Rapport - Projet de fin d'étude
Master Informatique pour l'Image et le Son
Interactive visualization of complex systems for collaborative artificial life research

Auteurs :

Victor Barrey (victor.barrey@etu.u-bordeaux.fr)
Léo Dupouey (leo.dupouey@etu.u-bordeaux.fr)
Bastien Morel (bastien.morel@u-bordeaux.fr)
Jonathan Moze (jonathan.moze@etu.u-bordeaux.fr)

Responsables de projet :

Pascal Barla (Inria - Manao)
Clément Moulin-Frier (Inria - Flowers)

Lien du projet : https://github.com/BastienMrl/Complex-systems-visualization

Lien vers démonstrations vidéos :
https://www.youtube.com/playlist?list=PLicm65r9hfEL8m3mh3B4AFHPbdZ8ZIOW9

Mars 2024

Table des matières

Introduction

Vocabulaire

Système complexe

On qualifie de complexe un système composé d'une multitude d'entités dont les interactions locales font émerger des propriétés globales difficilement prédictibles par la seule connaissance des propriétés de ces entités. Souvent, le seul moyen d'étudier la dynamique de ces système est de les observer (dans le monde naturel) ou de les simuler à partir d'un modèle. Des exemples de systèmes complexes dans le monde naturel sont : Une nuée d'oiseaux, un réseau social, des écosystèmes, des colonies de fourmis ou encore un réseau de neurones. Des exemples synthétiques (simulés sur ordinateur) sont : des systèmes d'oscillateurs couplés, des automates cellulaires, des générateurs de fractales (par ex. l'ensemble de Mandelbrot).

Auto-organisation

Mécanisme à partir duquel des interactions locales entre constituants d'un système complexe conduisent à l'émergence de structure globales, qui ne peuvent pas être observées au niveau de ses constituants. Par exemple les formes complexes observées dans les nuées d'oiseaux.

Automate cellulaire

Modèle informatique constitué d'une grille discrète dans laquelle chaque cellule est caractérisée par un état. L'état de chaque cellule est modifié à chaque pas de temps en fonction de l'état de ses voisins, menant à des phénomènes d'auto-organisation. Par exemple : le jeu de la vie de Conway.

Contexte

Dans la nature, nous pouvons rencontrer de nombreux systèmes dynamiques qui s'auto-organisent. Leur étude révèle très souvent l'émergence de comportements complexes, évolués et structurés afin de survivre dans leurs environnements. On peut particulièrement penser à l'organisation d'une fourmilière, des bancs de poissons ou les nuées d'oiseaux.

(a) Nuée d'oiseaux. *(b) Instance du Jeu de la vie de Conway.*

Figure 1 – Exemples de système complexe s'auto-organisant. (a) Un système naturel[1]*. (b) Un système synthétique*

Dans le monde artificiel, les automates cellulaires sont des systèmes complexes où nous pouvons observer l'émergence de motifs localisés spatialement, parfois appelés "créatures". Ces automates peuvent servir de modèles pour étudier certaines questions sur les origines de structures auto-organisées dans la nature, par

1. Image issue de https://medium.com/@adityaananthram/algorithms-in-nature-part-1-e80e80b29719

exemple sur les origines de la vie. On parle de Vie Artificielle, un domaine de recherche cherchant à modéliser et à simuler les mécanismes d'auto-organisation favorisant la formation de structures montrant certains degrés d'autonomie. [2, 8]

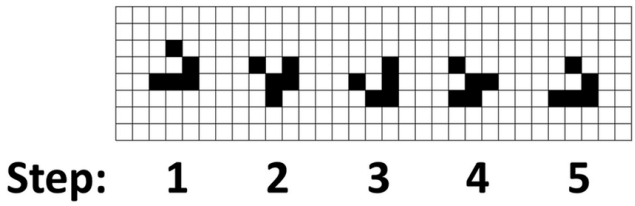
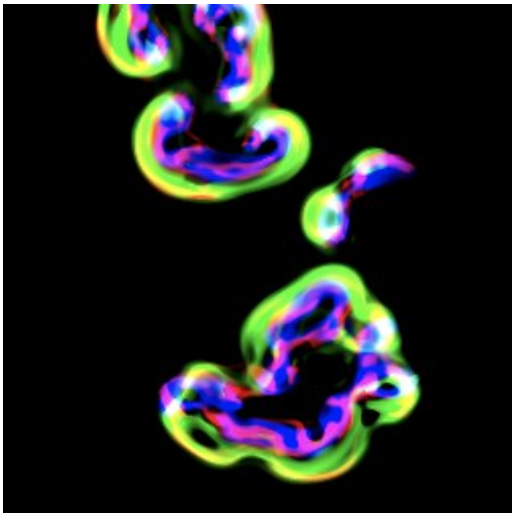


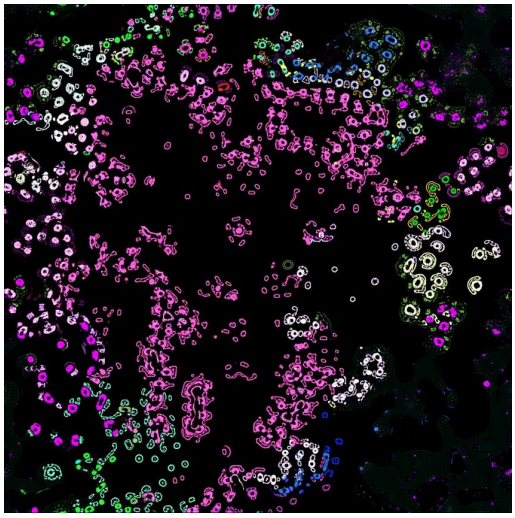
Figure 2 – Un glider*, une structure auto-organisée, émergeant au sein du Jeu de la vie. Cette structure effectue un cycle de 5 pas de temps, en se déplaçant diagonalement. Image issue de [10]*

Problématique

Actuellement, la recherche dans le domaine de la vie artificielle connaît un nouvel essor impressionnant suite aux avancées techniques des processeurs graphiques. En effet ceux-ci permettent maintenant de simuler des systèmes complexe de taille conséquente, passant d'un automate cellulaire simple à un automate continu dans l'espace et le temps, par accélération graphique. Avec toutes ces avancées, ces nouveaux systèmes ont aussi la capacité de créer des motifs dynamiques et esthétiques qui pourraient donc bénéficier des méthodes de visualisation et d'interactions utilisées dans le monde de l'informatique graphique. Cela permettrait d'ouvrir ce domaine dans un premier temps à la communauté scientifique puis au grand public en rendant l'expérimentation de ces systèmes intuitive et satisfaisante. Toutefois, comme mentionné plus haut, étudier un système complexe (comme un automate cellulaire) nécessite de pouvoir le simuler, afin d'étudier son comportement dans différentes conditions. Pour cela, il est important de fournir des outils efficaces pour pouvoir visualiser et interagir avec ces systèmes à grande échelle. L'objectif de ce projet de fin d'étude est de proposer un tel outil logiciel.



(a) Instance de Lenia.



(b) Instance de Flow-Lenia.

Figure 3 – Exemple de simulations et de motifs d'automates cellulaires. (a) Les différents états de chaque cellule sont représentés par des canaux RGB. Image issue de [9]. (b) Les règles de mise à jour locales sont représentés par une couleur distincte. Image issue de [18]

Sujet

L'objectif de ce projet est donc d'implémenter la première preuve de concept d'un logiciel rassemblant toutes les idées et tendances actuelles du domaine, avec trois axes principaux :

— Permettre la simulation rapide de différents systèmes complexes par accélération graphique,
— Permettre l'interaction avec les systèmes simulés et configurer son aspect visuel en temps réel dans une interface web,
— Fournir des outils sur cette interface afin de permettre l'évaluation, l'interaction directe avec le système complexe, le partage de simulations particulières avec ses paramètres ou des motifs obtenus entre différents utilisateurs, puis à terme, permettre le branchement ou la fusion de ces motifs.

1 État de l'art

1.1 Systèmes complexes

Un premier sous-ensemble des systèmes complexes sur lesquels nous allons nous appuyer suit une approche matricielle. Le système le plus connu de ce sous-ensemble est le Jeu de la Vie de John Conway (cf. figure 1b et figure 2) [12]. Il est composé d'une grille de cellules pour lesquelles un état (0 ou 1) est associé. Pour passer d'un pas de temps à un autre, une convolution est appliquée à la grille. Une cellule à 0 passe à 1 si elle a exactement 3 voisines à 1. Et une cellule à 1 reste à 1 si elle a 2 ou 3 voisines à 1. Ces deux règles sont configurables, bien que la diversité des motifs émergeant est maximisé avec les paramètres cités.

Depuis, de nombreux systèmes ont été conçus dans l'optique de généraliser le Jeu de la Vie à des espaces continus. Parmi-eux, on peut retrouver Lenia (cf. 3a) [8, 6]. Il s'agit d'une classe d'automate cellulaire, dont les états correspondent à des vecteurs. Ces états passent de binaire pour le jeu de la vie à continu (nombre flottant) pour Lenia et chaque cellule peut avoir un ou plusieurs états dans différents canaux. Cette généralisation s'applique aussi aux règles de mise à jour, prenant en compte un nombre arbitraire de convolutions, dans un voisinage étendu de la cellule.

Une extension de Lenia, Flow-Lenia (cf. 3b)[18], grâce à l'ajout d'un mécanisme de conservation de masse, le système a la possibilité d'avoir des règles de mise à jour locales. Ce qui implique qu'au-delà des paramètres initiaux et des paramètres des filtres de convolutions, le système est muni d'un ensemble de paramètre modifiant les règles de mise à jour uniquement dans un sous-ensemble de la grille.

Nous avons également souhaité nous intéresser aux systèmes particulaires dans lesquels des particules évoluent dans un espace continue. Parmi ces systèmes, le plus emblématique est probablement celui des boids de Reynod. Proposés en 1987 par W. Reynolds [19], les boids sont des particules évoluant en groupe en suivant 3 règles simples : alignement, évitement et cohésion. L'application de ces simples règles à un ensemble quelconque de particules permet de faire émerger des comportements qui rappellent des phénomènes de groupe observables dans la nature tel que le déplacement de nuées d'oiseaux ou de troupeaux d'animaux terrestres.

1.2 Visualisation

Une démonstration de Lenia est proposée par son créateur [7], la visualisation est créée directement au sein d'élément HTML *canvas*. Cette démonstration offre à l'utilisateur la possibilité de visualiser les états en sortie du système, mais aussi différents objets et étapes intermédiaires permettant le calcul de ces états. L'ensemble des paramètres du système est modifiable par l'utilisateur. Il y a aussi la possibilité d'instancier directement des *créatures* à partir d'un dossier les regroupant par taxonomie. La figure 4 est une capture de cette démonstration web.

En revanche, là où le système complexe est grandement personnalisable, la visualisation reste dépendante du système, il n'y a que quelques options pour modifier la palette de couleur et l'orientation des créatures affichées. L'interaction avec la simulation ne se fait qu'au travers de boutons, il n'est pas possible d'interagir directement avec la visualisation. De plus, la plupart des visualisations existantes qui permettent à l'utilisateur d'interagir avec le système se contentent actuellement d'un espace à 2 dimensions.

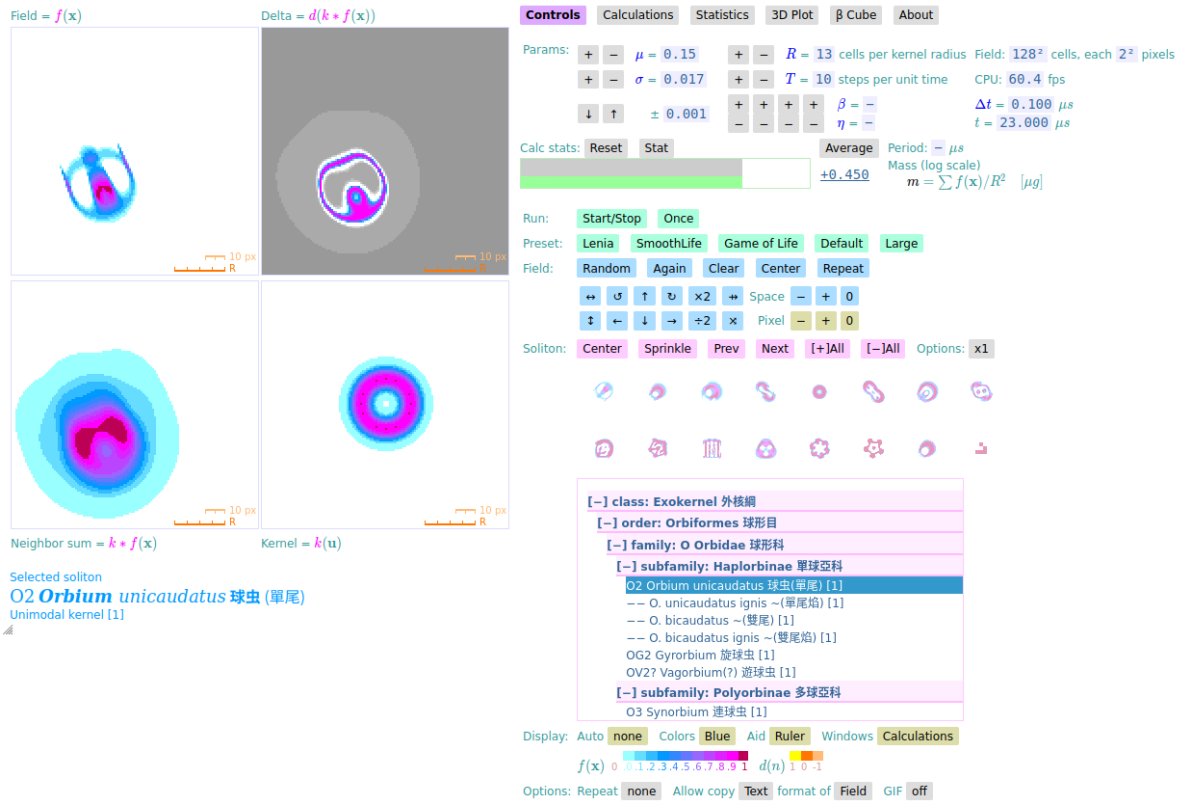


Figure 4 – Capture de la démonstration web de Lenia [7]. Un grand nombre de paramètres de la simulation est apparent et configurable. La personnalisation de la visualisation se fait avec les boutons en bas de la fenêtre. L'utilisateur peut instancier un motif à partir d'un répertoire.

Dans un aspect plus global et regroupant plusieurs systèmes, il y a le site internet Complexity Explorables [5]. Ce site répertorie un ensemble de systèmes complexes, avec une visualisation et une explication associée à chacun d'entre eux. Comme son nom l'indique, ce site a pour objectif de permettre à l'utilisateur une exploration des différents systèmes et de leur espace de paramètre associé. Chaque simulation et visualisation est implémentée en utilisant la bibliothèque `D3` [4], une bibliothèque `JavaScript` de visualisation dynamique de données.

Chaque simulation est associée à une page du site internet avec le même format (la figure 5 est une capture de l'une de ces pages). En revanche, la simulation et la visualisation sont couplées, ce qui n'offre aucune possibilité de personnalisation de la visualisation. Les interactions avec le système passent par des boutons, modifiant les différents paramètres associés.

`SwissGL` [17] et `gpu-io` [13] sont deux bibliothèques permettant la manipulation de `WebGL2` [14]. L'objectif de ces deux bibliothèques est d'offrir au développeur une interface lui permettant l'exécution de systèmes complexes sur GPU, sans avoir à manipuler les fonctions bas-niveau de `WebGL2`. Pour ce faire, le code du système est rédigé dans un format propre à la bibliothèque, puis il est exécuté au sein des *vertex shaders* et *fragment shaders*. `WebGL2` ne possède pas de *compute shader* comparé à `WebGPU` [16], l'utilisation de ces bibliothèques permet l'obtention d'un comportement similaire à un *compute shader* sans avoir à manipuler les différentes astuces d'implémentations. De plus, la visualisation des systèmes peut aussi se faire avec `WebGL2`, sans avoir à transformer les données avec un module intermédiaire.

La démonstration de `SwissGL` (cf. figure 6) possède une interface très minimaliste. Elle offre la possibilité de choisir un modèle à visualiser, ainsi qu'un ensemble de paramètres associés à manipuler. Les visualisations sont en 2D ou en 3D. Pour certaines, il y a aussi la possibilité de déplacer la caméra et d'interagir directement

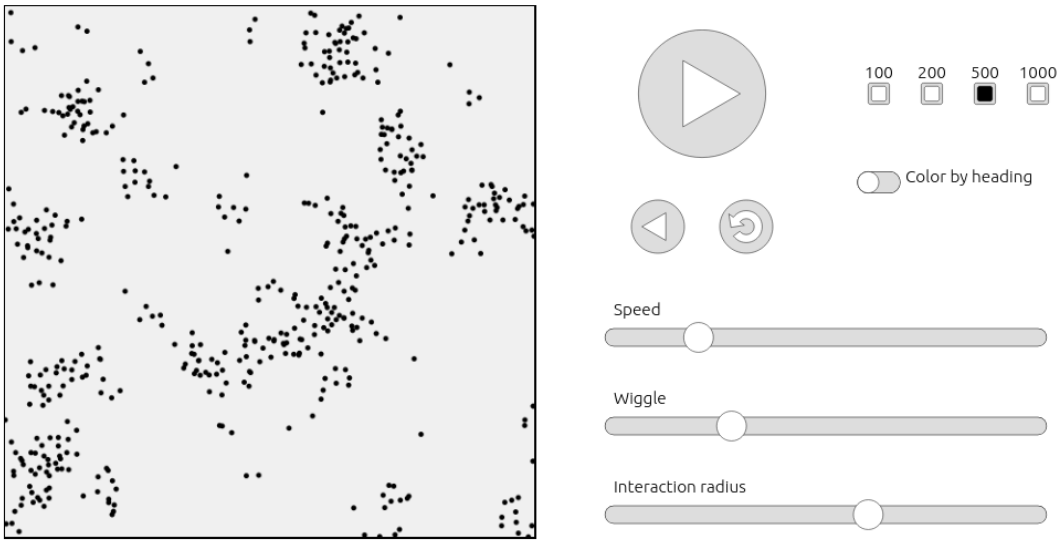


Figure 5 – Capture d'une visualisation issue du site web Complexity Explorable. *L'affichage est simple, l'utilisateur a la possibilité d'interagir avec les boutons à droite pour modifier le comportement de la simulation. En revanche, il n'est pas possible d'interagir avec la visualisation à gauche.[5]*

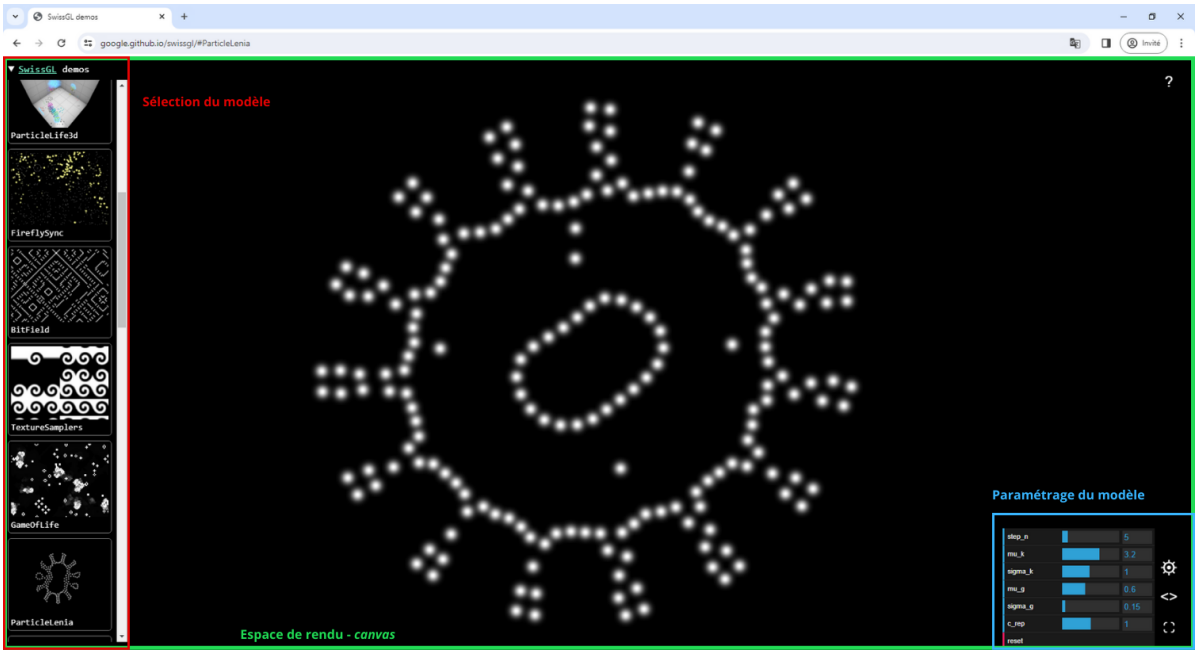


Figure 6 – Capture d'écran annotée de la démonstration de SwissGL. L'espace de rendu, représenté par un élément HTML canvas, *occupe toute la page ; le menu déroulant, positionné au-dessus du* canvas *à gauche, comprend les différents modèles disponibles ; le paramétrage du système se fait par des curseurs, eux aussi positionnés au-dessus du* canvas *en bas à droite.*

avec la simulation au travers de la visualisation. Cependant, les caractéristiques visuelles de la représentation ne sont pas personnalisable.

2 Récits utilisateurs

Récit utilisateur 1 : Visualisation d'un système complexe L'utilisateur doit avoir le choix entre plusieurs systèmes complexes à visualiser. Une fois ce choix effectué, il doit pouvoir lancer la simulation, la stopper, la réinitialiser et augmenter le nombre de pas de temps affichés par seconde.

Récit utilisateur 2 : Initialisation paramétrée du système complexe L'utilisateur doit pouvoir paramétrer comme il le souhaite les paramètres d'initialisation avant de lancer une simulation. Ces paramètres diffèrent en fonction de la simulation choisie. Il peut par exemple choisir d'initialiser la simulation avec des états définis aléatoirement ou non.

Récit utilisateur 3 : Modification des propriétés de visualisation L'utilisateur doit pouvoir configurer les propriétés de la visualisation, que ce soit des propriétés statiques ou des propriétés dépendantes de la sortie du système. Cette configuration est interactive, la modification est directement perceptible au sein de la visualisation.

Récit utilisateur 4 : Interaction avec la simulation L'utilisateur doit pouvoir interagir avec les éléments visualisés à l'aide de son curseur, que le système complexe soit en train d'être simulé ou non. Cette interaction a pour effet d'altérer la configuration courante du système.

Récit utilisateur 5 : Ergonomie de l'application L'utilisateur disposera d'une interface ergonomique pour interagir avec le système complexe dans tous les aspects cités dans les précédents récits utilisateurs. L'interface devra permettre des interactions sur les aspects visuels, sur les propriétés du système, sur le déroulement de la simulation et le choix du système à simuler. Cette interface doit tout de même être simple pour mettre en avant l'espace de simulation central.

3 Choix logiciels

3.1 Framework et architecture générale

Afin de réaliser au mieux les objectifs de ce projet, nous avons opté pour un logiciel avec une architecture client-serveur dans laquelle le coté serveur sera utilisé pour réaliser les simulations des systèmes complexes et le coté client utilisé pour visualiser ces simulations et pouvoir interagir avec elles. Nous pensons que c'est la meilleure alternative pour visualiser des systèmes complexes dans une interface web. Cette architecture rend le logiciel très modulaire des deux cotés pour des améliorations futures : Du coté serveur cela facilite l'implémentation de nouveaux systèmes complexes, et du coté client l'implémentation de nouveaux paramètres de visualisation ou même de visualisations complètement différentes. Cela facilitera donc, pour les utilisateurs, l'exploration de nouvelles simulations et l'émergence de nouveaux motifs non anticipés. De plus, les simulations existantes étant développées en Python, nous avons orienté notre choix vers le Framework open source *Django* [11] qui permet la mise en place rapide d'une application client-serveur avec le côté serveur développé en Python. Ce choix a été facilité par le fait que nous avions déjà des connaissances sur ce framework ce qui a permis d'accélérer la mise en place de l'application compte tenu du temps limité que nous avions pour réaliser ce projet qui était de deux mois. De plus, une bibliothèque additionnelle est disponible afin d'ajouter simplement la gestion de WebSockets à Django. Ceci nous a conforté dans notre choix car nous avons besoin de communiquer beaucoup de données de manières bi-directionnelle et asynchrone entre le serveur et le client, que ce soit pour l'envoi de données sortantes de la simulation vers le client, ou dans l'autre sens communiquer les interactions de l'utilisateur au serveur.

3.2 API pour le rendu graphique

Du côté du client, l'enjeu est de pouvoir effectuer un rendu graphique flexible tout en étant agnostique au type de système visualisé. Plusieurs API ont été considérés, notamment `three.js` [23] et `babylon.js` [1], des bibliothèque interfaçant les différentes commandes `WebGL2` dans l'optique d'en faire une utilisation

efficace dans la grande majorité des cas. Cependant, leur modularité a en contrepartie un surcoût en temps d'exécution. Étant donné que la partie simulation va elle aussi consommer des ressources en temps, nous voulons que le rendu côté client se fasse de la manière la plus optimisée possible. C'est pour cela que nous avons opté pour une solution plus bas-niveau. Parmi les API bas-niveaux, on peut retrouver `WebGPU` [16], une API visant à porter les capacités des GPU modernes sur le web, notamment avec la capacité d'utiliser des *compute shaders* outre les possibilités de rendu graphique classique. Cependant, cette technologie est encore récente et n'est disponible que sur une petite partie des plateformes / navigateurs. Notre choix s'est donc porté sur `WebGL2`, fonctionnant sur la plupart des navigateurs. Nous l'avons favorisé à `WebGL` notamment pour sa capacité à prendre en charge du rendu par instance (une fonctionnalité importante sur laquelle nous revenons dans la partie 5.4.3).

Par souci de lisibilité et de clarté du code, nous avons choisi d'écrire nos scripts en `TypeScript` pour ensuite les transpiler en `JavaScript`. Outre la capacité de typer les variables, ce langage offre des fonctionnalités telles que les énumérateurs et la modification d'accès au sein des classes. Ces fonctionnalités nous ont semblé nécessaires pour la mise en place de la hiérarchisation du projet.

Pour les manipulations mathématiques, plus précisément pour les vecteurs et les matrices, nous avons utilisé la bibliothèque `glMatrix` [15]. Le *parsing* des fichiers `obj` est effectué à l'aide de la bibliothèque `obj-file-parser-ts` [24].

4 Architecture

L'architecture côté serveur peut être divisée en cinq acteurs comme montré sur la fig 7. En bleu on retrouve le module Viewer du serveur Django, qui reçoit les requêtes HTTP du navigateur et génère les réponses en fonction des informations stockées dans le modèle (en vert) et le deuxième module Django Simulation (en violet). En vert on retrouve l'acteur responsable du modèle en lien direct avec la base de donnée. Il dispose d'un diagramme de classe des différentes classes python qui sont présentes dans la base de données. Le module Django Simulation est en charge des simulations, permettant d'accéder aux différents paramètres des systèmes complexes, et d'interagir avec eux. En rouge ce sont tous les fichiers HTML qui permettent de générer la page web et les fichiers statiques comprenant la mise en page (CSS) et les différents script typescript et javascript coté client dont l'architecture a été décrite précédemment. Finalement en orange on retrouve le navigateur, ou client, qui interagit seulement avec le module Viewer du serveur.

Le client (figure 8) est aussi segmenté en cinq sous-modules. Le module principal est celui par lequel le rendu est effectué, il s'agit du cœur de la partie client. Les interactions de l'utilisateur sont capturées dans le module *Interface*, les directives sont ensuite redirigées vers le module affecté par l'interaction. Il peut s'agir du module *Transformers* si l'interaction modifie l'apparence de la visualisation, ou le module *Selection* si elle modifie les états du système. Si le client a le besoin d'envoyer des données au serveur, le message doit passer par le module *Transmission*. Ce module est sur un thread séparé, il s'agit de la porte d'entrée des communications avec le serveur. Les échanges entre le module *Transmission* et le thread principal se font au travers de la classe `Viewer` du module *Rendering*, par le biais de messages ou de tampons de données formatés (*TransformableValues*).

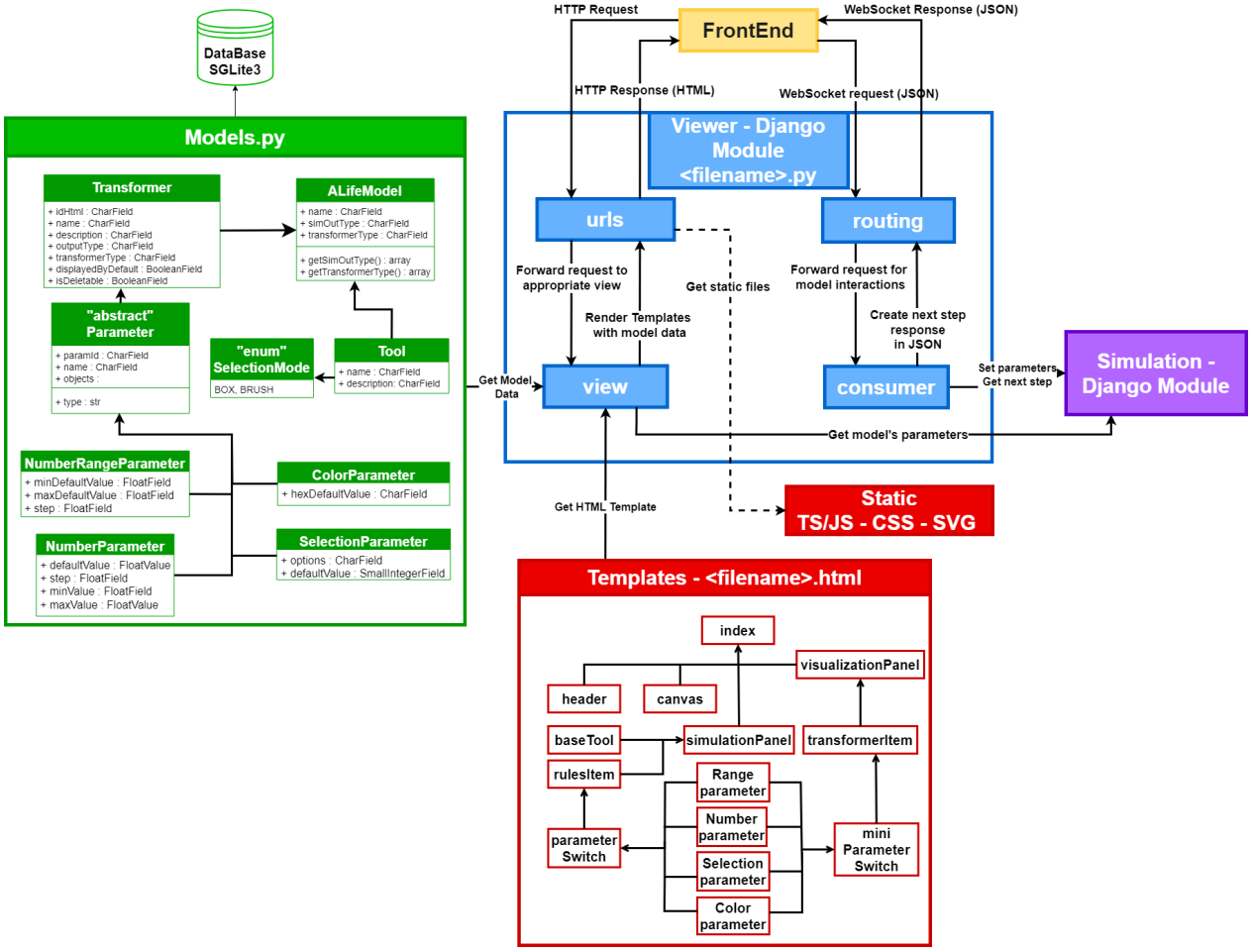


Figure 7 – Diagramme représentant les différents modules, fichiers et interactions du serveur Django

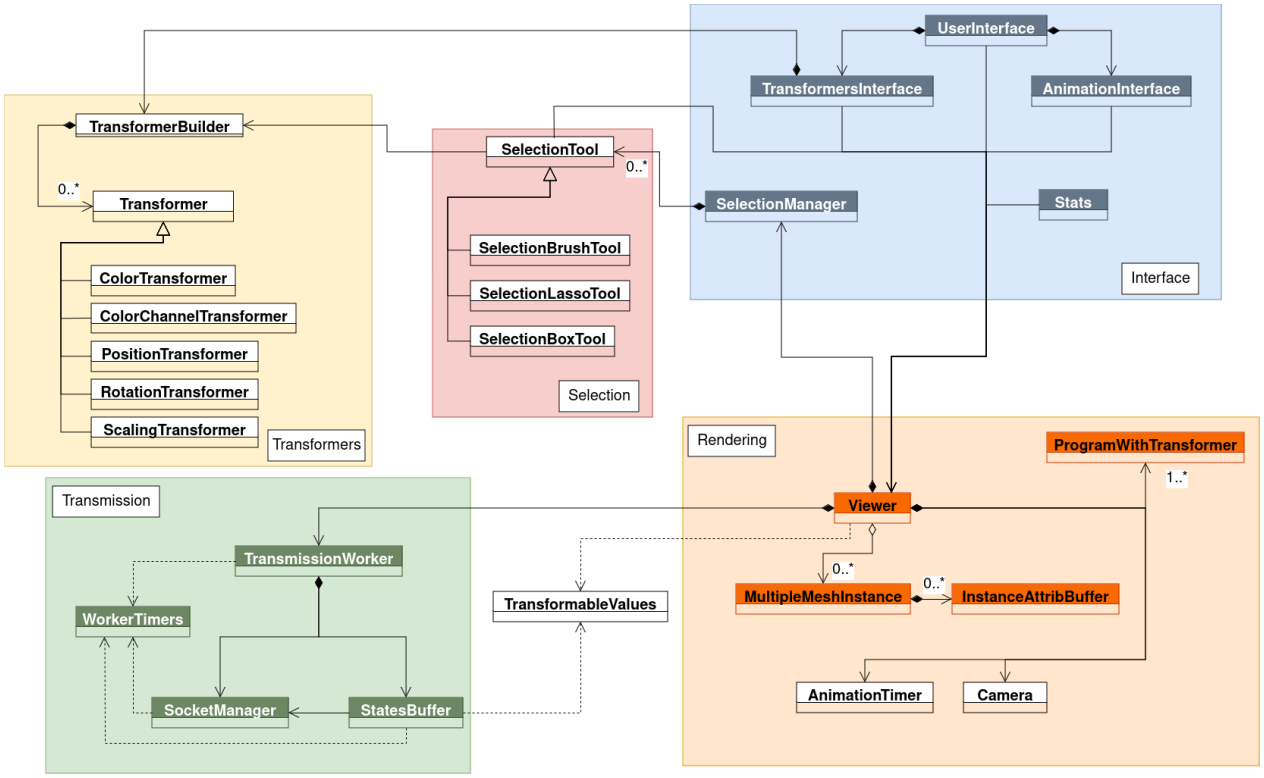


FIGURE 8 – Diagramme de classe de la partie visualisation du client. Les classes en bleu correspondent aux classes capturant les interactions de l'utilisateur au travers des éléments HTML. Les classes en orange correspondent aux classes ayant une interaction avec le GPU au travers de `WebGL2`*. Les classes en vert sont les classes fonctionnant sur un* WebWorker *distinct, c'est par ces classes que la communication avec le serveur est effectuée.*

5 Réalisation

5.1 Systèmes complexes

5.1.1 Paquetage, dépendances et responsabilités

Un des objectifs du projet était de simuler les systèmes complexes sur le serveur afin que ces derniers puissent être visualisés dans le navigateur. Nous avons pour cela choisi de développer un module python indépendant qui pourrait aussi bien être utilisé par notre serveur Django que par un autre programme. Cette contrainte que nous nous sommes fixée permet de mieux répartir les responsabilités logicielles au sein du projet, en s'assurant notamment que seul ce module soit chargé d'exécuter les simulations, et cela sans prendre en compte la manière dont sont transmises ou utilisées les données des modèles. Un autre avantage de cette approche est de faciliter le travail en parallèle au sein de l'équipe, en permettant à chacun de travailler sur un sous-ensemble du projet différent. Enfin, le développement d'un module python indépendant permettrait de pouvoir réutiliser ce module pour d'autres projets en lien avec les systèmes complexes.

5.1.2 Modularité

Un des principaux objectifs fixés par nos clients était la modularité de l'implémentation, à savoir la possibilité d'ajouter facilement de nouveaux modèles de systèmes complexes, mais aussi de pouvoir récupérer les états de ces systèmes dans un format de données connu et unique, qu'importe le modèle simulé. Pour garantir la possibilité d'ajouter facilement de nouveaux modèles, nous avons choisi d'articuler notre code autour de la classe abstraite `Simulation` qui définit les propriétés et comportements propres à tous les modèles et qui déclare également les méthodes que toutes les simulations doivent implémenter. Ainsi, pour ajouter un nouveau modèle au module, il suffit à un développeur de déclarer une nouvelle classe héritant de la classe abstraite `Simulation` et de définir les attributs et méthodes suivants :

- Un attribut `initialization_parameters` déclarant les paramètres nécessaires à l'initialisation de la simulation. L'utilisateur final pourra les modifier avant de lancer la simulation (par exemple la taille de la grille dans le cas d'un système matriciel),
- Un attribut `default_rules` déclarant des paramètres utiles à chaque pas de la simulation. L'utilisateur pourra les modifier avant ou pendant que la simulation s'effectue,
- Une méthode `initSimulation` implémentant les traitements à réaliser avant de lancer la simulation (initialisation des états et des paramètres par exemple),
- Une méthode `step` permettant d'effectuer un pas de simulation et de mettre à jour l'état de la simulation. C'est en particulier cette méthode qui diffère grandement d'un système complexe à un autre,
- Une méthode `set_current_state_from_array` permettant d'actualiser l'état de la simulation à partir d'un tableau de données. Cette méthode permet notamment de définir l'état de la simulation à partir de données extérieures au module.

Les détails pratiques concernant la manière d'ajouter un nouveaux système complexe au module sont détaillés dans la documentation en annexe A.

5.1.3 Gestion des différents types de modèle

Dans le cadre de ce projet, nous nous sommes intéressés à deux types de modèles de systèmes complexes : les modèles dits matriciels (ou *grid-based*), en particulier les automates cellulaires tels que le jeu de la vie de Conway [12] ou les systèmes de type Lenia [8] et les modèles dits particulaires tel que les Boids de Reynold [19]. Notre première intuition était de considérer les systèmes matriciels comme des cas particuliers de systèmes particulaires pour lesquels la position des particules serait discrète et non continue. Les différentes implémentations existantes permettant de simuler ces systèmes matriciels utilisent une grille pour représenter l'état de la simulation. Il nous fallait donc, après chaque pas de simulation, transformer les données de la grille en une liste de particules. Cette approche permettait effectivement de traiter indépendamment les systèmes matriciels et les systèmes particulaires, mais la conversion de l'état du système de grilles vers particules représentait un trop gros coup de calcul pour être utilisé avec des grilles pouvant atteindre plusieurs dizaines

de milliers d'éléments (une grille de 100x100 demandait d'instancier 10000 particules par exemple ce qui affectait grandement les performances du module). Face à ce problème nous avons choisis de déclarer deux sous-classes de la classe `State` : `ParticuleState` et `GridState` (voir annexe B), chargées respectivement de représenter l'état des systèmes particulaires et matriciels. Cependant le reste du programme, et en particulier la visualisation, est agnostique de cette distinction comme indiqué dans la section 5.1.1.

5.1.4 Systèmes complexes disponibles

Ce projet ne s'intéresse pas tant à l'implémentation de systèmes complexes qu'à leur visualisation. Nous avons donc choisi d'adapter des implémentations de système complexes existantes à l'architecture de notre module de simulation afin d'uniformiser les entrées et sorties de ces dernières et de pouvoir les visualiser. Une des principales contrainte des implémentations existantes est l'utilisation de la bibliothèque JAX [20] qui permet de grandement accélérer certains traitements numériques. Guidés par nos encadrants, nous avons choisi d'inclure au module de simulation les systèmes suivants :

— **Le Jeu de la vie** : Le jeu de la vie de Conway [12] est un système matriciel des plus simples que nous avons implémenté en premier afin de pouvoir valider notre conception et réaliser les premières visualisations. Bien que ce système soit relativement peu coûteux à simuler, la simulation avec des grilles de grande taille nous ont permises d'identifier et de corriger assez tôt des "goulots d'étranglement" - traitements demandant un temps d'exécution supérieur aux autres et ralentissant l'ensemble de la simulation - dans le module de simulation. Un exemple d'état du jeu de la vie est visible en figure 9.

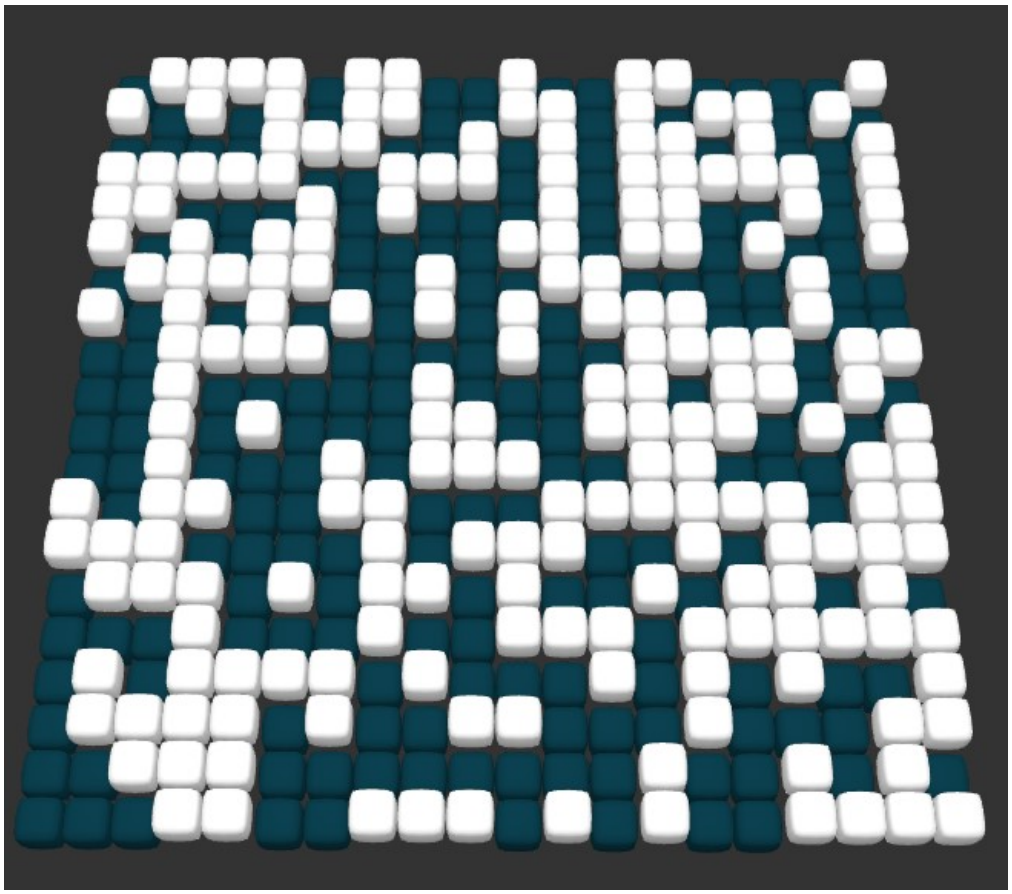


Figure 9 – Un état du jeu de la vie visualisé à l'aide de notre programme

— **Lenia** : Lenia [8] est un système complexe matriciel plus complexe que le jeu de la vie car chaque cellule peut prendre une ou plusieurs valeurs entre 0 et 1. Pour intégrer Lenia au module de simulation nous nous sommes appuyés sur une implémentation python fournie par nos encadrants utilisant la

bibliothèque JAX. Notre module était déjà en mesure de gérer les valeurs d'état continues nécessaires au fonctionnement de Lenia donc il n'y a pas eu de grande modification à faire sur cet aspect. Concernant le fait que chaque cellule puisse prendre simultanément plusieurs valeurs (version multi-canaux ou *multi-channel*), cela a demandé de mettre en place une représentation plus générique des grilles d'état dans la classe `GridState` en passant de grilles 2D de forme $(largeur, hauteur)$ à des grilles 3D de forme $(largeur, hauteur, nombre\ de\ canaux)$. Un exemple d'état de Lenia est visible en figure 10.

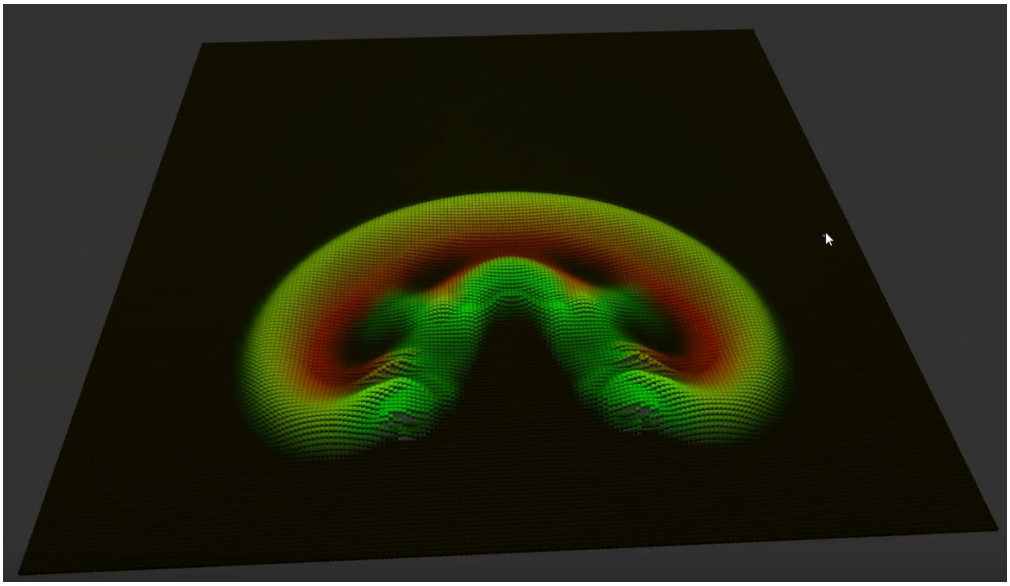


Figure 10 – Un état de Lenia visualisé à l'aide de notre programme

— **Boids** : Les boids de Reynolds [19] sont des particules dont le déplacements et régi par trois règles relativement simples : l'alignement, l'évitement et la cohésion. Ce système se différencie particulièrement des deux précédents du fait qu'il s'agit d'un système particulaire. Pour intégrer la simulation des boids au module, nous nous sommes basés sur une implémentation disponible sous forme de notebook Jupyter [2] sur le github de la bibliothèque Jax M.D [3], une bibliothèque qui offre des outils de simulation physique utilisant JAX [20]. Les particules se déplacent dans un espace continu, il n'est donc plus pertinent de représenter l'état par une grille. Nous avons donc introduit la classe `ParticleState`, une nouvelle sous-classe de la classe `State` qui représente un état de simulation par un ensemble d'instances de la classe `Particle`. Ce nouveau type de système nous a permis de prouver la flexibilité du module de simulation qui peut maintenant aussi bien simuler les systèmes matriciels que les systèmes particulaires. Un exemple d'état de simulation de boids est visible en figure 11.

2. Les notebooks Jupyter permettent de combiner dans un même fichier du code python avec du texte ou des images et sont souvent utilisés à des fins pédagogiques.

3. https://github.com/jax-md/jax-md

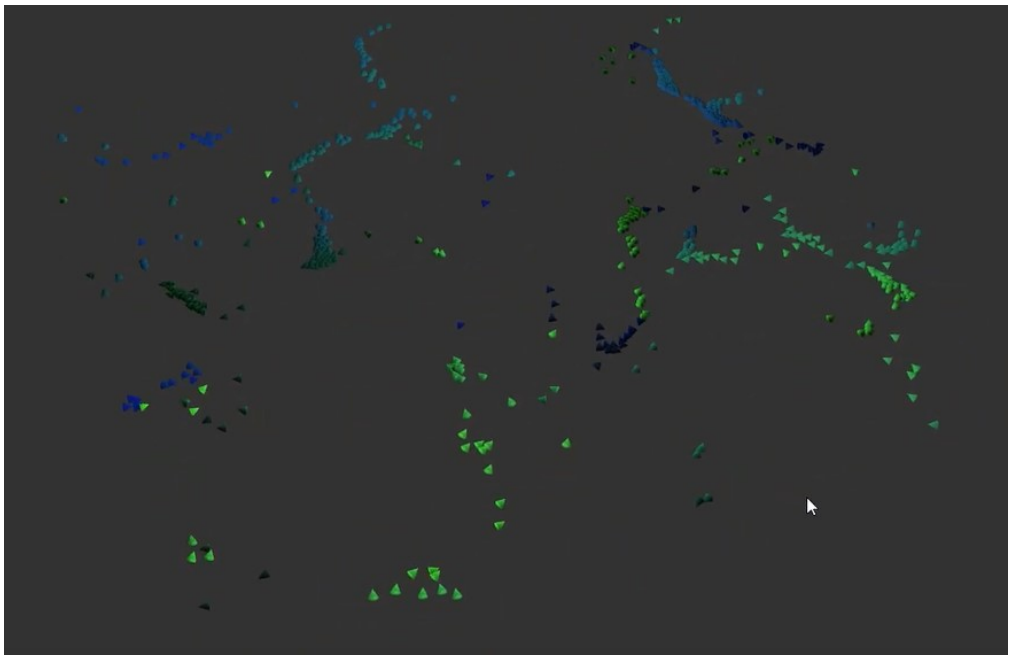


Figure 11 – Un état de simulation de boids visualisé à l'aide de notre programme

5.1.5 Exposition des paramètres de simulation

Le fait que l'utilisateur puisse modifier certains paramètres de la simulation était un besoin fonctionnel essentiel de notre projet. La mise en place de cette fonctionnalité nécessitait de définir une structure de données, commune à toutes les implémentations de systèmes complexes, qui puisse être utilisée en dehors du module de simulation (pour l'affichage dans le navigateur par exemple). Nous avons pour cela défini une super-classe `Param` et différentes sous-classes permettant de représenter des paramètres de différents types (voir annexe B). Cette approche permet d'exposer des paramètres de différents types primitifs de python comme des nombres flottants ou entiers mais aussi des paramètres plus complexes comme des intervalles de valeurs. De plus ces classes ne se contentent pas de stocker la valeur et le nom d'un paramètre mais permettent aussi de définir des contraintes sur les valeurs autorisées (minimum et maximum) et de suggérer un pas d'incrément pour la modification de la valeur. Ces différents paramètres peuvent donc être récupérés, affichés et mis à jour à l'extérieur du module de simulation tout en étant pris en compte lors de la simulation.

Aussi, comme évoqué dans la section 5.1.2, nous avons fais le choix de définir deux listes distinctes de paramètres pour chaque système complexe : une première contenant les paramètres relatifs au domaine spatial, temporel et dimensionnel, utilisés pour créer l'état de départ de la simulation et une seconde contenant des paramètres relatifs aux règles de la simulation, utilisés à chaque pas de la simulation.

Une fois encore, cette approche permet au reste du programme de pouvoir interagir avec les paramètres exposés sans avoir connaissance du système complexe auquel ils correspondent.

5.1.6 Réactions aux interactions utilisateur

Une fois les premières simulations en place, nous avons discuté avec les encadrants du projet des améliorations à mettre en place en priorité. L'une était la possibilité pour l'utilisateur d'interagir avec la simulation. Pour cela nous avons choisi de donner la possibilité aux développeurs de définir pour chaque modèle la manière de traiter les interactions utilisateur. Ces interactions, représentées par des instances de la classe `Interaction`, doivent notamment définir une méthode qui, étant donné un masque de valeurs entre 0 et 1, doit pouvoir mettre à jour une liste d'états de simulation. Par exemple pour le jeu de la vie nous avons implémenté une interaction qui change l'état des cellules correspondantes au masque de sélection à 1 (en vie). L'utilisation d'un masque de ce type rend la simulation complètement agnostique de la manière dont est effectuée l'interaction et offre la possibilité d'envisager d'autres types d'interactions (voir section 7).

5.2 Transmission des données

Du fait de l'architecture client-serveur avec du coté client la visualisation et du coté serveur les simulations des systèmes complexes, nous devons établir un lien entre les deux. Dans le sens serveur vers client pour transmettre les données de la simulation à la visualisation, et dans l'autre sens pour transmettre les modifications effectuées par l'utilisateur à la simulation. Puisque nous devons avoir une communication bi-directionnelle et besoin d'envoyer de nombreuses données rapidement, nous avons choisi d'utiliser une communication WebSocket au lieu d'une communication HTTP classique. Django possède une bibliothèque additionnelle permettant la mise en place de cette communication nommée "Django Channels". Avec cette bibliothèque, notre serveur Django peut être déployé sous ASGI (Asynchronous Server Gateway Interface), un standard python pour les serveurs et applications asynchrones.

5.2.1 Requêtes WebSocket

Du coté serveur, avec Django Channels nous avons maintenant un gestionnaire de route spécialisé pour les communications WebSocket. Ainsi lorsque Channels va recevoir une requête de communication *WebSocket* (requête venant d'une URL commençant par `"ws://"`) il va consulter ce gestionnaire pour lui attribuer un consommateur adéquat.

5.2.2 Consumer

Le `consumer` est la pièce maitresse de la gestion des *WebSocket*. Il est chargé de gérer la communication bi-directionnelle entre le client et la simulation. Ce consommateur possède un attribut `"simulation"`, qui va permettre d'instancier nos différentes simulations, ainsi qu'un paramètre `"init_parameters"` contenant les paramètres d'initialisation de la simulation tels que sa taille ou un état de départ aléatoire par exemple.

Un `consumer` possède nativement quatre fonctions importantes :

- **connect** : fonction permettant d'accepter une demande de connexion de la part d'un WebSocket et de lui notifier,
- **disconnect** : fonction permettant de cloturer une connection avec un WebSocket,
- **receive** : fonction appelée lorsque le consommateur reçoit un message du WebSocket par le canal de communication,
- **send** : fonction utilisée pour envoyer un message au WebSocket par le canal de communication.

Au début nous pensions envoyer les données en continu à travers le consumer lorsque la simulation était lancée depuis l'interface mais cela ne s'est pas révélé efficace en terme de performance (cf 6) et relativement limité en terme d'utilisation puisque nous avions des soucis de synchronisation lors des interactions depuis l'interface. Par exemple si l'utilisateur mettait en pause la simulations, la pause n'était pas instantanée et la simulation continuait un petit moment avant de vraiment se mettre en pause. Pour cela nous avons donc décidé de changer la philosophie d'interaction entre le client et le serveur. Nous avons mis en place un système de requête de chaque état de la part du client et le serveur délivre un pas de simulation à chaque fois qu'il reçoit une requête de la part du client.

Pour cela, nous avons rajouté six fonctions à ce consommateur pour gérer les simulations en fonctions des messages envoyés par le client :

- **initNewSimulation** : fonction permettant d'initialiser une nouvelle simulation avec ses paramètres d'initialisation. Le choix de cette simulation est spécifiée par un identifiant en paramètre,
- **resetSimulation** : fonction permettant de réinitialiser la simulation actuellement en place, sans changer les paramètres d'initialisation,
- **sendOneStep** : fonction permettant d'envoyer un pas de simulation vers le client,
- **updateInitParameters** : fonction permettant de changer les paramètres d'initialisation, cette fonction prend en paramètre un fichier Json avec en clé le nom du paramètre à modifier et sa nouvelle valeur associée,
- **updateRule** : fonction permettant de modifier les règles spécifiques à la simulation actuelle,
- **applyInteraction** : fonction permettant d'appliquer une modification à la simulation suite à une interaction de l'utilisateur.

5.3 Interface Utilisateur

Avant de commencer la réalisation de l'interface nous avons choisit de réaliser une maquette afin de vérifier avec nos clients si elle correspondait à leurs attentes. Pour ce faire nous nous sommes inspirés du site de démonstration de SwissGl (cf. fig 6). Comme discuté dans l'état de l'art, ce site est minimaliste et met en avant l'espace de rendu 3D. Nous avons choisi de ne faire qu'une seule page plutôt qu'une arborescence de pages afin de simplifier l'expérience utilisateur.

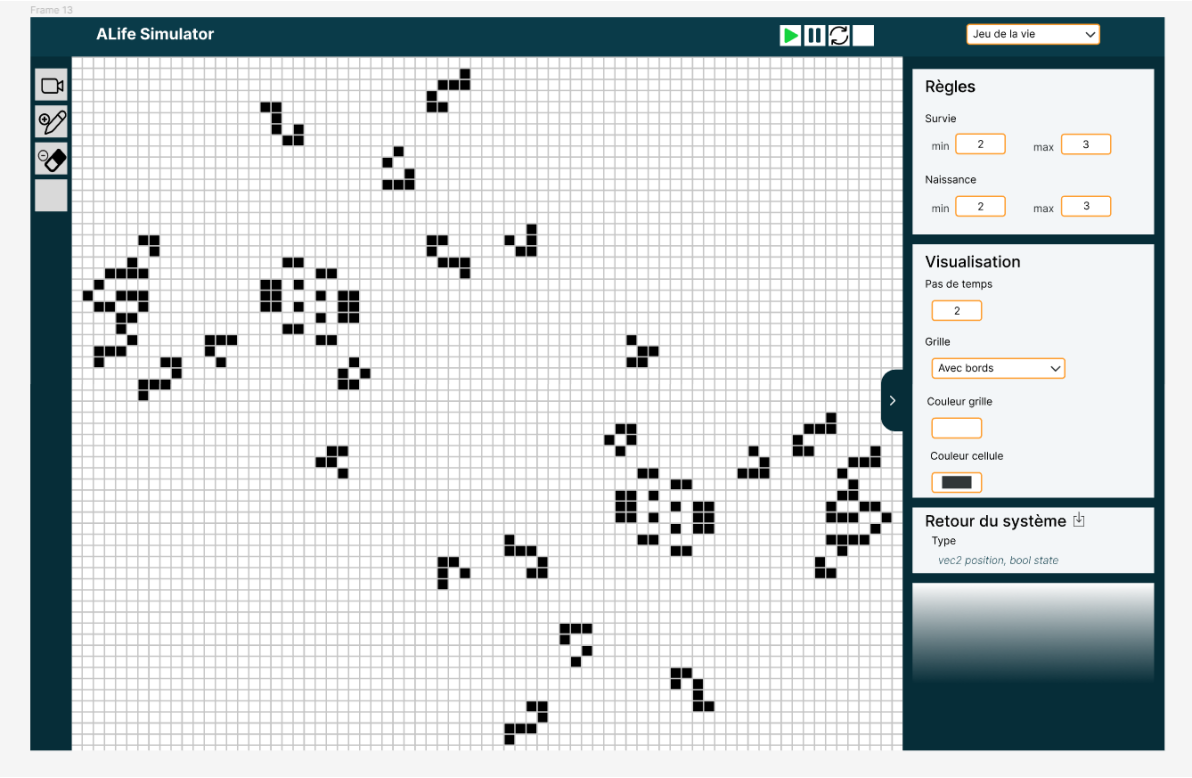


Figure 12 – Maquette du site web avant la réalisation

L'interface finale a nettement évolué par rapport à la maquette du fait des différents échanges que nous avons pu avoir avec nos clients. Tous d'abord le panneau de configuration à droite de la maquette, qui regroupait tous les paramètres (simulation et visualisation), a été divisé en deux panneaux. Le but était d'améliorer la lisibilité de la page avec chaque panneau ayant sa responsabilité claire. Nous avons aussi décidé de retirer la section "Retour du systeme", qui aurait permis de voir les différents attributs de sortie du modèle simulé et leurs types (*vec2 position* et *bool state* pour GOL dans la maquette). Cette information sera intégrée directement dans le paramétrage de la visualisation à l'aide des `Transformers`.

Le template principal `index.html` est responsable du chargement des scripts JavaScript, du CSS et des quatre principaux éléments de l'interface suivant :

— **Header** : Le bandeaux supérieur comprenant les informations de performance de l'application et des boutons pour gérer le déroulement de la simulation tel que : jouer, stopper, réinitialiser ou gérer la vitesse de simulation.

— **Simulation Panel** : Le panneau de configuration de la simulation pour sélectionner le modèle à simuler, contrôler ses paramètres d'initialisation et ses règles. Ce panneau comprend aussi les outils de sélection pour interagir avec le système.

— **Canvas** : L'espace de rendu 3D ou se dérouleront les simulations des systèmes complexes.

— **Visualization Panel** : Le panneau de configuration de la visualisation permet de gérer la fonction d'animation pour passer d'un état à un autre. C'est aussi dans ce panneau qu'il est possible de de définir comment seront associés les attributs de la simulation avec les attributs de visualisation à l'aide de `Transformer`.

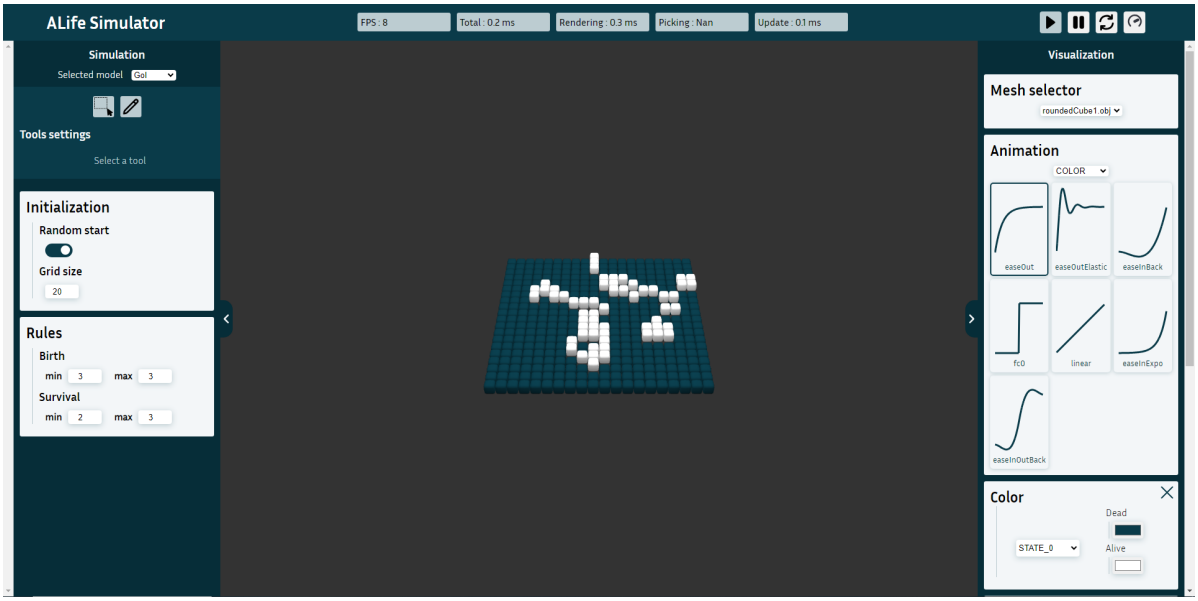


Figure 13 – Capture d'écran de la version finale du site

Les quatre éléments cités précédemment possèdent chacun leur fichier HTML afin de faciliter la maintenance du code et sa lisibilité. De plus tous les templates sont écrits avec le langage de template Django. Celui-ci permet d'intégrer de la logique (if..else, for etc..) directement dans le fichier HTML et ainsi modifier le rendu en fonction du contexte fourni par `Views.py`. Le comportements des différents éléments (clic, changement de valeur etc..) est géré, sauf mention contraire, dans le fichier TypeScript `userInterface.ts`.

5.3.1 Header

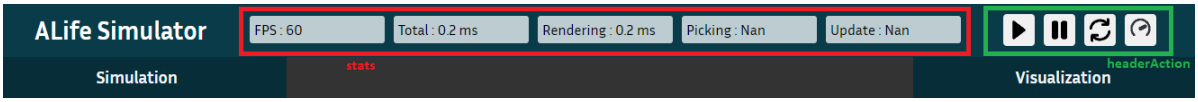


Figure 14 – Mise en avant la section Header de la page. En rouge les éléments de la balise stats et en vert ceux de la balise headerAction

Dans cette section de la page nous avons décidé de placer les différentes informations de performance de notre application regroupé dans une balise div avec l'identifiant *stats* en rouge sur la fig14. On retrouve aussi quatre boutons, encadré en vert sur la fig14, qui permettent de gérer le déroulement de la simulation : play, stop, reset et speed. En plus de ces éléments, nous avons jugé intéressant d'inscrire le nom de l'application, qui est temporaire, à notre page web afin qu'il apparaisse dans de possibles captures d'écran des futurs utilisateurs.

Voici plus en détail les actions réalisées lors d'un clic sur un des quatre boutons :

— **Play** : La boucle d'animation de l'`AnimationTimer` est lancé, le pas de simulation suivant qui était stocké devient le pas courant et une requête est envoyé au serveur à l'aide du Worker pour récupérer le pas de simulation suivant.

— **Pause** : La boucle d'animation est stoppé ainsi que les requêtes pour les pas de simulations suivants.

— **Reset** : Applique les mêmes actions que le bouton "Pause" et envoie un message au serveur pour que la simulation se réinitialise.

— **Speed** : Le comportement de ce bouton se trouve dans le fichier TypeScript *setDurationInterface*. Lorsque le curseur est déplacé, la duré d'animation stocké dans `viewer.ts` est modifié et impacte directement la fréquence de rafraîchissement.

La mise à jour des quatre éléments de *stats* est effectué dans un fichier TypeScript à part nommé *stats.ts*. On retrouve les différentes fonctions pour calculer les statistiques et celles pour actualiser le texte des différents éléments. La nature des mesures est détaillée à la section 5.4.2.

5.3.2 Panneau de configuration

Les deux panneau latéraux `Simulation` et `Visualization` possèdent tous les deux beaucoup de similarités au niveau de leur structure HTML mais aussi dans leurs style CSS. Nous avons donc choisi de créer un fichier de style nommé `configurationPanel.css` dans lequel on retrouve le style commun des deux panneaux, du bouton pour cacher le panneau, des items et paramètres par exemple. Les deux panneau ajoutent par la suite leurs propres styles, en particulier celui pour le positionnement sur la page et celui de la `scrollbar`.

De plus de nombreux paramètres sont affichés par ces deux panneaux, nous avons donc choisi de réaliser des `templates` pour chacun d'entre eux afin de faciliter leurs générations dans les autres `templates`. Il existe donc un `template` de paramètre complet comprenant le nom, la valeur par défaut et tout autres attributs pour chaque type de paramètres. Pour finir, et puisque cette balise n'existe pas dans la liste de celles proposées par Django, nous avons décidé de créer deux `templates` (`parameterSwitch.html` et `miniParameterSwitch.html`) pour réaliser l'équivalent d'un `switch...case` sur le type d'un paramètre. De cette façon nous n'avons pas dupliqué le bloc de code à chaque fois que nous voulons afficher un paramètre sans savoir son type à l'avance.

Figure 15 – Mise en évidence des éléments du panneau Simulation.

5.3.3 Simulation Panel

Comme indiqué précédemment le panneau `Simulation` regroupe les items d'interaction ayant un impact direct sur la simulation.

On y retrouve tout d'abord un menu déroulant avec l'ensemble des simulations disponibles, encadré en rouge sur la fig 15. Cet ensemble est mis en forme à l'aide d'une balise `select` dans laquelle est générée une balise `option` par nom de simulation. La liste de nom est donnée dans le contexte de rendu par `views.py`, qui les récupère grâce à une énumération dans `simulationManager.py`.

On retrouve ensuite les outils de sélection représentés par des icônes décrivant au mieux leur fonctionnement, encadré en vert sur la fig 15. Les icônes sont stockés localement sur le serveur dans le dossier `svg` des fichiers statiques. En dessous se trouve leur item de paramétrage, car chaque outils de sélection possède un ensemble de paramètres spécifiques. Ces paramètres sont représentés sous la forme de curseur pour définir leurs valeurs, affichée à côté pour plus de précision. Par exemple l'outil de sélection *BRUSH* représenté par un pinceau possède deux paramètres : l'intensité et le rayon. Cet item de paramétrage des outils peut aussi accepter qu'un outil ne possède aucun paramètre et l'indique le cas échéant. La façon dont les outils de sélection interagissent avec la simulation est détaillé dans la section 5.4.7.

Les deux derniers items du panneau `Simulation`, encadré en violet sur la fig 15, permettent de gérer les paramètres d'initialisation d'une simulation (nommé *Initialization* encadré en orange sur la fig 15) et les règles de cette dernière (nommé *Rules* encadré en bleu sur la fig 15). Dans notre architecture, ces deux items sont générés par le même template HTML mais avec un contexte de rendu différent.

Le fichier regroupant les deux items, appelé `simulationConfigSet.html`, demande deux rendus du template `simulationItem.html` en changeant le nom, l'identifiant HTML et la liste des paramètres à afficher. La modification des paramètres d'`Initialization` n'ont d'effet que lorsque l'utilisateur clique sur le bouton de réinitialisation. La modification des paramètres de `Rules` sont quant à eux mis à jour en temps réel.

À chaque paramètres de ces deux items, étant un ou plusieurs éléments HTML `input`, on associe un écouteur d'événement. Lorsqu'une nouvelle valeur est donné à l'`input`, la fonction de l'écouteur d'événement envoie à l'aide du WebSocket une requête au `Consumer`. Dans cette requête on retrouve l'identifiant du paramètre ainsi que sa nouvelle valeur.

5.3.4 Visualization Panel

Comme énoncé précédemment, le panneau `Visualization` permet à l'utilisateur de configurer la visualisation du système comme il le souhaite.

Le premier item de ce panneau (en rouge sur la fig 16) permet de choisir le maillage qui sera utilisé pour visualiser les cellules ou particules de la simulation. Tous les maillages disponibles se trouvent dans le dossier `models` des fichiers statiques. La balise `select` permettant cette sélection est automatiquement remplie à l'initialisation de la page avec tout ceux présents dans le dossier et les cellules sont directement changées lors de la sélection d'un nouveau maillage.

Le second item de ce panneau (en vert sur la fig 16) permet le choix de la fonction de transition pour chaque cellule entre deux pas simulation. Cet item contient donc différentes fonctions d'animations disponibles ainsi qu'un sélecteur pour pouvoir attribuer cette fonction seulement à certains attributs tel que la couleur ou la position. Le sélecteur possède également une option `All` pour attribuer la même fonction à tous

Figure 16 – Mise en évidence des éléments du panneau Visualization.

les attributs de visualisation. Cet item est construit a l'initialisation de la page puis géré par un fichier TypeScript nommé `animationInterface.ts` qui va automatiquement remplir le panneau sous forme de grille avec les animations présentes dans le fichier typescript `animationFunctions.ts` en dessinant la courbe à l'aide d'éléments *canvas* avec un contexte 2D. Ainsi, si l'on veut rajouter une autre fonction d'animation il suffit simplement de la rajouter dans le fichier typescript prévu à cet effet.

À partir du troisième item du panneau de `Visualization` on retrouve les item `Transformer` (en violet sur la fig 16). Nous avons eu beaucoup d'idées concernant l'interface utilisateur que nous allions développer pour associer les attributs de sortie de la simulation avec les attributs d'entrée de la visualisation. Pour des raisons de temps, nous avons finalement choisit une solution simple qui permet d'associer une seule sortie de la simulation avec une seule entrée de la visualisation pour un item de `Transformer`.

La sortie de la simulation est choisie par l'utilisateur via le menu déroulant sur la gauche de l'item. La liste des options disponibles est mise à jour en fonction du système complexe choisi et de ses paramètres. L'entrée de la visualisation affectée par l'item est indiquée grâce à son titre. Par exemple l'item encadré en orange sur la fig 16 nommée "Color" permet donc d'agir sur la couleur des cellules ou particule de la visualisation. Voici un exemple de trois associations différentes seulement en changeant la sortie de la simulation à associer à la couleur des cellules du *"jeu de la vie"* :

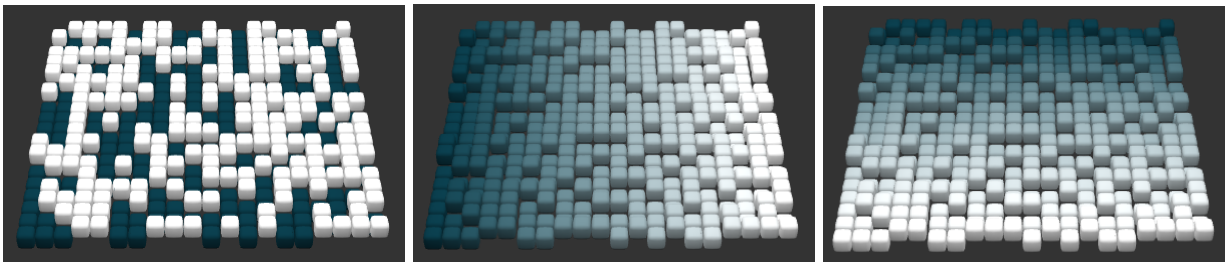


Figure 17 – Respectivement l'association de l'état, la position X et Y d'une cellule du "jeu de la vie" avec la couleur de la visualisation. Le bleu foncé est associé à 0 et le blanc à 1.

Une fois que l'association est faite entre simulation et visualisation, divers paramètres sont disponibles. Par exemple pour la couleur l'utilisateur doit choisir deux couleur, l'une pour la valeur 0 et l'autre pour la valeur 1. Le `Transformer` est ensuite responsable de la possible normalisation et interpolation linéaire entre ces deux couleurs. Certains `Transformer` sont nécessaire pour une visualisation satisfaisante (position X et Y par exemple) et d'autre non comme la couleur qui peut être gérée indépendemment sur chaque canaux RGB à l'aide de `Transformer` dédiés. Dans le cas où un il n'est pas nécessaire, l'item du `Transformer` possède une croix en haut à droite pour le supprimer, comme sur la fig 16 avec le `Transformer Color` encadré en orange.

En bas du panneau `Visualization` nous avons placé une interface permettant d'ajouter de nouveau `Transformer`. L'utilisateur doit préciser quel aspect de la visualisation il veut impacter et appuie ensuite sur le bouton/icône "+".

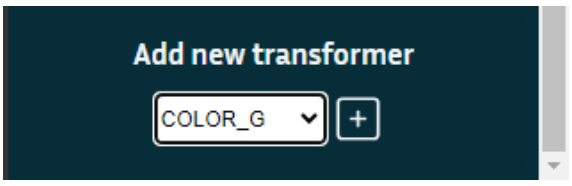


Figure 18 – Section en bas du panneau `Visualization` permettant d'ajouter un nouveau `Transformer`

L'écouteur d'événement du bouton "+" a pour but d'envoyer une requête HTTP au serveur en récupérant la valeur sélectionné dans la balise `select` adjacente. La requête est adressée à une URL de la forme *"addTransformer/<transformerType>"* qui appel la fonction `addTransformer` de `views.py`, fichier en charge des

rendus HTML. La fonction récupère l'objet `Transformer` correspondant au type passé en paramètre ainsi que les paramètres de ce dernier afin de créer le contexte de rendu. La fonction renvoie le rendu du template Django `transformerItem.html` comme réponse. Lorsque que l'écouteur d'événement du bouton "+" reçois la réponse du serveur, il analyse sont contenu afin de créer un objet HTML et l'ajoute en bas du panneau `Visualization`. Le fonctionnement des `Transformer` est détaillé à la section 5.4.5.

5.4 Visualisation

5.4.1 Caméra

Pour la mise en place du rendu graphique, nous avons dans un premier lieu créé une caméra dynamique, afin de tirer profit des capacités d'une visualisation 3D. La caméra que nous avons implémenté est de type *Trackball*. Elle permet de tourner autour de la visualisation, sans perdre de vue son centre. Nous avons limité sa position à la partie supérieure de la visualisation, avec la possibilité de zoomer vers le point central.

Afin de déplacer la caméra, l'utilisateur peut effectuer un clic maintenu sur la molette de la souris à la manière des logiciels de modélisation 3D comme `Blender` [3]. Mais il a aussi la possibilité d'utiliser la touche *Shift*, nous avons ajouté cette possibilité pour les utilisateurs de *trackpad*. Le zoom se fait de la même manière que le défilement vertical au sein d'une page (roulement de la molette ou glissement sur le *trackpad*).

5.4.2 Mesure des performances

L'un des aspects centraux de ce projet est la vitesse de rendu, nous voulons que l'utilisateur puisse avoir une visualisation cadencée à la fréquence de rafraîchissement de son écran. C'est dans l'objectif d'avoir une meilleure compréhension de ces performances que nous avons ajouté une classe *Stats*. Elle permet la gestion de différents chronomètres pouvant être démarrés et stoppés par une classe extérieure. Nous avons fait le choix de chronométrer les points clés suivants : le rendu d'une image, la sélection par l'utilisateur et la mise à jour des différents *buffers*. Les temps d'exécution sont ensuite directement affichés dans l'interface avec un compteur d'image par seconde (voir la figure 14).

En plus des mesures affichées dans l'interface, nous avons ajouté différents chronomètres au sein du module *Transmission*, ces mesures sont exploitées par les différents tests de performance. Nous fournissons plus de détails sur ceux-ci dans la section 6.

5.4.3 Affichage d'un grand nombre de maillages

Le premier enjeu de la visualisation est la mise en place d'une technique de rendu permettant l'affichage d'un grand nombre de maillages. En effet, pour chaque cellule d'une grille (ou pour chaque particule), nous voulons que l'ensemble des paramètres visuels lui soit propre et unique.

Dans le cas où les cellules sont représentées par des *vertices* appartenant à un même maillage (définissant la grille), un seul *draw call* est nécessaire, les attributs des cellules sont représentés par les attributs des *vertices*. En revanche, si nous voulons que les cellules soient représentées par un maillage qui lui est propre, l'approche naïve va consister en un *draw call* par cellule, avec ses attributs représentés par des *uniforms*. Le problème de cette approche est qu'elle va conduire à un goulot d'étranglement du côté CPU, avec un trop grand nombre de *draw calls* (40000 pour une grille de 200 par 200). Le second frein à l'utilisation de cette méthode est la perte de généralisation, le passage d'une visualisation par des *vertices* à une visualisation par des maillages introduit une modification du côté CPU et GPU (une modification de la procédure de rendu et une modification du *vertex shader*).

L'approche retenue pour contourner ce problème est l'utilisation de l'instanciation. Avec cette approche, dans tous les cas, les attributs des cellules sont considérés comme des attributs de *vertex*. Dans le cas de la représentation par des *vertices*, il n'y a aucune modification à réaliser. Dans le cas où les cellules sont représentées par des maillages, le GPU réalise l'indexation des attributs non plus par *vertex*, mais par instance. En un *draw call*, le rendu de l'ensemble des maillages est réalisé, les attributs communs sont indexés par *vertex* et les attributs propres à chaque cellule sont indexés par instance.

Cette méthode permet donc l'utilisation d'un seul et même *draw call* pour l'ensemble des cellules, mais aussi l'écriture d'un unique *vertex shader* indépendant du mode de représentation. Cela facilite aussi le

passage des données de sortie des systèmes aux données d'entrée des *shaders*, aucun traitement dépendant de la visualisation n'est nécessaire.

La gestion des différents tampons envoyés au GPU est effectuée au sein de la classe `MultipleMeshInstance`. Elle est initialisée avec un maillage donné dans le format `.obj`, les données de ce maillage sont directement sauvegardées au sein de la classe. Les attributs de chaque instance sont sauvegardés et mis à jour à l'aide de la classe `InstanceAttribBuffer`, qui va s'occuper du transfert des données fournies sous forme de `TransformableValues` vers les tampons du GPU.

5.4.4 Récupération des valeurs de sortie des systèmes

Notre méthode de transmission (via un *Web Socket*) encode les données sous forme de JSON. Pour pouvoir être exploitées lors de la visualisation, les données de sortie des systèmes (i.e. un ensemble de positions et d'états pouvant être des scalaires ou des vecteurs) doivent être mise sous forme de tableau, pour ensuite être transformées en caractéristiques visuelles.

L'enjeu de cette partie du logiciel (la partie *Transmission* de la figure 8) est de fournir des données préparées pour la visualisation sans que son exécution puisse interférer avec la boucle de rendu principale.

À l'aide d'outils de profilage, nous avons rapidement constaté que l'exécution synchrone du rendu graphique et de la réception / préparation des données des systèmes conduisait à une perte significative de performance et de qualité d'utilisation.

Afin d'assurer l'indépendance et la performance de chacune de ces deux composantes, la récupération des données est mise sur un *thread* différent à l'aide des `WebWorkers` disponibles au sein de `JavaScript`. Ce `Worker` contient l'interface avec le *thread* principal, l'interface du `WebSocket` et une classe qui prépare les données reçues tout en les mettant dans un tampon, le `StatesBuffer`.

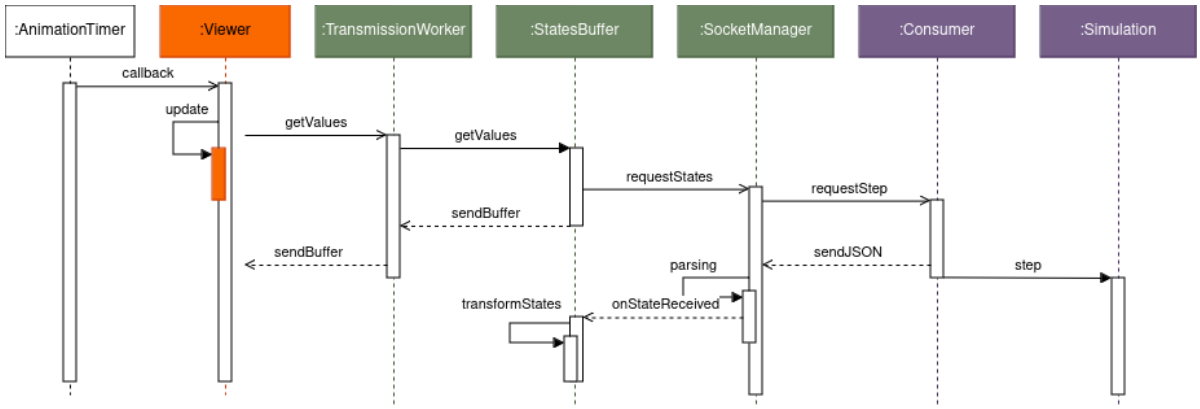


Figure 19 – Diagramme de séquence correspondant à une mise à jour du pas de temps visualisé. L'`AnimationTimer` déclenche la mise à jour de manière cadencée. Le `Viewer` fait la mise à jour de la visualisation (les `InstanceAttribBuffer`) et fait la demande du prochain pas de temps de manière asynchrone. L'ensemble du module Transmission *retourne le pas de temps qui est en mémoire tout en faisant la demande du pas de temps suivant au serveur. Le serveur (en violet) retourne le pas de temps courant et déclenche la simulation du pas suivant. En somme, chaque acteur garde en mémoire un pas de temps et fait sa mise à jour qu'une fois que ce pas lui a été demandé. La visualisation est donc en retard de trois pas de temps sur la simulation.*

À la réception de la transmission par le `SocketManager`, les données sont *parsées* et l'objet `JavaScript` ainsi créé est directement envoyé au `StatesBuffer`. Ce dernier a pour tâche de transformer l'objet *parsé* en un ensemble de tableaux (`TransformableValues`) dont l'emplacement mémoire sera transmis au *thread* principal sur demande. Lorsque les valeurs du tampon sont envoyées, le `Web Socket` fait immédiatement une demande au serveur pour récupérer le prochain ensemble de valeurs. Cette séquence est illustrée sur la figure 19.

Le `TransmissionWorker` est l'unique point de passage entre la visualisation et les systèmes complexes. C'est donc au travers de celui-ci que passent toutes les requêtes d'interaction et de paramétrage des modèles, outre les transmissions des états du système.

5.4.5 Application de transformations visuelles

Afin de permettre à l'utilisateur une personnalisation de la visualisation en fonction des données issues des systèmes complexes, un point de passage crucial est la transformation de ces données en caractéristiques visuelles. Étant donné que les simulations sont effectuées en même temps que le rendu graphique, un enjeu de cette transformation est sa vitesse d'exécution. De plus, ces transformations doivent être flexibles et adaptables aux paramètres déterminés par l'utilisateur.

La méthode implémentée passe par des *shaders* compilés à la volée. Les attributs fournis à ces *shaders* sont ceux propres à la primitive représentée ainsi qu'un ensemble de valeurs par instance, il s'agit des tableaux fournis par le `TransmissionWorker` pour le pas de temps courant et suivant (cf. 5.4.6).

De plus, ces *shaders* sont munis d'un tag. Ce tag est remplacé par les différentes transformations que l'utilisateur désire appliquer. Les transformations sont définies par trois propriétés : la caractéristique visuelle qu'elle modifie, la sortie du système qu'elle prend en entrée et ses différents paramètres.

La combinaison de ces trois propriétés permet la construction de trois blocs d'instructions. Le premier est la définition des différentes valeurs utilisées pour effectuer les transformations (il s'agit d'une interpolation des attributs du *shader*, cf. 5.4.6). Le second bloc correspond à la définition de l'ensemble des paramètres de chaque transformation. Le dernier bloc effectue l'ensemble des opérations à appliquer à l'entrée, il s'agit de multiplications ou d'interpolations.

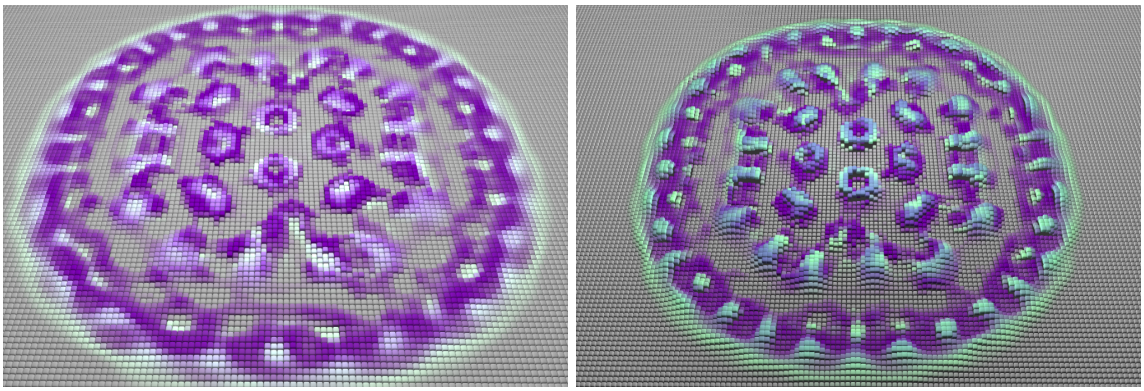


Figure 20 – Une instance de Lénia avec deux canaux d'états. Une accumulation de transformations est appliquée. La représentation à droite, exacerbe le canal représenté en vert, en augmentant la taille et la hauteur des éléments lui étant associé.

De cette manière, l'ajout ou le retrait d'une transformation engendre l'ajout ou la suppression des lignes de code qui lui sont associées au sein de chacun de ces trois blocs. La modification d'un paramètre de transformation n'engendre que la modification du deuxième bloc. Le *shader* est directement recompilé après modification. Les opérations au sein du troisième bloc sont réalisées dans l'ordre de création des transformations, et chacune d'entre elles modifie un paramètre visuel de manière additive. Ainsi, si l'utilisateur crée plusieurs transformations ayant un impact sur la couleur, le résultat sera la somme de chacune de ces transformations. Ce comportement est comparable à une approche par calque (correspondant à l'interface, cf. section 5.3.4), son évolution fait partie des éléments envisageables pour les travaux futurs. La figure 20 illustre la combinaison de plusieurs de ces transformations.

La création et la suppression des transformations est gérée par le `TransformerBuilder`, il va associer un identifiant unique à chaque transformation, cet identifiant est utilisé par le module *Interface*. C'est aussi

cette classe qui va générer le premier bloc d'instructions, étant donné qu'une même entrée peut être commune à plusieurs transformations. Les deuxièmes et troisièmes blocs sont générés par les classes héritant de `Transformer`. Les variables et les fonctions utilisées par ces blocs d'instructions sont définies dans le fichier `shaderUtils.ts`, ce fichier permet l'interfaçage de la récupération des informations des *shaders*, un avantage pour la maintenabilité du logiciel.

5.4.6 L'animation du rendu

La fréquence à laquelle la scène est rendue coïncide avec la fréquence de rafraîchissement de l'écran utilisé. Afin d'avoir une récupération uniforme et fluide des différents pas de temps de la simulation, les demandes de mise à jour de la visualisation sont définies par un `AnimationTimer`. Cette classe génère un signal à une fréquence définie par l'utilisateur, ce signal est ensuite envoyé au `TransmissionWorker` pour que le *thread* principal puisse récupérer un nouveau pas de temps de la simulation et mettre à jour les différents attributs des *shaders*, cette séquence est illustrée sur la figure 19.

Afin d'offrir à l'utilisateur plus de personnalisation, et dans l'optique de rendre les visualisations plus esthétiques, un système d'animation entre les pas de temps de la simulation est ajouté. Les *shaders* prennent en attribut les différentes valeurs pour le pas de temps de la simulation courant mais aussi pour le pas de temps suivant. Entre deux signaux de l'`AnimationTimer`, les valeurs de ces attributs sont interpolées.

Par défaut, cette interpolation est linéaire avec un *spacing* constant. Cependant, l'utilisateur a la possibilité de définir un *spacing* différent par paramètre visuel (position, couleur, etc), il s'agit de la région verte de la figure 16. Ces différents timings sont envoyés aux *shaders* sous forme de *uniforms* et sont utilisés pour interpoler les valeurs d'entrée des différentes transformations.

5.4.7 Outils de sélection et interactions

La classe `Viewer`, qui est au centre de l'architecture du logiciel, n'a pas l'information de la simulation pour laquelle elle effectue une visualisation. Ce qui implique que les différentes interactions avec les éléments de la simulation doivent passer par le serveur. En revanche, la classe `Viewer` détient tout de même l'information des différentes bornes des domaines de définition des valeurs de sortie de la simulation. À partir de celles-ci, un masque peut être constitué pour déterminer les zones avec lesquelles l'utilisateur interagit.

Pour ce faire, différents outils de sélection sont à disposition de l'utilisateur : sélection par boite ou brosse. Ces deux outils reposent sur la même méthode. Un rayon est tiré dans la direction du curseur de l'utilisateur, à partir du domaine de définition du système et des différentes transformations qui lui sont appliquées, une position au sein du masque de sélection est donnée à ce rayon. En fonction de cette position et de l'outil utilisé, le masque est rempli (avec des valeurs flottantes) puis il est envoyé au serveur pour qu'il effectue un traitement en conséquence.

La sélection par boite met tous les éléments sélectionnés à 1 dans le masque. Pour la sélection par brosse (représentée sur la figure 21), une distance maximale est définie par l'utilisateur. La distance entre l'élément sélectionné et les autres éléments du masque est mesurée, si cette distance est inférieure à la maximale, l'élément prend une valeur proportionnelle à une intensité définie par l'utilisateur. De plus, la brosse peut être circulaire ou carrée, ceci affecte la manière dont la distance est calculée (distance euclidienne ou de Tchebychev). Nous avons aussi essayé d'implémenter une sélection par lasso, mais celle-ci étant sujette à des bugs, nous avons priorisé le développement de la sélection par brosse.

L'outil de sélection courant est défini par le `SelectionManager`, chaque outil hérite de la classe `SelectionTool`. Une fois que le masque est défini, il est envoyé au serveur avec le `TransformableValues` correspondant au pas de temps qui est visualisé. Le serveur applique ensuite le traitement et retourne un nouvel ensemble d'états correspondant au pas de temps après modifications. Cette opération est très couteuse en temps, notamment à cause du fait que l'ensemble des états courant est envoyé au serveur, il s'agit d'un point d'amélioration envisagé pour les travaux futurs.

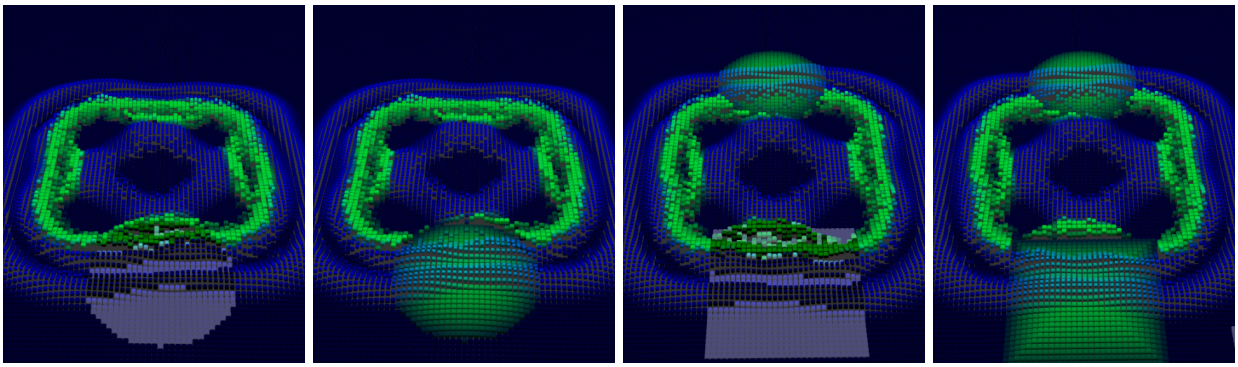
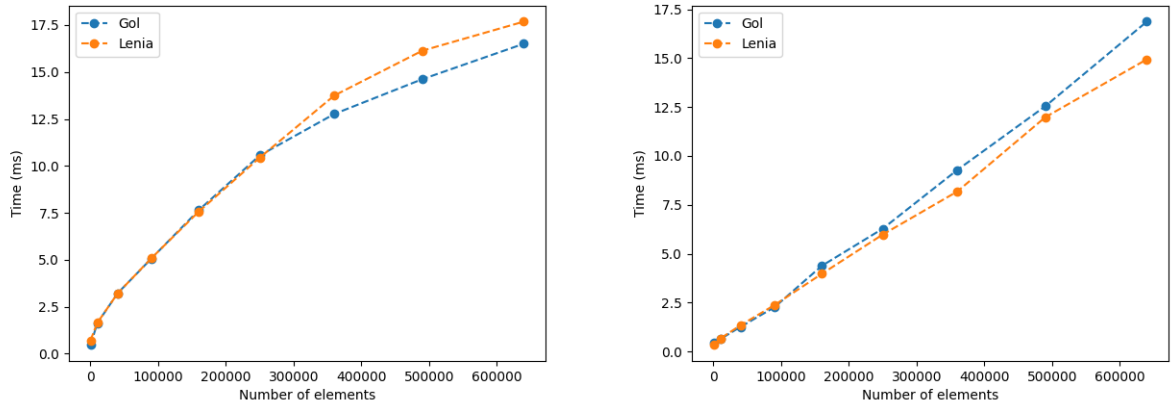


Figure 21 – Représentation des deux types de brosses, ainsi que le résultat de l'interaction une fois le masque appliqué. La simulation est une instance de Lenia à deux canaux. Le vert et le bleu représente chacun d'entre-eux.

6 Tests de performances

Afin de tester les performances de la visualisation, nous avons mis en place un script `Python` utilisant `Selenium` [22]. À l'aide de cette bibliothèque, nous avons pu simuler les interactions de l'utilisateur avec l'interface. Les différents chronomètres sont gérés par la classe `Stats` (comme décrit en section 5.4.2) et `Selenium` accède à leur valeur au travers des éléments HTML de la page. L'ensemble des valeurs mesurées dans les figures 22, 23 et 24 ont été effectuées en parallèle durant la même session.

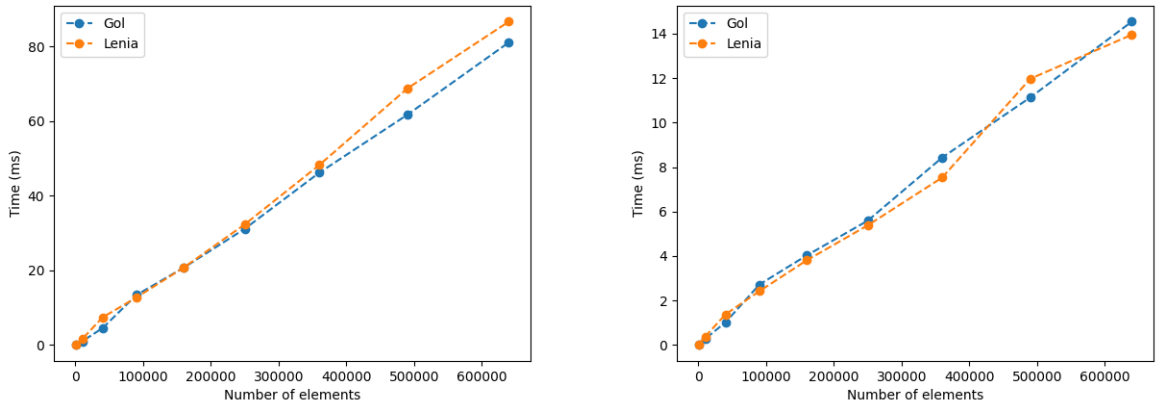


(a) Temps de rendu moyen

(b) Temps moyen de mise à jour des buffers *du GPU*

Figure 22 – Mesures des performances de la visualisation. Les modèles représentés sont le jeu de la vie et Lenia, en fonction du nombre d'éléments composant les matrices. (a) Le temps mis pour rendre une image. (b) Le temps mis pour mettre à jour les différents buffers *du GPU.*

Nous avons concentré nos mesures au niveau des différents goulots d'étranglement. Le premier test de performance se concentre sur les temps de rendu et de mise à jour des *buffers* du GPU, la figure 22 représente ces différentes mesures. Il est important de remarquer, au vu de ces résultats, que la performance du rendu est indépendante du modèle visualisé, ce qui valide l'un des objectifs de notre projet. De plus, cela met en évidence l'un des goulots d'étranglement de notre application : la mise à jour des données sur le GPU. En effet, on peut voir la croissance linéaire du temps mis par le système pour effectuer cette action. Bien que cela n'ait pas lieu à chaque image rendue, la perte de fluidité est ressentie au moment de la mise à jour.



(a) Temps moyen mis pour parser les données issues du serveur

(b) Temps moyen mis pour transformer les données en un format copiable pour les buffers *du GPU*

Figure 23 – Mesures des performances de la transformation des données issues du serveur. Les modèles représentés sont le jeu de la vie et Lenia, en fonction du nombre d'éléments composant les matrices. (a) Le temps mis pour parser *les données issues du serveur. (b) Temps mis pour transformer les données parsées en des tableaux transmissibles aux* buffers *du GPU.*

La deuxième batterie de mesures que nous avons effectué se trouve au niveau du `WebWorker`, faisant l'interface entre le serveur et la visualisation. Plus particulièrement, nous avons mesuré le temps mis pour parser les données reçues (figure 23a), ainsi que le temps mis pour transformer ces données (figure 23b). Ces mesures montrent l'intérêt d'implémenter une méthode de transmission plus efficace, afin de contourner le coût engendré par le *parsing* des JSON.

Lors de sa réception, les différents états du système ont besoin d'être réagencés afin de correspondre à un format compatible avec les *buffers* du GPU. Ce réagencement est nécessaire pour garantir le découplage de la simulation et de la visualisation. Cependant, il peut avoir un coût significatif sur les performances, bien qu'il soit effectué de manière asynchrone à la mise à jour des *buffers*.

Pour finir, nous avons aussi mesuré le temps entre la demande et la réception d'un nouvel ensemble d'états (figure 24). Bien que ces mesures soient fortement dépendantes de l'infrastructure du réseau, nous pouvons en retirer une information. Comme l'ensemble des mesures des figures présentées dans cette section ont été effectuées en parallèle, on peut constater que les durées de transmission n'ont aucun impacte sur la fluidité du rendu, ceci validant le découplage des systèmes et de la visualisation. La potentielle latence de la transmission n'est ressentie qu'au moment de la mise à jour des éléments du système par l'utilisateur (au moment de l'initialisation du système et de l'utilisation d'outil de sélection).

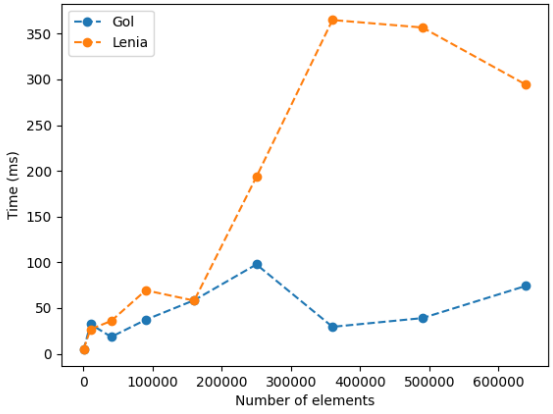


Figure 24 – Mesures du temps moyen mis entre l'envoi d'une requête des états par le `WebWorker` *et la réception de la réponse.*

Toutes les mesures ont été effectuées avec une machine ayant la configuration suivante :

Ubuntu 22.04.03 LTS, Firefox 123.0, AMD Radeon RX6600, AMD Ryzen 5 3600.

7 Gestion de projet

Pour mener à bien ce projet nous nous sommes organisé à l'aide de la méthode SCRUM. Puisque notre projet est organisé sous forme de modules nous avons pu nous répartir les tâche aisément avec chaque membre du groupe responsable d'un des quatre modules principaux à savoir : la simulation des modèles de systèmes complexe, la transmission des données, l'interface utilisateur et la visualisation 3D.

Les objectifs du projet, établi au début de celui-ci, ont servi de base à la conception de nos différentes tâches. À l'aide des réunions hebdomadaires que nous avons organisé avec nos clients, nous définissions les tâches à réaliser pour notre prochain *sprint* et effectuions des démonstration de notre application. Ces tâches étaient ensuite mis en forme dans l'application web Trello, le même logiciel qui nous à permit de créer et gérer notre diagramme de Gantt C. Nous alimentions aussi un forum privé OpenLab avec des vidéos démonstratives afin de partager nos avancées avec nos clients et d'autres membres de l'Inria.

Pour le versionnage du code de l'application nous avons utilisé Git. Notre dépôt était constitué d'une branche principale (main), d'une branche par module et d'autres branches pour effectuer des implémentations à risque ou des tests. Nous organisions des réunions d'équipes hebdomadaire dans lesquelles nous fusionnions les modifications de nos différentes branches du *sprint* précédent sur la branche principale en prenant soin que l'application soit toujours exécutable pour de potentielles démonstrations.

Conclusion (Bilan / persepectives)

Ce projet avait pour objectif de fournir une application web permettant la simulation et la visualisation de systèmes complexes. Et ce, dans la perspective d'avoir un outil pour l'exploration de ces systèmes, au travers de l'interaction.

Notre réalisation offre en premier lieu une formalisation des données issues des systèmes complexes, ce format permet le découplage de la visualisation et de la simulation. En effet, notre application est capable de représenter des automates cellulaires, aussi bien discrets que continus, avec la possibilité d'exploiter un nombre variable de leurs valeurs de sortie. Cette flexibilité s'étend aussi au domaine de définition des systèmes, offrant la capacité de visualiser des systèmes définis dans un espace continu au sein de la même application. De plus, la visualisation étant en temps réel, offre à l'utilisateur la possibilité d'interagir avec le système au travers de celle-ci. L'utilisation de calques de transformations permet une personnalisation des caractéristiques visuelles de la représentation, ceci laissant la possibilité de mettre en évidence des phénomènes émergents du système. L'interaction ne se cantonne pas uniquement à l'aspect visuel, mais elle permet aussi de modifier directement l'état du système.

Cependant, notre application souffre tout de même de différents goulots d'étranglements. Notamment au moment de la mise à jour des tampons du GPU. Pour contourner ce problème, nous avons pensé utiliser deux ensembles de tampons, le premier serait celui présent sur le GPU, le second serait préparé entre deux boucles d'animation. Au moment où la mise à jour est demandée, les deux ensembles de tampons sont échangés. Le *parsing* des données envoyées par le serveur est aussi trop coûteux, nous avons utilisé le format JSON pour ne pas nous focaliser sur l'aspect réseau de l'application. Cependant, la mise en place d'une communication pouvant être lue rapidement par le client sera nécessaire pour améliorer les performances de l'application.

La sélection, bien que fonctionnelle, ne garantit pas encore une totale indépendance vis-à-vis du système représenté. L'utilisation du domaine de définition n'est pas encore pleinement exploitée, cantonnant la sélection aux systèmes matriciels. De plus, afin d'améliorer la vitesse de réponse à l'interaction, le client devrait envoyer le masque de sélection accompagné d'un numéro de pas de temps, il est actuellement accompagné de l'ensemble des valeurs du pas courant. Cette modification impliquerait la mise en mémoire des différents pas de la simulation par le serveur.

Actuellement, afin de garantir l'utilisation de l'application sur une machine avec une configuration minimale, le nombre d'ensembles d'états représentables est de quatre. Ce nombre peut être passé à 16 en utilisant des `vec4` au lieu de `float` au sein des attributs du *shader*. Cependant, cette modification va entraîner un surcoût au moment de la préparation des données (figure 23b).

La méthodologie que nous avons employée pour la réalisation des mesures de performance ne s'est avérée fonctionnelle que pour une configuration de machine. Pour une prise en compte globale des mesures sur tous les supports, une gestion au cas par cas doit être mise en place. La contrainte étant principalement induite par le navigateur utilisé.

Du côté du serveur, l'utilisation de la bibliothèque `JAX` permet d'accélérer le calcul des pas de simulation, mais cela requiert la conversion de toutes les données objets en `floats` (un des seuls types supportés par `JAX`). Cette conversion peut s'avérer très coûteuse et il faudrait donc réfléchir à améliorer la compatibilité entre le code du module simulation et `JAX`. Il est important de noter que le découplage de la simulation et de la visualisation offre la possibilité future d'implémenter différents *backend*, la seule contrainte à leur mise en place est le respect du format de donnée et de communication utilisé par le `TransmissionWorker`.

Ces limitations sont principalement dues à la nature du projet. Cette preuve de concept, bien que perfectible, laisse entrevoir de nombreux axes de développement pouvant mener à une application d'exploration, visualisation et hybridation de systèmes complexes.

Le partage entre utilisateurs et la combinaison des paramètres de simulation était initialement un besoin du logiciel, on peut comparer cette fonctionnalité à celles offertes par `Picbreeder` [21]. Cependant, ce projet a été recentré, avec l'accord des clients, dans le perfectionnement de l'aspect modulaire de l'application. Cette modularité a toute fois été pensée avec les besoins initiaux, ce qui laisse l'opportunité future d'implémenter un processus de sauvegarde de la configuration du système et des paramètres de visualisation. Ouvrant la voie à un partage entre les utilisateurs et une hybridation automatique ou contrôlée des différents paramètres. Le tout dans l'objectif de renforcer l'aspect exploratoire de l'application.

En plus des interactions réalisées par l'utilisateur à l'aide de la souris, il pourrait être intéressant de faire interagir des simulations entre elles. En effet, on pourrait imaginer que l'état d'une première simulation soit converti en masque qui pourrait être utilisé pour déclencher une interaction sur une seconde simulation. Cela demanderait principalement de développer les options adéquates dans l'interface graphique de l'application.

Pour enrichir l'information transmise par le biais de la visualisation, il pourrait être possible d'ajouter un module effectuant des traitements sur les données de la simulation en amont. Ce traitement pourrait, par exemple, exacerber la variation d'un élément du système au fil du temps. Ou bien, faire ressortir des caractéristiques locales par le biais de filtres de convolution.

Nous avons fait le choix de suivre une approche par calque pour la personnalisation des caractéristiques visuelles, cette approche pourrait être développée afin d'augmenter les différentes associations possibles entre les calques. Il peut aussi être envisagé de changer cette approche pour un système nodal, offrant plus de possibilités à l'utilisateur. Ou bien, intégrer un éditeur de code à l'application, afin de directement modifier le *shader* effectuant le rendu.

Couplé à ces options de personnalisations, d'autres visualisations peuvent être envisagées. Nous nous sommes cantonnés à l'association d'un élément du système à un maillage, il pourrait être possible d'associer cet élément à un unique *vertex* pour former un champ de hauteur, à un texel au sein d'une texture plaquée sur un maillage, ou bien même à un son, ce qui augmenterait grandement la capacité à mettre en exergue l'auto-organisation des systèmes complexes simulés.

8 Annexes

A Documentation de la classe Simulation

Simulation class documentation

class **simulation.simulation.Simulation**(*init_states: list[State] | None = None*, *rules: list[Param] | None = None*)

Abstract super-class to extend if you want to add a new simulation

Parameters:
- **init_states** (*list[State]*) – Initial states of the simulation
- **rules** (*list[Param]*) – parameters required to run the simulation

applyInteraction(*id: str*, *mask: Array*)

Apply the specified interaction to the current states of the simulation

Parameters:
- **id** (*str*) – Id of the interaction
- **mask** (*jnp.ndarray*) – A 2D mask of floats used to apply the interaction

current_states= *None*

A list of State to be updated at each step. This attibute can contain a single state as long it is in a single element list

abstract property **default_rules**

Abstract attribute : list of Param that will be exposed to the user and can be modified anytime during the simulation

Return type: list[Param]

getRuleById(*id: str*)

Access the current value of a paramer used to run the simulation

Parameters: **id** – Name of the paramer
Returns: The value of the parameter

getRules()

Access the rules parameters

Returns: the exposed parameters used to run the simulation
Return type: list[Param]

abstract **initSimulation**(*init_states=None*, *rules=None*, *init_param=None*)

Method called before the simulation starts. It is expected to set the initial states of the simulation.

Parameters:
- **init_states** (*list[Param]*) – Optional states for the simulation
- **init_states** – Optional parameters for the initializatin

abstract property **initialization_parameters**

Abstract attribute containing list of Param that will be exposed to the user and set before the simulation starts

Return type: list[Param]

abstract **set_current_state_from_array(*new_state*)**

Set the states of the simulation form an arbitrary array-like representation. Used for convenience when handlind external data.

Parameters: **new_state** (*list* | *ndarray*) – Reprensnetation of states of the simulation

abstract **step()**

Method executing a state of the simulation. It is expected to update the attribute :py:attr:current_states.

to_JSON_object()

Converts the current states of the simulation to JSON-serializable python object. Used for convenience when manipulating states from outside of the simualtion module.

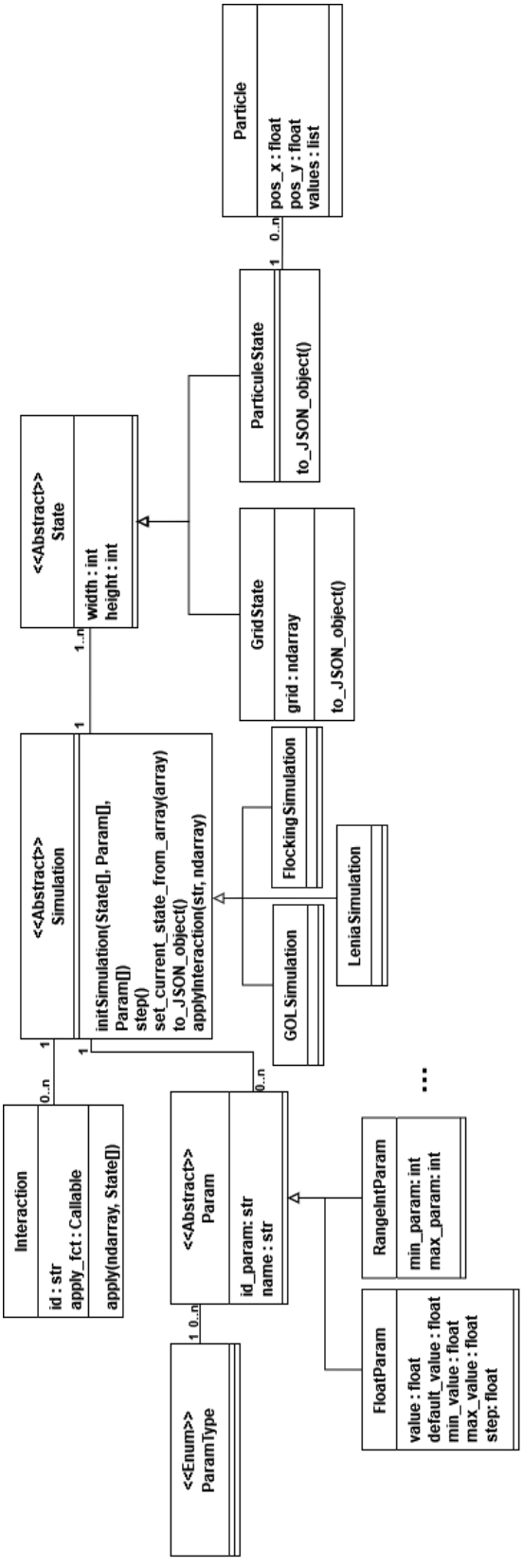
Returns: a JSON-serializable representation of the current states of the simulation.

updateRule(*json*)

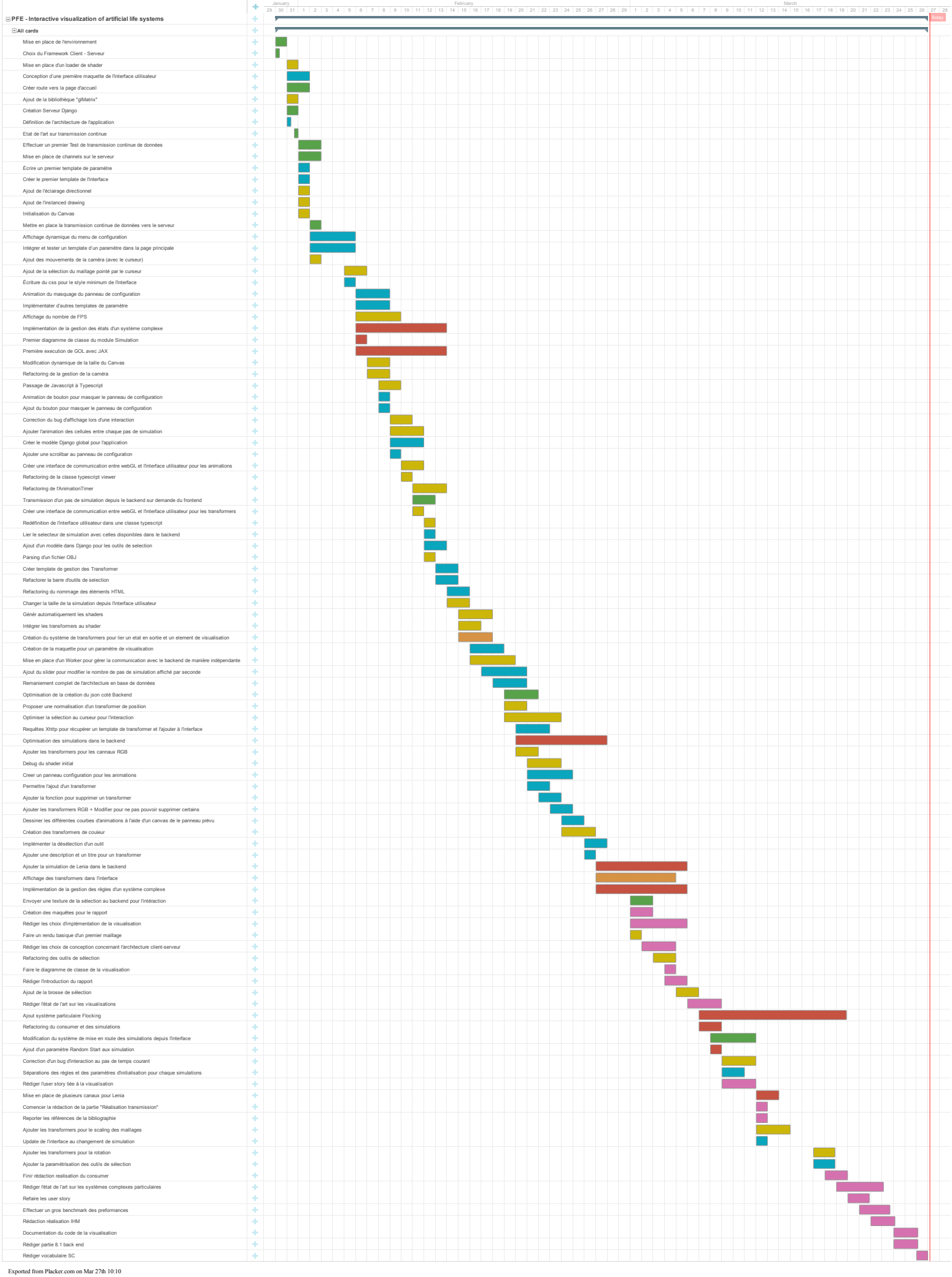
Set the value of a paramer used to run the simulation

Parameters: **json** – a dictionary representation of the parameter”

B Diagramme de classes du module de simulation



C Diagramme de Gantt



Références

[1] BABYLON.JS CONTRIBUTORS. *babylon.js*. Version 6.39.0. URL : https://www.babylonjs.com/.

[2] Randall D. BEER. « The Cognitive Domain of a Glider in the Game of Life ». In : *Artificial Life* 20.2 (avr. 2014), p. 183-206. ISSN : 1064-5462. DOI : 10.1162/ARTL_a_00125. URL : https://doi.org/10.1162/ARTL%5C_a%5C_00125.

[3] BLENDER DEVELOPMENT TEAM. *Blender*. Version 4.0.2. URL : https://www.blender.org/.

[4] Mike BOSTOCK et OBSERVABLE INC. *D3*. Version 7.8.5. URL : https://d3js.org/.

[5] Dirk BROCKMANN. *Complexity Explorables*. (accédé le 02/03/2024). URL : https://www.complexity-explorables.org/.

[6] Bert Wang-Chak CHAN. « Lenia and Expanded Universe ». In : ALIFE 2023 : Ghost in the Machine : Proceedings of the 2023 Artificial Life Conference ALIFE 2020 : The 2020 Conference on Artificial Life (2020), p. 221-229.

[7] Bert Wang-Chak CHAN. *Lenia Web Demo*. (accédé le 02/03/2024). URL : https://chakazul.github.io/Lenia/JavaScript/Lenia.html.

[8] Bert Wang-Chak CHAN. « Lenia : Biology of Artificial Life ». In : *Complex Systems* 28.3 (oct. 2019), p. 251-286. ISSN : 0891-2513. DOI : 10.25088/complexsystems.28.3.251. URL : http://dx.doi.org/10.25088/ComplexSystems.28.3.251.

[9] Bert Wang-Chak CHAN. « Towards Large-Scale Simulations of Open-Ended Evolution in Continuous Cellular Automata ». In : *Proceedings of the Companion Conference on Genetic and Evolutionary Computation*. GECCO '23 Companion. Lisbon, Portugal : Association for Computing Machinery, 2023, p. 127-130. DOI : 10.1145/3583133.3590670. URL : https://doi.org/10.1145/3583133.3590670.

[10] Graeme CUMMING. « Introduction to Mechanistic Spatial Models for Social-Ecological Systems ». In : jan. 2011, p. 67-85. ISBN : 978-94-007-0306-3. DOI : 10.1007/978-94-007-0307-0_4.

[11] DJANGO SOFTWARE FOUNDATION. *Django*. Version 5.0.1. 4 jan. 2024. URL : https://djangoproject.com.

[12] Martin GARDNER. « Mathematical Games : the fantastic combinations of John Conway's new solitaire game "life" ». In : *Scientific American* 223.4 (1970), p. 120-123. ISSN : 00368733, 19467087. URL : http://www.jstor.org/stable/24927642 (visité le 21/12/2023).

[13] Amanda GHASSAEI. *gpu-io*. (accédé le 02/03/2024). URL : https://github.com/amandaghassaei/gpu-io.

[14] Dean JACKSON et Jeff GILBERT. *WebGL2*. editor draft. URL : https://registry.khronos.org/webgl/specs/latest/2.0/.

[15] Brandon JONES et Colin MACKENZIE IV. *glMatrix*. (accédé le 02/03/2024). URL : https://github.com/toji/gl-matrix/tree/glmatrix-next.

[16] Kai NINOMIYA ; Brandon JONES ; Jim BLANDY ; Myles C. MAXFIELD ; Dzmitry MALYSHAU et Justin FAN. *WebGPU*. working draft. URL : https://www.w3.org/TR/webgpu/.

[17] Alexander MORDVINTSEV. *SwissGL*. (accédé le 02/03/2024). URL : https://github.com/google/swissgl.

[18] Erwan PLANTEC et al. « Flow-Lenia : Towards open-ended evolution in cellular automata through mass conservation and parameter localization ». In : *ALIFE 2023 : Ghost in the Machine : Proceedings of the 2023 Artificial Life Conference*. 2023.

[19] Craig W. REYNOLDS. « Flocks, herds and schools : A distributed behavioral model ». In : *SIGGRAPH Comput. Graph.* 21.4 (août 1987), p. 25-34. ISSN : 0097-8930. DOI : 10.1145/37402.37406. URL : https://doi.org/10.1145/37402.37406.

[20] Samuel S. SCHOENHOLZ et Ekin D. CUBUK. *JAX, M.D. : A Framework for Differentiable Physics*. 2020. arXiv : 1912.04232 [physics.comp-ph].

[21] Jimmy Secretan et al. « Picbreeder : A Case Study in Collaborative Evolutionary Exploration of Design Space ». In : *Evolutionary Computation* 19.3 (sept. 2011), p. 373-403. ISSN : 1063-6560. DOI : 10.1162/EVCO_a_00030. URL : https://doi.org/10.1162/EVCO%5C_a%5C_00030.

[22] Software Freedom Conservancy. *Selenium*. Version 4.18.1. 20 fév. 2024. URL : https://www.selenium.dev/.

[23] three.js authors. *three.js*. Version r160. URL : https://threejs.org/.

[24] Wesley Unwin. *obj-file-parser-ts*. (accédé le 02/03/2024). URL : https://github.com/Deckeraga/obj-file-parser-ts.