Rapport Final du Projet Long Technologie Objet

Simulation d'un Monde Évolutif Généré Procéduralement.

Équipe AB-1
ARMAGHAN Théo, BOHIN Arthur, CHEIBANY Mina,
EL MARIKY Ilias, GALET Matteo,
GUESSOUS Saad, HESS Florian.

26 mai 2025

Table des matières

1 Rappel Succinct du Sujet

Notre projet, "Simulation d'un Monde Évolutif Généré Procéduralement", avait pour ambition de créer un univers virtuel en voxels capable de prendre vie et de se transformer de manière autonome. L'idée centrale était d'observer l'émergence de dynamiques naturelles, comme la formation de paysages variés et le développement d'un écosystème simple, sans intervention directe d'un joueur. Un aspect qui nous a particulièrement motivés était le défi de générer et d'afficher un monde cohérent, peuplé d'arbres et d'animaux, offrant ainsi une base pour une future simulation d'évolution. Le joueur est conçu comme un observateur, pouvant explorer ce monde en constante, bien que lente, transformation.

2 Principales Fonctionnalités

Au cours de nos trois sprints de développement, nous nous sommes concentrés sur la mise en place des briques fondamentales de notre simulateur. Voici un aperçu des grandes fonctionnalités que nous visions et de leur état d'avancement à la fin de cette troisième itération.

Génération Procédurale d'un Monde Dynamique

La création d'un monde voxel varié et évolutif était au cœur de nos préoccupations.

— **Terrain et Biomes :** Nous avons réussi à générer un terrain en voxels, capable de former des montagnes, des plaines et d'intégrer des étendues d'eau. La base pour différents biomes est là, avec des paramètres influençant la topographie (Sprint 1 pour les fondations, Sprint 2 et 3 pour l'amélioration des types de terrains et l'intégration de paramètres de génération via les menus).

— **Algorithmes de Bruit :** L'utilisation du bruit de Perlin a été essentielle pour obtenir des paysages naturels. Nous avons consacré du temps à son implémentation dès le Sprint 1.

— **Évolution du Terrain :** Nous avons jeté les bases pour l'évolution de la flore avec la capacité de faire apparaître des structures végétales comme des arbres (Sprint 2). L'évolution plus poussée du terrain, comme l'érosion ou la modification dynamique des cours d'eau, a été repoussée faute de temps, car nous avons priorisé la consolidation des aspects de génération et d'affichage.

Simulation d'un Écosystème Vivant

Donner vie à notre monde était notre second grand objectif.

— **Faune et Flore :** Nous avons implémenté les structures de données pour les espèces dès le Sprint 1. Au Sprint 2, nous avons introduit des entités animales basiques et des structures pour les plantes (arbres). Le Sprint 3 a permis de commencer l'intégration de modèles 3D et d'animations pour certains animaux, et d'ajouter plus de types de structures/plantes, bien que ce chantier reste partiellement terminé et offre de belles perspectives d'amélioration.

— **Adaptation et Équilibre :** Un système de score de compatibilité entre les espèces et leur environnement a été conçu au Sprint 1, posant une première pierre à la simulation d'adaptation. Les mécanismes d'évolution plus complexes (reproduction, dynamique des populations détaillée) n'ont malheureusement pas pu être implémentés durant ces itérations, notre focus ayant été mis sur la création d'entités fonctionnelles et leur intégration dans le monde.

Interaction du Joueur et Visualisation

L'expérience de l'utilisateur, même en tant qu'observateur, était importante.

— **Exploration :** Le joueur peut se déplacer librement pour observer le monde. Ceci a été fonctionnel dès le Sprint 1, et amélioré au Sprint 2 avec des modes de caméra libre, première et troisième personne, et une gestion physique du joueur terminée au Sprint 3.

— **Paramétrage et Observation :** Nous avons mis en place des menus de configuration (terminés au Sprint 3) permettant à l'utilisateur d'ajuster les paramètres de génération du monde. Les interfaces HUD pour afficher des statistiques détaillées sur le monde et ses habitants, ainsi qu'une mini-carte, sont partiellement implémentées et nécessiteraient plus de travail pour être pleinement fonctionnelles.

— **Affichage et Rendu :** L'affichage 3D en voxels a été une base de travail dès le Sprint 1. Nous avons également finalisé des améliorations du rendu visuel (ombres, éclairage) au Sprint 3.

Évolution et Civilisation (Perspectives)

Certaines ambitions initiales ont dû être mises de côté par réalisme.

— **Civilisation Primitive et Sélection Naturelle Poussée :** L'émergence d'une civilisation ou des mécanismes de sélection naturelle très élaborés étaient des objectifs à long terme. Ils n'ont pas été entamés, car ils dépendaient de la stabilisation de nombreuses autres fonctionnalités de base.

3 Diagramme de Classe

Pour organiser notre code et assurer une certaine modularité, nous avons structuré notre application en plusieurs paquetages principaux. Cette organisation a évolué au fil des sprints pour mieux refléter les différentes responsabilités au sein du système. Voici la structuration de notre projet :

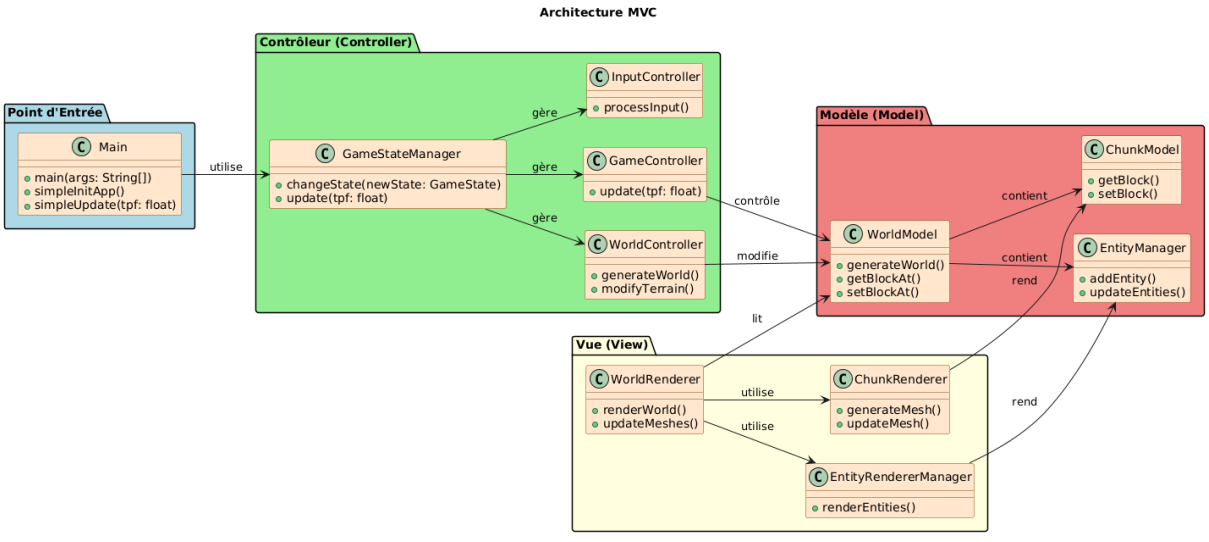

FIGURE 1 – Vue d'ensemble de l'architecture de notre application.

3.1 Point d'Entrée

La classe `Main` constitue le point d'entrée de l'application. Elle possède les méthodes suivantes :

— `main(args: String[])` : point d'entrée de l'application.

— `simpleInitApp()` : initialise les ressources et les composants.

— `simpleUpdate(tpf: float)` : boucle principale de mise à jour (appelée à chaque frame).

Cette classe utilise la classe `GameStateManager` pour gérer l'état global du jeu.

3.2 Contrôleur (Controller)

Le contrôleur est centralisé autour de la classe `GameStateManager`, qui gère les différents contrôleurs spécifiques et permet la gestion des états du jeu.

— **`GameStateManager`** :
  — Méthodes : `changeState(newState: GameState)`, `update(tpf: float)`.
  — Gère les contrôleurs suivants :
    — `InputController` : s'occupe de traiter les entrées utilisateur (`processInput()`).
    — `GameController` : s'occupe de la logique de jeu globale (`update(tpf: float)`).
    — `WorldController` : gère la génération et la modification du monde virtuel (`generateWorld()`, `modifyTerrain()`).

Ces contrôleurs sont responsables d'agir sur les modèles en réponse aux événements du jeu ou aux interactions utilisateur.

3.3 Modèle (Model)

Le modèle contient toutes les données et la logique associée au monde du jeu. Il est structuré autour de plusieurs classes :

— **`WorldModel`** :
  — Méthodes : `generateWorld()`, `getBlockAt()`, `setBlockAt()`.
  — Contient les instances de `ChunkModel` et `EntityManager`.

— **`ChunkModel`** :
  — Méthodes : `getBlock()`, `setBlock()`.
  — Représente les portions (chunks) du monde.

— **`EntityManager`** :
  — Méthodes : `addEntity()`, `updateEntities()`.
  — Gère les entités dynamiques du monde (joueurs, objets...).

Le `WorldModel` centralise la logique du monde en modifiant ou en lisant les données des chunks et des entités selon les actions demandées par les contrôleurs.

3.4 Vue (View)

La partie Vue est responsable de l'affichage et de la représentation graphique du monde. Elle est composée de trois classes principales :

— **`WorldRenderer`** :
  — Méthodes : `renderWorld()`, `updateMeshes()`.
  — Utilise deux renderers pour représenter visuellement les données :
    — `ChunkRenderer` : génère et met à jour les maillages des chunks (`generateMesh()`, `updateMesh()`).
    — `EntityRendererManager` : gère l'affichage des entités (`renderEntities()`).

Ces composants Vue lisent les données du modèle pour les rendre à l'écran, sans les modifier.

3.5 Relations entre les composants

La classe Main démarre l'application et délègue la gestion au GameStateManager, qui contrôle les différents contrôleurs, à savoir InputController, GameController et WorldController. Ces contrôleurs manipulent les données contenues dans le modèle, par exemple lorsque le WorldController modifie les données du WorldModel. La Vue, quant à elle, lit les données du modèle afin de les afficher à l'écran. Les dépendances sont clairement établies et respectent la séparation des préoccupations : le modèle ne connaît pas la vue ni le contrôleur, la vue n'interagit qu'avec les données à afficher, et les contrôleurs orchestrent le flux global de l'application.

Ce choix nous a semblé pertinent car il permet une bonne séparation des préoccupations. Le code du modèle, de la vue et du contrôleur sont distincts, ce qui facilite la compréhension, la maintenance et l'évolution de chaque partie. Par exemple, nous pouvions modifier la manière dont le monde est affiché (la Vue) sans impacter la logique de génération du monde (le Modèle).

4 Principaux Choix de Conception et Réalisation

4.1 Choix Technologiques

Dès le début, nous avons opté pour le langage **Java** en raison de sa portabilité et de notre familiarité collective avec cet environnement. Pour la partie graphique et la gestion de la 3D, nous avons choisi le moteur de jeu **JMonkeyEngine3**. Ce moteur nous a offert un cadre robuste pour la gestion des scènes 3D, des entrées utilisateur, et des aspects de rendu, tout en étant suffisamment flexible pour y intégrer notre logique de génération procédurale de voxels.

4.2 Algorithmes Clés et Optimisations

4.2.1 Génération Procédurale avec Bruit de Perlin

La génération des paysages repose sur l'utilisation du bruit de Perlin. Obtenir des résultats variés et contrôlables (mondes plus montagneux, plaines, etc.) a représenté un défi intéressant. Il a fallu de nombreux essais et ajustements des paramètres du bruit (fréquence, amplitude, octaves, persistance) pour parvenir à générer des terrains qui correspondaient à nos attentes et qui pouvaient être influencés par les réglages de l'utilisateur via les menus.

4.2.2 Optimisation de l'Affichage des Voxels

Un monde en voxels peut rapidement devenir très coûteux à afficher si chaque cube est dessiné intégralement. Pour optimiser cela, notre première approche a été de n'afficher que les faces des cubes qui sont en contact avec l'air (ou un bloc transparent). Ainsi, les faces internes, non visibles, ne sont pas envoyées à la carte graphique, réduisant considérablement le nombre de polygones à traiter. Nous avions envisagé d'aller plus loin avec des techniques comme le "greedy meshing" pour fusionner les faces coplanaires de même type en de plus grands polygones, mais nous n'avons malheureusement pas eu le temps de l'implémenter durant ces sprints. Cela reste une piste d'optimisation future.

4.3 Problèmes Rencontrés et Solutions Apportées

Au cours du projet, nous avons fait face à plusieurs défis :

— **Évolution de la Conception des Biomes :** Initialement, nous pensions proposer à l'utilisateur une sélection de biomes prédéfinis. Cependant, lors de nos discussions au Sprint 1, nous avons réalisé que cela limiterait la créativité et l'aspect "évolutif" que nous visions. Nous avons donc réorienté notre conception pour permettre à l'utilisateur de définir des paramètres environnementaux (humidité, température, relief via les menus, etc.), à partir

desquels le système génère dynamiquement un biome et sélectionne des espèces adaptées. Cela nous a semblé plus en phase avec l'idée d'un monde qui se construit selon des règles plutôt que d'être entièrement prédéfini.

— **Gestion des Paramètres du Bruit de Perlin :** Comme mentionné, trouver les bons réglages pour le bruit de Perlin afin de créer des terrains variés et intéressants, tout en les liant aux paramètres utilisateur, a nécessité beaucoup d'expérimentation.

— **Complexité de la Simulation d'Évolution :** Nous avions sous-estimé le temps nécessaire pour développer des mécanismes d'évolution crédibles pour la faune et la flore (reproduction, prédation, adaptation génétique, etc.). Face aux échéances et à la nécessité de consolider d'abord la génération du monde et l'affichage, nous avons dû repousser l'implémentation de ces aspects. Nous avons posé les bases avec les classes d'entités et le système de compatibilité, mais le véritable comportement évolutif reste à faire.

— **Performances :** Bien que notre technique d'affichage des faces visibles ait aidé, la gestion d'un grand monde voxel reste gourmande. Des optimisations plus poussées (comme le greedy meshing ou un système de Level of Detail (LOD) plus avancé) seraient nécessaires pour des mondes de très grande taille.

5 Organisation de l'Équipe et Mise en Œuvre des Méthodes Agiles

Notre équipe, composée de sept membres, a adopté une approche agile pour mener à bien ce projet. Cette méthodologie nous a semblé la plus adaptée pour gérer la complexité et l'aspect itératif du développement d'une simulation.

5.1 Fonctionnement en Sprints

Le projet a été découpé en trois sprints principaux. Au début de chaque sprint, nous tenions une réunion de lancement (que nous appelions notre "mêlée" de mise en place) pour définir les objectifs prioritaires et répartir les macro-tâches issues du "Sprint Backlog". Ces tâches étaient ensuite détaillées et suivies via un tableau Trello partagé, nous permettant d'avoir une vision claire de ce qui était "À faire", "En cours" et "Terminé".

5.2 Répartition des Tâches et Collaboration

Pour la réalisation des tâches, nous avons souvent fonctionné en duos. Nous essayions de varier la composition des duos d'un sprint à l'autre pour favoriser le partage des connaissances et des compétences au sein de l'équipe. La communication quotidienne se faisait principalement via un serveur Discord dédié au projet, où nous pouvions échanger rapidement, poser des questions et partager nos avancées. SVN a été notre outil de choix pour la gestion des versions du code.

5.3 Suivi et Rapports

Des "mêlées" brèves et régulières (plus informelles que la réunion de lancement) nous aidaient à synchroniser nos efforts et à identifier rapidement les blocages. À la fin de chaque sprint, chaque membre rédigeait un rapport individuel synthétisant son travail sur la période écoulée, ce qui alimentait ensuite le rapport de sprint collectif.

5.4 Bilan de l'Approche Agile

L'utilisation des méthodes agiles a été très bénéfique pour nous. Elle nous a permis de :

— **Mieux répartir le travail :** Le découpage en tâches claires et la priorisation au sein des sprints ont facilité la distribution équitable du travail.

— **Mieux gérer notre temps :** Avoir des objectifs définis pour chaque sprint nous a aidés à rester concentrés et à mieux estimer le temps nécessaire.

— **Améliorer notre communication et notre cohésion :** Les points réguliers et la nécessité de collaborer étroitement sur les tâches ont renforcé la communication au sein de l'équipe.

— **Être plus réactifs au changement :** Comme l'a montré notre réorientation sur la génération des biomes, l'approche agile nous a donné la flexibilité nécessaire pour adapter notre conception en cours de route face à de nouvelles idées ou contraintes.

Bien sûr, estimer la charge de travail et tenir tous les objectifs de sprint a parfois été un défi, mais globalement, cette méthode a été un atout majeur pour le projet.