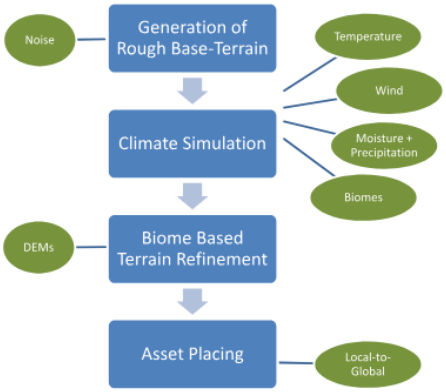
FIGURE 1 – Diagramme de Gantt

Ce diagramme montre notre organisation initiale pour ce projet.

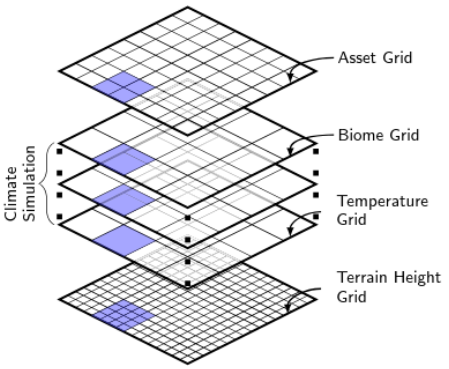
2.3 Recherche documentaire

2.3.1 AutoBiomes : procedural generation of multi-biome landscapes (2020)

Approche proposée pour la génération procédurale de terrains Les auteurs présentent un système de génération procédurale de terrain combinant trois types d'approches : synthétique, physique et exemple-based (basée sur des exemples). L'objectif est de créer des paysages étendus, composés de différents biomes et peuplés de nombreux éléments.
Le système repose sur un pipeline, structuré en quatre étapes principales. Chaque étape peut être visualisée directement, ce qui améliore la facilité d'utilisation.Chaque étape est placée sur une grille comme présenté dans le figure ci dessous. Ce design offre un bon compromis entre les exigences souvent opposées : performance, réalisme, flexibilité. Voici les différentes étapes de ce pipeline :



(a) Pipeline présenté dans le papier image



(b) Structure de grilles utilisé pour la génération

FIGURE 2 – Images tirés du papier

Terrain de Base Généré à l'aide de fonctions de bruit, notamment du simplex noise multi-octaves. Cela permet une création rapide et flexible d'un terrain général, sans se soucier des détails réalistes dès cette étape. L'utilisateur peut ajuster les paramètres, comme le nombre d'octaves ou le seuil de niveau de la mer.

Simulation climatique Cette étape a pour but de calculer la distribution des biomes la plus réaliste possible. Pour cela aucun bruit n'est utilisé mais une approche basée sur la physique a été choisie elle reste simple pour qu'elle soit relativement rapide contrairement a d'autres méthodes basées sur la simulation physique. Elle se compose de 4 étapes :

- **Température** : Calculée par interpolation bilinéaire ou sinusoïdale, en prenant en compte l'altitude (plus c'est haut, plus c'est froid). Cela permet d'avoir un gradient entre régions chaudes et froides.
- **Vent** : Le vent est simulé en définissant des forces aux quatre coins de la carte. Une approche itérative propage les directions de vent dans un champ vectoriel, en combinant les directions des cellules voisines et en ajoutant de petites perturbations aléatoires pour simuler les micro-variations. La proximité d'un coin renforce la persistance de son influence, produisant ainsi un lissage réaliste ou des annulations aux frontières entre courants principaux.
- **Précipitations** : La répartition des précipitations est calculée de manière itérative, en utilisant les données de vent et de température. Les cellules d'eau servent de sources d'humidité, avec une évaporation dépendant de la température. L'humidité est ensuite transportée par le vent vers les cellules voisines, avec une dispersion partielle vers les cellules adjacentes. Cette méthode permet de de simuler des phénomènes plus avancés comme les ombres pluviométriques ou l'effet de foehn.

- **Placement des biomes** : Calcul de la grille de Biomes en fonction des propriétés calculées, en particulier la température et les précipitations. À chaque paire de valeurs température-précipitation est ainsi associé un identifiant de biome spécifique, selon une table.

Enrichissement du terrain : Pour enrichir le terrain de base, les auteurs utilisent une approche par exemple en intégrant des DEMs (modèles numériques d'élévation) spécifiques à chaque biome, tirés de sources disponibles publiquement. Les DEMs sont une représentation numérique de la hauteur du terrain sur une surface donnée, généralement sous forme d'une grille régulière où chaque cellule contient une valeur d'altitude. Cette méthode offre un réalisme élevé avec peu d'efforts. Pour obtenir des transitions naturelles entre biomes, les frontières sont déformées avec du bruit fractal (simplex noise), générant une grille de biomes à plus haute résolution. Ensuite, une convolution avec un noyau réglable permet de mélanger les DEMs voisins selon la proportion de chaque biome dans la zone. Le résultat est une somme pondérée des DEMs, fusionnée au terrain de base pour produire une carte de hauteur finale réaliste et cohérente.

Placement d'objets : Dans la dernière étape, les biomes sont peuplés par des objets 3D (assets) placés selon un modèle itératif. Ce modèle permet des distributions naturelles et adaptables aux transitions entre biomes. Les objets sont sélectionnés depuis une base d'assets (arbre, rocher, etc.), chacun possédant des propriétés spécifiques (ex. : tolérance à l'ombre, distance minimale, etc.). Le placement se fait via un échantillonnage de type Poisson-disk, adapté pour gérer des distributions complexes avec contraintes. Les objets sont traités par catégories (organique/inorganique, petite/grande taille) pour une répartition plus réaliste. On obtient une répartition uniforme et performante d'objets dur le terrain généré.

Résultats obtenus Voici les résultats présentés par les auteurs :

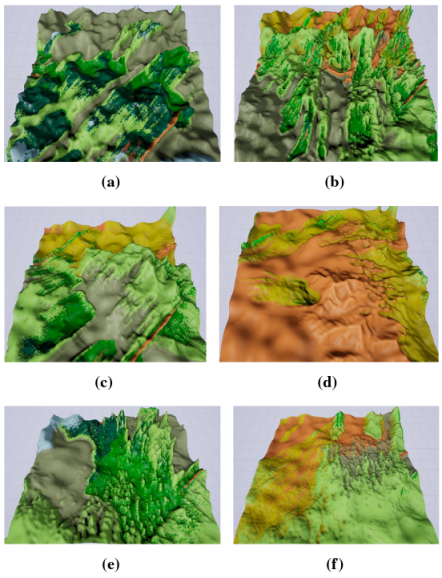


FIGURE 3 – Un ensemble de terrains différents générés avec différents paramètres

Conclusion Ce papier présente une méthode intéressante pour la génération de terrain et de placement d'objets mais nous n'avons pas décidé de suivre cette méthode pour notre projet, celle ci est complexe et les résultats présentés ne nous ont pas convaincus. En effet pour une méthode qui met l'accent sur le réalisme nous trouvons les résultats trop aléatoires et peu esthétiques. Nous avons quand même utilisé certaines idées comme la table en fonction de l'élévation et de l'humidité pour les biomes.

2.3.2 papier 2

2.3.3 papier 3

2.3.4 papier 4

3 Développement

3.1 Génération de la carte

Pour ce projet nous voulions créer un outil pour créer une carte procéduralement. Pour cela nous avons écrit un programme en C++ qui crée un fichier .OBJ a partir de cartes de bruits.

Cartes de bruits Une carte de bruit, ou "noisemap", dans le contexte de la génération procédurale, est une technique utilisée pour créer des textures, des terrains ou des environnements de manière algorithmique. Les noisemaps dans la génération procédurale sont utilisées pour introduire des variations naturelles et réalistes dans les modèles générés par ordinateur.
La génération d'une noisemap implique l'utilisation de fonctions de bruit, telles que le bruit de Perlin, le bruit de Simplex ou d'autres algorithmes de bruit procédural. Ces fonctions génèrent des valeurs pseudo-aléatoires qui varient de manière continue et douce à travers l'espace. L'utilisation la plus elementaire de ce genre de cartes et de lier l'élévation du terrain aux valeurs de la carte de bruit.

FIGURE 4 – Visualisation de Carte de bruit

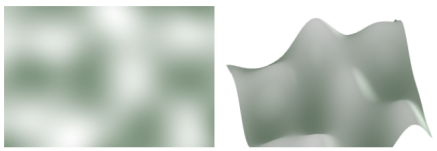
Nous pouvons manipuler l'aléatoire des fonctions qui generent ce genre de carte grace a differents parametres :

- La fréquence dans le contexte des fonctions de bruit, comme le bruit de Perlin, détermine la granularité ou la finesse des détails dans la carte de bruit générée. Une fréquence élevée signifie que les variations de bruit se produisent plus rapidement, ce qui entraîne des détails plus fins et plus nombreux dans la carte. À l'inverse, une fréquence basse produit des variations plus lentes et plus douces, résultant en des caractéristiques plus larges et moins détaillées.
- Les octaves sont utilisées pour ajouter de la complexité et du réalisme aux cartes de bruit. Chaque octave est une couche supplémentaire de bruit générée à une fréquence différente. En superposant plusieurs octaves, on peut créer des textures plus riches et plus variées. Les octaves sont généralement ajoutées avec des fréquences de plus en plus élevées et des amplitudes de plus en plus faibles.

Effet sur la Carte Générée

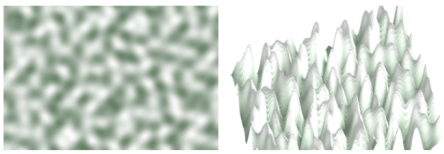
- *Fréquence Basse* : Une fréquence basse produit des caractéristiques larges et douces. Par exemple, dans la génération de terrains, une fréquence basse peut créer des collines et des vallées larges et douces.
- *Fréquence Élevée* : Une fréquence élevée produit des détails fins et nombreux. Par exemple, dans la génération de terrains, une fréquence élevée peut ajouter des rochers, des crevasses et d'autres détails fins à la surface.
- *Octaves Multiples* : L'utilisation de plusieurs octaves permet de combiner des caractéristiques larges et douces avec des détails fins. Par exemple, une première octave avec une fréquence basse peut créer les grandes formes du terrain, tandis que des octaves supplémentaires avec des fréquences plus élevées ajoutent des détails fins comme des rochers et des textures de surface.

Voici des exemples en images de la différence.



(a) Exemple carte et élévation avec une fréquence basse

image



(b) Exemple carte et élévation avec une haute fréquence

FIGURE 5 – Comparaison de niveau de fréquences

Génération de biomes Sur notre carte nous voulons différents biomes c'est a dire différents environnements repartiimages le plus naturellement possible. Pour cela en plus de la carte de bruit qui va gérer l'élévation du terrain nous allons générer une autre carte de bruit qui va simuler l'humidité du milieux. En combinant ces deux cartes on peut obtenir des biomes plutôt cohérents. En plus de cela nous pouvons définir une élévation minimale qui va permettre de créer des océans. Enfin nous avons défini une aire de jeu circulaire au milieu de la carte celle ci sera plate mais utilise quand même les noisemap pour gérer les biomes.

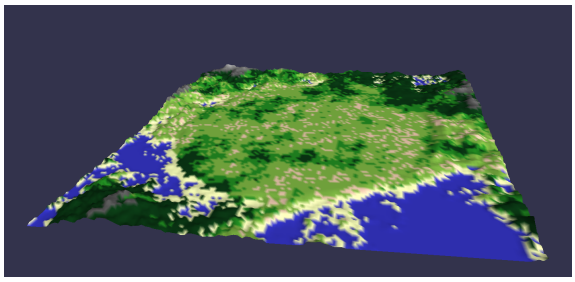
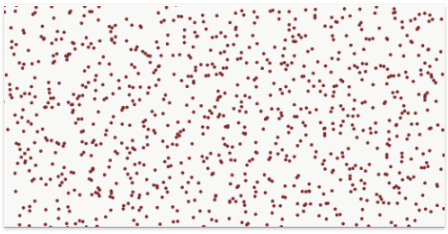


FIGURE 6 – Une carte générée avec notre code

Placer des ressources Un autre défi de la création de la carte est de placer des ressources sur la carte en favorisant les potentielles interaction entre les royaumes, c'est a dire créer des ressources qui serons rares et ne pas les répartir de manière uniformes pour obliger les royaumes a interagir entre eux. Si on prend simplement des positions aléatoires sur la carte par exemple on se retrouve avec un résultat trop uniforme on aimerait avoir une génération avec des trous et des endroits plus concentrés. La solution que nous avons trouvé pour cela ce sont les "Jittered grid" on peut traduire cela par "grille perturbée". Le principe est de générer une grille carrée ou hexagonales et d'appliquer une perturbation aux points de la grille. Voici le résultat.

(a) Grille avant la perturbation



(b) Gille après la perturbation

FIGURE 7 – Visualisation de jittered grid

On voit apparaître des zones avec peu de points et d'autres avec beaucoup de points.

Résultat sur la carte Les triangles violets représentent les positions des ressources.

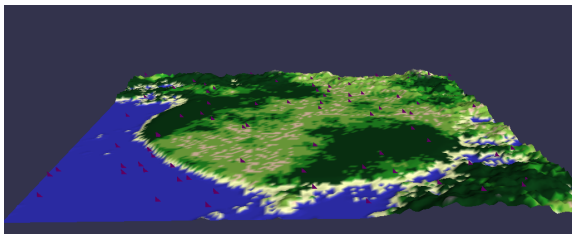


FIGURE 8 – Carte avec position de ressources générés

Il suffira de re-exécuter ce code le même nombre de fois que de ressources. Les emplacements des ressources sont enregistrés dans un fichier csv qui pourra être lu par le moteur de jeu.

3.2 Moteur de jeu

Technologies L'objectif étant de créer un jeu par navigateur, nous avons dû choisir entre trois approches distinctes :

- **Moteur de jeu existant et export HTML5**
  Avec cette méthode, nous utilisons un moteur de jeu (par exemple, Godot ou Unity) pour développer le jeu puis l'exporter en HTML5. Cependant, cela nous donne moins de contrôle sur les performances et les capacités de notre jeu.
- **typescript + ThreeJS**
  typescript est une surcouche de javascript permettant un typage fort et une programmation par classe plus solide. *ThreeJS* est une librairie javascript permettant de faire de la 3D sur navigateur (*voir WebGL*) Cette méthode nous oblige à créer beaucoup plus de choses de zéro, mais elle nous offre aussi davantage de contrôle sur le résultat et est la plus intéressante d'un point de vue pédagogique.
- **Angular + ThreeJS**
  Angular est un framework en typescript pour la création de sites web. Il propose extension pour fonctionner avec *ThreeJS*. Cette méthode nous facilite la création d'interfaces utilisateur et la gestion de la partie web en général. Cependant, cela entraîne un coût en temps de formation et en performances.

Comme nous sommes des élèves d'Imagine, nous avons décidé de partir sur la deuxième option. C'est celle qui nous permet le mieux de mettre en action et d'améliorer nos compétences. De plus, cela nous donne l'occasion d'apprendre le développement web orienté 3D et applications interactives.

Le design d'un moteur de jeu La manière classique de structurer un jeu vidéo (ainsi que la majorité des applications 3D interactives) est par l'usage d'un concept d'**entité de jeu**. Classiquement, une entité a une existence dans le monde du jeu, par exemple, une montagne, un personnage, une zone qui détecte d'autres entité, etc.
Les entités sont souvent organisées de manière hiérarchique dans une structure arborescente appelée **Graphe de scène**. Une entité qui est enfante d'une autre hérite généralement de sa transformation 3D : en d'autres termes, les mouvements de l'entité parente sont répercutés sur l'entité enfante (un chevalier qui court entraîne aussi l'épée qu'il tient en main). D'autre part, ces entités sont spécialisées selon leur utilité : une porte, un zombie ou une explosion nécessitent un traitement différent de la part du développeur.
La méthode d'implémentation du graphe de scène et des entités varie beaucoup selon le moteur :

- *Gamebryo*, un moteur de jeu sorti en 1998 et sur lequel sont basés plusieurs moteurs récents (par exemple, celui du jeu *Skyrim*), utilise une hiérarchie d'entités qui peuvent hériter de la transformation et d'autres propriétés de leur parent. Il est par ailleurs possible d'ajouter des comportement aux entités en *LUA* ou en *C++*
- *Godot* utilise un graph de "Noeuds". Chaque noeud est une classe héritée de la classe *Node* lui donnant des comportement spécifiques (par exemple, "corps de simulation physique" ou "maillage 3D".
- *Source Engine* sépare les objets du monde en deux catégories, les objets "Monde", qui sont statiques, et les objets "Entité", qui requiert l'exécution régulère de leur logique (customisable en LUA)
- *Unity* utilise un arbre de *Gameobjects*, lesquels contiennent une liste de composants. Les composants spécialisent le comportement de l'objet.
- *Dark Engine*, un autre moteur publié en 1998 et utilisé notamment pour les jeux *System Shock 2* et *Thief*, utilise une architecture entity-component-system, dont nous expliquerons le fonctionnement plus tard.

Pour faire notre choix parmi toutes ces possibilités (et davantage encore !), nous avons dressé un cahier le cahier des charges de notre jeu : Tout d'abord, nous devions pouvoir afficher et mettre à jour **plusieurs centaines d'entités** sans chute majeure de performances. Nous avions besoin

(a) Thief, 1998 utilise une architecture ECS

(b) Disco Elysium, 2019, utilise une architecture Entity-Component sur le moteur Unity

de pouvoir recevoir du serveur un **flux constant** de données et de le propager dans le jeu. Nos entités seraient peu complexes (non-nécessité d'une hiérarchie d'entités), et peu variées dans leur logique (préférence pour une approche *data-driven*).

3.2.1 Entity-component-system

L'architecture Entité-Composant-Système (souvent abrégé ECS) est une architecture permettant de décrire des entités en séparant totalement les données des traitements. Comme son nom l'indique, l'ECS est basé sur trois concepts clés :

- **Les entités** (dans le jargon ECS) représentent exactement ce que l'on a appelé jusqu'ici "entité" : *quelque chose* qui est dans le monde du jeu. Par elles mêmes, elles ne contiennent ni logique ni donnée, mais elle sont chacune associées à un ensemble de composants.
- **Les composants** sont des conteneurs de données typés. Leur type indique la fonction de leurs données. Par exemple, un composant 'Rendu' pourra contenir un maillage 3D et un composant 'Transformation', une matrice de $M_4(\mathbb{R})$. Les composants ne contiennent aucune logique, mais c'est les systèmes qui vont exploiter leurs données.
- **Les systèmes** sont des fonctions qui agissent sur les entités de manière régulière ou ponctuelle. Chaque système n'agit que sur un sous-ensemble des entités qui possèdent les composants nécessaire à l'application du système. Par exemple, un système "Affichage3D" pourrait agir sur toutes les entités ayant au moins les composants "Rendu" et "Transformation".

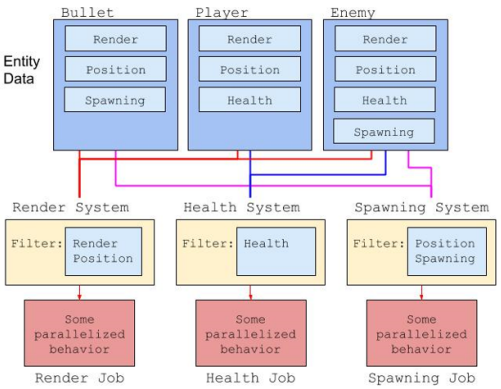


FIGURE 10 – Diagramme produit par Intel illustrant une architecture ECS

Les avantages de l'ECS sont aussi nombreux que ses problèmes. La centralisation des traitements permet de gagner en performance lorsque le nombre d'entités est bien plus grand que le nombre de systèmes (donc que chaque système agit sur beaucoup d'entités en moyenne). En outre, la

concentration des données dans les composants permet une approche *data-driven*[1] D'un autre côté, pour des cas d'utilisation avec peu d'entités et beaucoup de variance dans la logique de traitement, l'ECS est peu souhaitable car il est moins intuitif à utiliser que la plupart des alternatives, et les gains de performance sont moindres voir négatifs.
Nous utilisons une architecture ECS car ses forces coïncident avec notre cas d'utilisation : Traitement de beaucoup d'entités avec un nombre de traitement relativement restreint, et spécialisation d'un grand nombre d'entités par une approche orientée-données.

3.2.2 Notre moteur

Pour rappel, nous avons choisi de développer notre propre moteur en *typescript*, en utilisant la librairie de rendu 3D *ThreeJS*.
ThreeJS nous offre une interface haut niveau vers le WebGL. En d'autres termes, on lui délègue le rendu graphique. Cependant, le reste des concepts servant de base au jeu doivent être implémentés par nos soins. On décide de créer un moteur de jeu basique, spécialisé pour le jeu que l'on crée (en opposition à un moteur généraliste comme Godot ou Unreal Engine). Ce moteur est composé de plusieurs modules :

- **ECS**
- **Components**
- **Systems**
- **Input**
- **User Interface**
- **Game**

Ces modules fonctionnent ensemble pour fournir une base solide à la création du jeu. Expliquons chacun d'eux en détails.

ECS Notre implémentation de l'architecture ECS est basée sur *cet article de 2021 écrit par Maxwell Forbes*. Les entités sont simplement représentées par un entier. La classe *ECS* lie ces identifiants à un ensemble d'objets héritant de *Component*. La classe *ECS* contient aussi l'ensemble des systèmes.
Quand un système, une entité ou un composant est ajouté(e) ou enlevé(e) l'ECS, la liste des entités concernées par chaque système est mise à jour. Une fonction permet d'appliquer le traitement de chaque système aux entités concernées. Voici un exemple-jouet d'utilisation de notre système ECS pour faire rebondir des objets sur le sol.
TODO EXEMPLE DE CODE
Nous définissons une liste de composants et systèmes "*Core*" : des éléments généraux et fondamentaux au jeu. Comme notre moteur est très lié à notre jeu, cette frontière n'est pas toujours claire, par exemple, le code de synchronisation avec le serveur et le code d'interaction joueur-monde sont tous deux définis dans le jeu. Ils seront explicités dans la partie suivante qui portera sur le jeu en lui même.

Components (*Core*) TODO

Systems (*Core*) TODO

Input TODO

User Interface Plusieurs librairies d'interface utilisateur existent. Après une recherche extensive, aucune n'a semblé satisfaisante. L'un des *motto* du projet est de faire les choses simplement et efficacement, ce qu'aucune librairie candidate ne semblait faire.
C'est pourquoi nous avons choisi de créer notre propre solution d'interfaces utilisateur.
TODO clément c'est ta partie hehe

1. "régie par les données". Paradigme de programmation où les données sont spécifiées en amont des traitements. Le langage de transformation XML XSLT en est un exemple.

Game (Classe) TODO

3.3 Interface

3.4 Assets

3.5 Développement gameplay

3.5.1 Mode "péon"

Mini-jeux Les joueurs péons s'occupent de récupérer les ressources de construire les bâtiments. Il faut donc intégrer ces mécaniques dans le jeu. Plutôt que de simplement cliquer sur un arbre pour récupérer le bois par exemple nous avons décidé d'intégrer des mini-jeux très simples. Ces mini-jeux sont de toutes nouvelles scènes TREEjs qui se trouvent en dehors du système ECS. Quand un joueur interagi avec une ressource ou autre interaction du genre, une fenêtre de mini-jeu s'ouvre au centre de son écran. Le mini-jeu diffère en fonction de l'action a accomplir et plus on fait bien le mini-jeu plus on obtient de ressources. Par exemple pour la coupe du bois, plus vous faites d'aller retour avec votre souris plus vous obtiendrai de bûches comme si la souris était une scie.

3.5.2 Mode "seigneur"

3.6 communication avec le backend

4 Conclusion