Cellulart

Projet du cours Vie Artificielle

par Olivier FAVRE, Haykel HADDAJI, Yassin PATEL, Quentin PRADET

Lundi 4 Avril 2011

Table des matières

Introduction

Cellulart est notre contribution pour le projet de vie artficielle de l'option IAD 2011. C'est un logiciel libre disponible sur gitorious à l'adresse http://gitorious.org/cellulart/. Ce rapport présente la conception du logiciel, en allant des objectifs de conception à l'implémentation, puis discute à propos des agents implémentés.

Le terme « agent » signifie simplement les différents éléments que nous avions à implémenter : automate cellulaire, système multi-agents, L-système.

Objectifs

Avant d'implémenter Cellulart, il fallait se mettre d'accord sur un langage de programmation à utiliser, mais aussi sur les principes de conception qui allaient guider le développement.

Le premier objectif est la facilité d'utilisation : il faut que ce soit très facile d'ajouter un nouvel agent, de le modifier, bref, d'expérimenter ! C'est, selon nous, la principale qualité d'un tel projet, et c'est seulement grâce à de nombreuses itérations qu'on peut parvenir à un résultat correct. C'est aussi cette philosophie qui explique l'interface graphique intuitive permettant de gérer simplement les différents calques (affichage ou non, niveau de transparence) et permettant de mettre l'animation en pause ou d'en changer la vitesse.

Le second objectif est la modularité. Cet objectif découle naturellement du premier, étant donné que nous avons décidé de découper le programme en différents modules bien distincts et faiblement dépendants les uns des autres. Le programme est donc suffisament générique pour qu'il soit possible d'ajouter facilement de nouveaux « agents ». L'intérêt est de pouvoir très facilement déclarer de nouveaux agents, modifier leur comportement, et faire en sorte que le couplage entre ces agents soit extrêmement léger. Ce couplage faible nous permet d'avoir des interactions très naturelles entre les différents agents : il suffit de charger dynamiquer le code nécessaire à ces interactions au début de l'implémentation, et Cellulart s'occupe du reste.

Le troisième objectif est la performance. C'est important mais ne devait pas sacrifier les avantages des deux objectifs précédents. Le programme a été profilé, les goulots d'étranglement ont étés identifiés, afin de gagner suffisament de performances pour pouvoir avoir l'impression de temps réel dans les cas les plus courants, c'est-à-dire une dizaine d'agents sur une grille de 265 par 256 pixels. L'interface graphique a été modifé pour pouvoir choisir le taux d'affichage. Il est ainsi possible de laisser le programme calculer un certain nombre d'itérations avant d'afficher le résultat, ce qui est utile lorsque l'on veut avancer rapidement dans le temps.

Implémentation

Dans cette section nous allons détailler l'implémentation des diverses fonctionnalités du projet.

Coeur

Le code étant modulaire, nous allons commencer par décrire la partie commune et la partie gérant les modules.

Interface graphique

Commençons par l'interface graphique, l'endroit le plus facile pour comprendre le programme dans sa globalité.

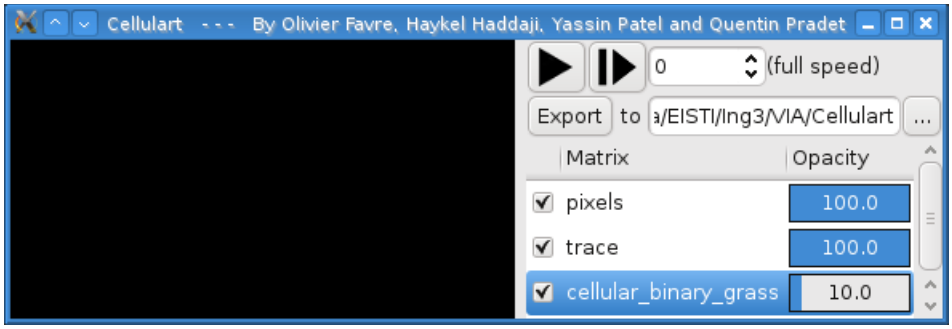

Figure 1. Interface graphique du projet

Elle est décomposée en 2 parties principales : à gauche la visualisation et à droite les contrôles. La visualisation est basée sur un composant OpenGL, ceci nous permet des manipulations et des compositions rapides des images, telles des zoom et déplacements.

Les contrôles sont eux décomposés en 3 parties.

La première ligne gère l'animation avec un bouton play, pause-step et une réglette de vitesse. La vitesse est spécifiée en millisecondes d'attente entre chaque calcul d'itération et d'affichage d'image, ou en millisecondes d'attente entre chaque affichage d'image, le calcul des itérations étant fait en continu, ou bien encore calcul en continu d'une image et affichage, le plus rapidement possible. Le premier mode permet de bien voir le détail de ce qui se passe, le second d'accélérer les calculs en perdant moins de temps à visualiser, le dernier permet de ne rien rater de ce qui se passe en allant le plus vite possible.

La seconde ligne permet l'exportation de la visualisation en fichier PNG et d'en choisir le dossier de destination. La dernière partie enfin est une liste des différents calques disponibles. Chaque cache est une matrice de valeurs transformée en pixels de couleur. Il est possible d'en gérer l'opacité ainsi que l'ordre d'affichage, par glisser-déposer.

Nous voyons donc ici le premier concept : l'image affichée est formée de différentes couches, chacune étant une matrice de données qui subit une conversion en pixels. Les pixels et les données sont effectivement totalement dissociés. Une palette de couleur est chargée d'en faire la conversion. Les palettes sont interchangeables.

Gestion des modules

Nous manipulons plusieurs composants dans notre projet : des matrices, des automates cellulaires, des systèmes multi-agents, des L-systèmes mais également des objets arbitraires et des moteurs de requêtes. Les systèmes multi-agents sont subdivisés en : états, percepts, actions et cerveaux. Les matrices possèdent plusieurs types de bouclage aux bords. Chacun de ces types de composants fait l'objet d'une ou de plusieurs classes, ainsi que d'une méthode d'instanciation.

Modules

Nous allons maintenant décrire plus en détail la manière dont chaque module fonctionne.

Matrices

Elles ont toutes la même dimension que le monde lui même, mais leur type de donnée peut varier du flottant au quadruplet d'entiers en passant par des objets Python.

Une matrice possède également des préférences de visualisation (opacité, visibilité, palette de couleurs), mais également un système de bouclage aux bords. En effet nous pouvons avoir envie que le monde soit torique, ou comme un anneau de Mobius.

Les matrices ont également un code d'initialisation, qui permet par exemple de placer certaines valeurs au centre du monde ou sur certaines lignes.

Elles représentent bien souvent une plateforme de support pour tous les agents. Lorsqu'une information est beaucoup plus éparse par contre, on utilisera des objets.

Automates cellulaires

Les automates cellulaires eux se branchent sur une ou plusieurs matrices qu'ils peuvent lire ou écrire. Typiquement un automate cellulaire est couplé avec une et une seule matrice dont il gère exclusivement la mise à jour. Beaucoup de calcul matriciel est utilisé ici via NumPy pour optimiser les traitements comme le calcul du nombre de voisins et l'écriture des nouvelles valeurs en fonction de tests booléens. Ceci a beaucoup contribué à l'accélération des traitements, mais les a aussi complexifiés en les rendant moins explicites.

Systèmes multi-agents

Ce sont des composants très complexes. Chaque agent d'un tel système possède des états, des percepts et des actions disponibles, ainsi qu'un ou plusieurs cerveaux, centres de décision.

Un agent possède une collection prédéfinie d'états, qui sont initialisés lors de sa naissance, suivant le code donné dans le module représentant ledit état. Les états d'un agent est en réalité un dictionnaire qui peut être très dynamique en taille comme en type de valeurs.

Les percepts sont des modules qui sont chargés de retourner une valeur particulière, mesure d'une perception de l'agent. Un percept dépend donc des états de l'agent, mais également du monde qui l'entoure, il a donc accès à tout cela.

Une action est très similaire à un percept dans son implémentation, mais est habité d'un rôle opposé. Une action peut changer un état ou modifier le monde, alors qu'un percept ne doit se contenter que d'y accéder en lecture.

Si des comportements sont indépendants les uns des autres, ils pourront alors sans problème être implémentés dans plusieurs cerveaux différents. Dans le cas contraire, moyennant une communication entre cerveaux basés sur des états particuliers, il est également possible de séparer des comportements génériques, bien que cela soit plus difficile.

Objets

Des objets arbitraires peuvent être créés, supprimés et modifiés. Le monde possède une liste d'objet d'un type donné – un simple nom – que l'on peut récupérer.

Les objets sont notifiés de leur création, leur suppression mais également du fait qu'ils n'aient été laissé tranquilles. Ceci permet par exemple de tracer un pixel coloré à une certaine position et de le cacher à leur suppression.

Les objets permettent de stocker une information arbitraire, mais également des objets censés être positionnés dans le monde, sur une matrice, mais dont la quantité ne justifie par une représentation aussi dense en information et difficile à chercher.

Requêtes

Le moteur de requêtes sert à effectuer un prétraitement unique par itération après la mise à jour des automates cellulaires et avant celle des systèmes multi-agents, puis doit répondre à de multiples requêtes, en général une par agent.

Ce système permet d'améliorer les performances en optant encore une fois pour un point de vue plus macro du problème en construisant une structure optimisée de recherche par exemple, afin de perdre moins de temps à répondre à chaque requête unique centrée sur le point de vue micro d'un l'agent. Il était également nécessaire de suivre le fil des itérations, chose qui n'était faisable qu'en créant un véritable moteur de requêtes.

Agents implémentés

Jeu de la vie

Le jeu de la vie est un grand classique que nous avons pris plaisir à réimplémenter. Il n'y a rien de particulier à expliquer concernant son implémentation : nous mettons à jour la matrice en utilisant les règles dictées par Conway. Nous avons d'abord essayé de faire fonctionner un plan-

neur pour vérifier que le comportement était correct, puis nous avons décidé de faire davantages d'expérimentations.

Tout l'intérêt du jeu de la vie réside dans l'état initial. Nous avons essayé des états initiaux aléatoires, et il s'avère que ce n'était pas très intéressant. Nous obtenions, après un certain nombre d'itérations, toujours les mêmes motifs. Soit des carrés qui restaient stables, soit des « lignes » qui suivaient une rotation de 90° à chaque itération. Nous avons cependant trouvé un motif intéressant : à partir de deux « carrés » imbriqués, nous avons découvert une suite de sortes d'arabesques remarquables. Peu à peu, l'image se stabilise, avant d'arriver sur un cycle infini.

Ce qui est intéressant ici c'est de voir à quel point de règles simples et un état initial plutôt simple mène à des comportements dits « complexes », mais a priori pas compliqués.

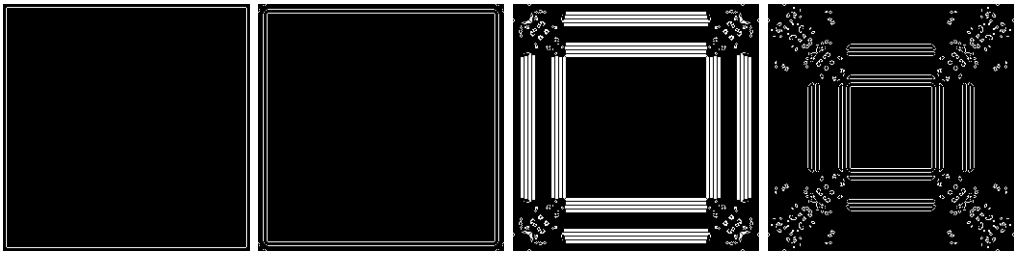

Figure 2. Premières itérations. Alors que des bandes se propagent vers le centre, des fluctuations se créent dans les coins, initialisant l'arabesque avec des motifs symétriques.

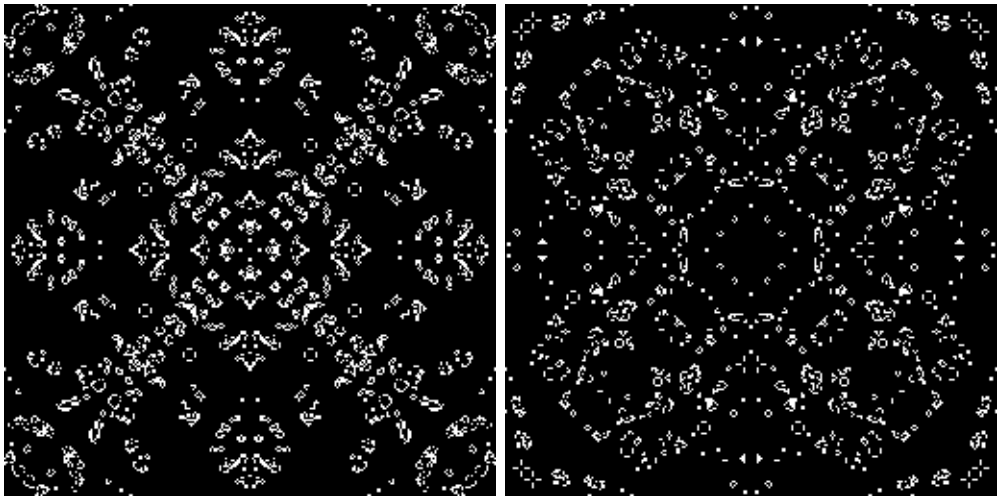

Figure 3. Deux des premiers états jolis de l'arabesque

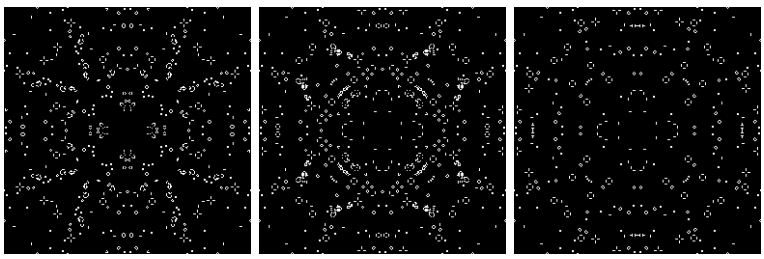

Figure 4. Décadance et dernier état bouclant (période 15).

Nous avons remarqué moins de monotonie dans les états avancés pour des taille de grille non puissance de 2.

Flocking birds

Nous avons souhaité poursuivre cette expérience en implémentant un autre système simple. Le principe ici est de simuler le comportement d'une « nuée » d'oiseaux, comme on a pu le voir en démonstration en cours. Le principe, communément admis, est plutôt simple : il suffit de trois règles pour implémenter ce comportement. C'est donc un choix attirant : facile pour la première réalisation, un certain nombre de paramètres à étudier, et un résultat amusant. Les trois règles, par ordre d'importance, sont :

- Éviter les oiseaux proches (pour ne pas les cogner) ;
- Suivre la direction des oiseaux autour (pas que les plus proches) ;
- Aller dans la même direction générale qu'eux.

L'implémentation est donc simple : si on est trop proche d'un ou plusieurs oiseaux, on les évite. Sinon, on peut se concentrer sur le fait de suivre les oiseaux aux alentours. Le comportement obtenu est assez réaliste et on pourrait vraiment avoir l'impression de se retrouver dans une nuée. Malheureusement, c'est un comportement qui n'est pas facile à montrer dans un rapport, et on devra ici se contenter des traces laissées par nos oiseaux.

Il est amusant de constater qu'avec deux oiseaux, c'est un cercle entier qui est réalisé. En effet, la règle d'évitement ne se déclenche pas, et la direction moyenne est défini par un angle constant de rotation. Dans le code, on demande aux agents de se tourner vers la moyenne des positions, ce qui donne toujours le même angle où qu'on soit dans le cercle.

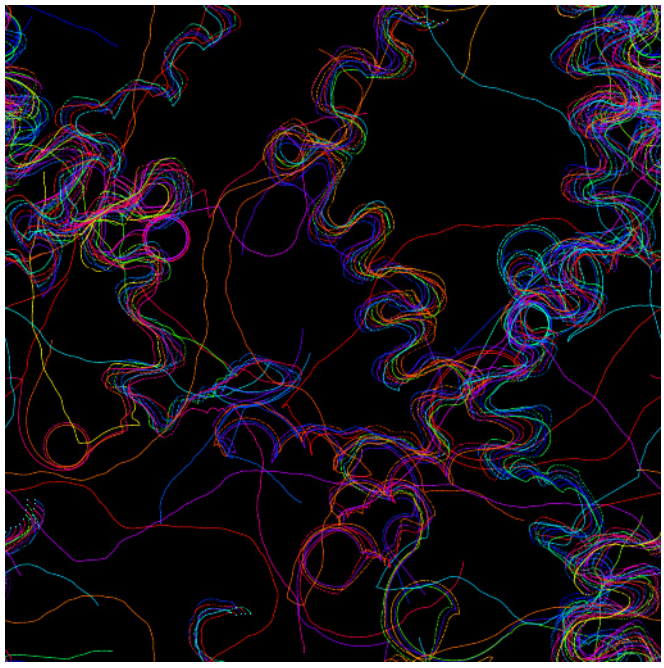

Figure 5. Flocking birds

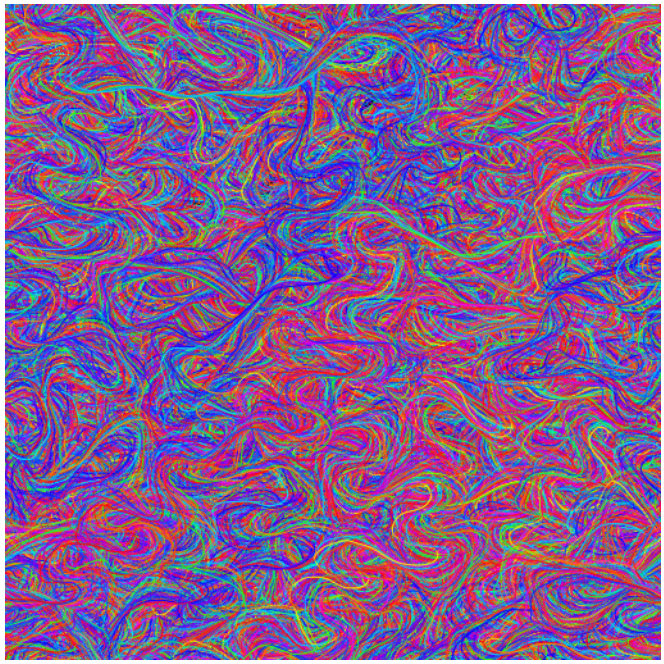

Figure 6. Après beaucoup d'itérations

Ce comportement émergent est en quelque sorte un équilibre instable : dès qu'un agent vient perturber nos deux agents, le cercle est cassé et ne se reconstruira plus. On obtient au contraire des sortes de zigzag que l'on aurait pas pu prédire autrement, et qu'il aurait été impossible de deviner en voyant à chaque instant les positions des oiseaux : seule la trace le révèle.

Encore une fois, c'est précisément ce genre de comportement inattendus et passionants auxquels nous nous attentions avec ce travail : comment produire des comportements complexes avec des règles simples.

Vaches et herbe

Afin de tester un autre automate cellulaire de notre conception et le moteur de requêtes, ainsi que la recherche locale dans des matrices, nous avons implémenté une autre démonstration. Cette démonstration était également l'occasion de créer une intéraction entre un automate cellulaire et les systèmes multi-agents

Premièrement, l'automate cellulaire a été pense de façon à faire des formes rondes, et pleines. Ainsi une cellule survit si au moins 3 de ses voisins sont vivants, et une cellule naît si au moins 5 de ses voisins sont vivants. Ceci fait des petites « tâches d'huile », que nous avons plutôt interprété comme des touffes d'herbe. Cet automate cellulaire est très très stable, et il faut un minimum de 4 cellules en carré pour qu'un paquet survive.

Sur cette prairie, des vaches viennent brouter. Si elles voient de la nourriture dans un rayon de 4 cases, elles s'y dirigent. Dans le cas contraire elle cherchent à se diriger vers la zone la plus dense en herbe dans un rayon de 4 zones de 8 cases sur 8 cases (détails plus loin). S'il n'y a pas d'herbe en vue dans ce périmètre, la vache avance aléatoirement. De plus, dans tous ses déplacements, la vache s'écarte un peu de ses voisines dans un rayon de 5 cases, sans pour autant aller faire des virages à 180° car elle se limite à des ajustements de 30° par itération.

Ainsi, une vache effectue une recherche locale dans une matrice, en recherchant la case la plus proche possédant de l'herbe. Ceci est fait par un percept.

Mais une vache effectue également une recherche de *densité*. Cette dernière a été implémentée en réduisant par 8 la taille de la matrice d'herbe, de manière à délimiter des zones de 8 cases par 8 cases dont la valeur est le nombre de cases d'herbe vivantes dans la zone. Cette étape n'est réalisée qu'une fois par itération, grâce au moteur de requêtes. Après réduction, chaque vache va réaliser une recherche locale pour la zone avec la plus grande valeur dans un rayon de 4 zones. Cette recherche est plus semblable au percepts décrit précédement, mais se fait au sein du moteur de requête. Il est important de noter que la résolution de la réponse retournée est limitée à la taille des zones décrites, dont on retourne le centre.

La réalisation fonctionne comme prévu, mais à cause des faibles ajustement de direction, il n'est pas rare qu'une vache passe juste à côté d'une case d'herbe, ou se rapproche beaucoup plus de sa voisine que ce que le rayon de recherche suggérait. D'un autre côté, la diminution de la densité d'agents autonomes est difficile, surtout quand tous les agents se dirigent vers la même direction !

En rajoutant aux vaches une trace qui s'évapore un peu, on remarque qu'elles redessinent des bouts des zones qu'elles broutent, en s'attaquant par les bords. En effet, le centre d'une touffe repousse instantanément, et la vache n'a aucune raison de tourner dans un sens plutôt que dans un autre. De ce fait les vaches passent plus de temps à revenir aux bords des touffes qu'elles viennent de traverser.

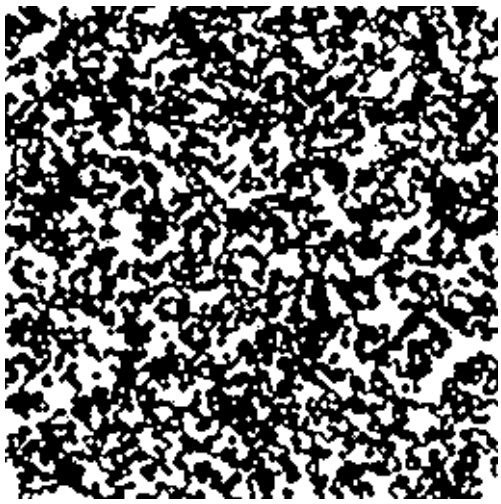

Figure 7. Automate cellulaire de fond, gérant la pousse de l'herbe.

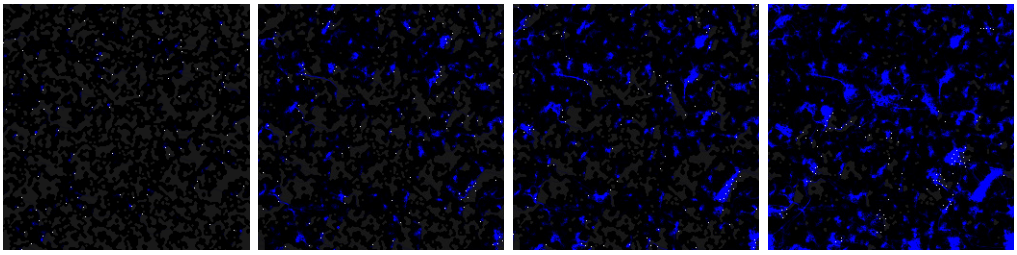

Figure 8. Évolution du système. Très peu d'herbe subsiste sous les couches bleues.

Figure 9. Concentration des vaches aux rares endroits abondants en herbe.

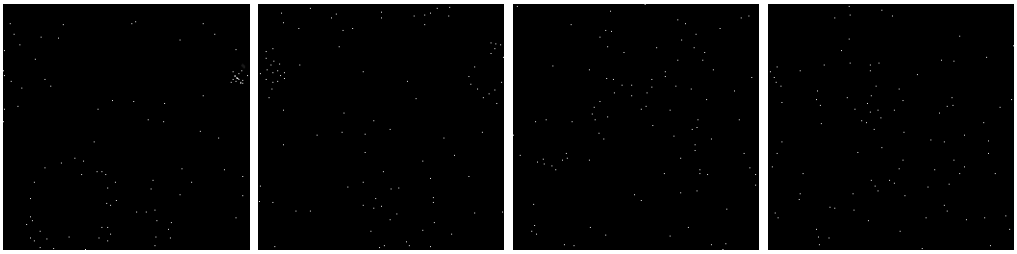

Figure 10. Dispersion finale des vaches quand toute l'herbe a été mangée.

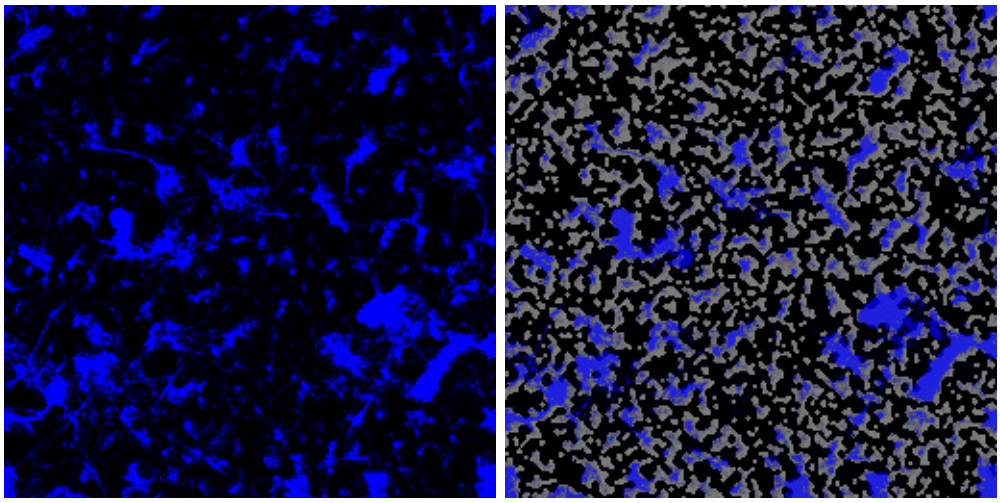

Figure 11. Trace finale laissée par les vaches, quand il n'y a tout juste plus d'herbe.
Et superposition avec l'image originale, pour comparaison.

Gardes, voleurs et trésors

Une des premières et des plus jolies démonstrations que nous avons implémentées simule un mode avec quelques trésors qui attirent les convoitises de quelques voleurs, mais surveillés par un bon nombre de gardes.

Les trésors sont des objets dont la valeur se limitent à leur coordonnées, car il sont censés être placés dans le monde. Les fonctions de trésors ne font que dessiner ou cacher le pixel coloré jaune qui marque leur position. Le fait que leur valeur soit égale à leur coordonnées nous permet surtout de pouvoir faire une recherche locale.

Un voleur se dirige, lentement, en marchant, vers le trésor le plus proche. Il se ballade aléatoirement si aucun n'est en vue dans un rayon de 50 cases. Une fois que le voleur passe sur le trésor, il le ramasse – le supprime du monde – et prend les jambes à son coup pour entâmer une folle course poursuite ! Un voleur en fuite cherche à éviter les 5 gardes les plus proches de lui, dans un rayon de 20 cases. Il évite leur position moyenne, pondérée par leur rapprochement.

Les gardes quant à eux marchent aléatoirement, au milieu des passants – les voleurs qui ont encore les mains vides (distinction faite en observant l'état des agents voisins). Lorsqu'un garde aperçoit un voleur, dans un rayon de 50 cases, il fonce vers lui pour le rattraper. S'il en voit plusieurs il se dirige vers le plus proche. Si le garde se trouve à moins d'une case du voleur, il lui prend le trésor (en modifiant l'état du voleur qui se remet alors à marcher). Une fois qu'un garde a confisqué un trésor, il trotine et cherche à s'éloigner le plus possible des passants – des voleurs potentiels – et dépose le trésor (en recrée un nouveau avec ses nouvelles coordonnées) si personne n'est à moins de 50 cases. Une fois le trésor reposé, il recommence à marcher et peut recommencer à scruter les horizons pour voir un éventuel voleur.

Afin de faire un joli rendu, nous avons attribué une couleur à chaque garde, et nous avons fait en sorte qu'il laisse une trace colorée lorsqu'il se met à courses un voleur. On observe alors un regroupement de plusieurs lignes lors du début d'une course poursuite. Lors de virages serrés (lorsqu'un garde libre s'interpose devant un voleur) les lignes se déserrent. Il arrive également qu'une ligne passe d'un faisceau à un autre, quand deux courses poursuites se croisent.

Nous avons rencontré un problème lors de cette démonstration : les agents ne pouvant pas réguler leur vitesse, certains se retrouvaient à tourner sans cesses autour du même point. Par exemple un voleur ayant un mauvais angle d'arrivée vers un trésor (ne venant pas de tout droit), va se mettre à tourner autours, comme s'il gravitait. Quelques fois, et c'était plus embêtant à cause de la trace, les gardes et le voleur tournaient en rond sur eux mêmes, souvent jusqu'à ce qu'un nouveau garde vienne « casser la boucle ».

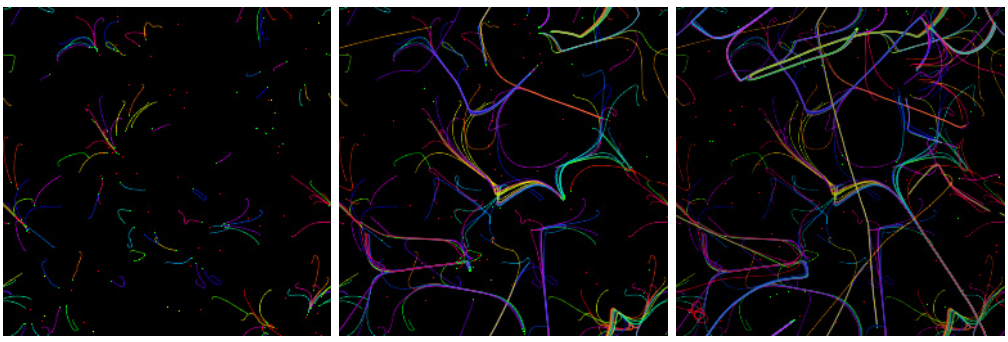

Figure 12. Début du système.

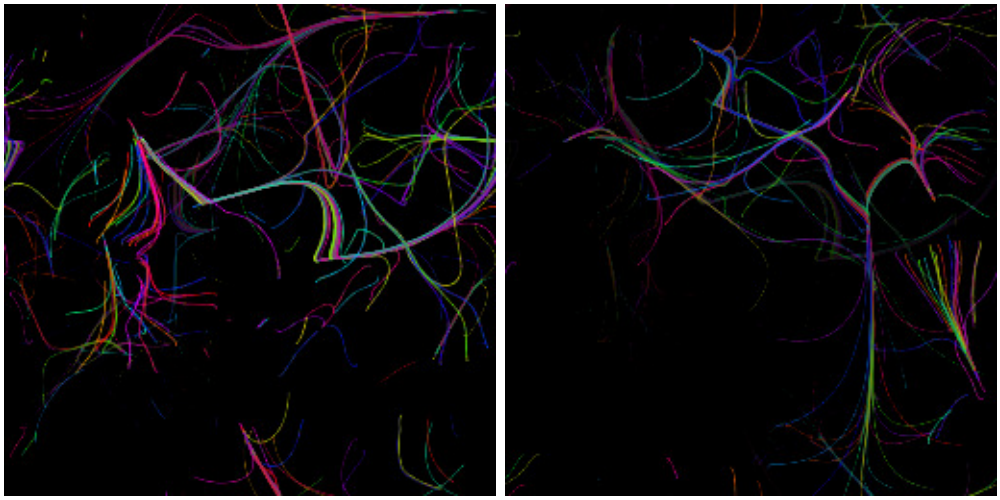

Figure 13. Au bout d'un certain temps. Les traces s'évaporant, le système ne parrait jamais vieux.

Variation de paramètres

Flocking birds

Nous avons choisi de commencer l'étude des paramètres avec les oiseaux parce que des réglages ont été faits avant d'obtenir un résultat. Contrairement aux autres programmes, nous avions une idée du résultat, et il était possible d'itérer jusqu'à être satisfait.

Il y a deux types de paramètres sur lesquels nous avons influés : la rotation maximum et les rayons de vision de nos oiseaux.

La rotation maximum exprime le tournant qu'un oiseau peut réaliser pour satisfaire une contrainte. Par exemple, pour s'éloigner des oiseaux trop proches, il est interdit de faire un demi-tour complet.

Le rayon de vision intérieur détermine les oiseaux qui sont considérés comme gênants et le rayon de vision extérieur détermine les oiseaux considérés comme étant à suivre.

La rotation maximum a été limitée pour éviter les retours intempestifs en arrière qu'un oiseau réel est incapable de faire. Il a fallu autoriser un angle plus important pour les répulsions locales : c'est une réaction plus forte, destinée à rétablir l'ordre. La rotation qui fait aller vers la position moyenne est plus faible, mais pas trop : celà doit correspondre à un tournant moyen pour un oiseau. La rotation pour suivre la direction moyenne est encore plus faible, pour ne correspondre qu'à un faible ajustement. Sans ces valeurs, les mouvements étaient insensés et n'avaient pas de réel correspondance.

Pour le rayon de vision, nous avons finalement décidé d'avoir un rayon compris entre 0.1 et 3 pour le rayon interne. Cela permet de ne pas se prendre en compte soi-même et d'éviter les menaces locales. Un rayon entre 3 et 20 s'est avéré satisfaisant pour le rayon externe. Celà n'empêche pas que de groupes plus gros se forment, mais il arrive que ces groupes se cassent et se reforment, ce qui semble extrêmement naturel.

Il est très difficile d'expliquer ceci en image, étant donné que ce sont de comportements dynamiques, qu'il faut voir en mouvement, et que nous étions satisfaits lorsque ça "avait l'air" bien, sans pouvoir l'exprimer.

Enfin, la modification de paramètre la plus amusante aura été le fait de changer la vitesse lors de la répulsion locale pour permettre aux oiseaux de s'échapper le plus vite possible. Cela donne l'impression que nos oiseaux deviennent des poissons, qui sont capable tout d'un cop d'accélerer pour rester dans le banc ou éviter un prédateur.

Gardes, voleurs et trésors

Regardons plus en détail l'astuce permettant d'éviter des boucles infernales.

Lors de la course poursuite, il ne faut surtout pas dépasser le voleur, sinon il va se mettre à tourner très fortement. Pour éviter cela, il faut que le garde décelère lorsqu'il est à une case du voleur (étant donné que le garde en course poursuite avance d'une case par itération). Nous avons décidé de le faire avancer de la distance qui le sépare du voleur, moins une certaine quantité fixe, le tout restant scrupuleusement borné entre 0 et 1. On a donc :

$$speed = \min\left(1.00,\ \max\left(0.00,\ distance - \varepsilon\right)\right)$$

Voici les différents résultats obtenus pour des valeurs de ε comprises entre 0 et 1 :

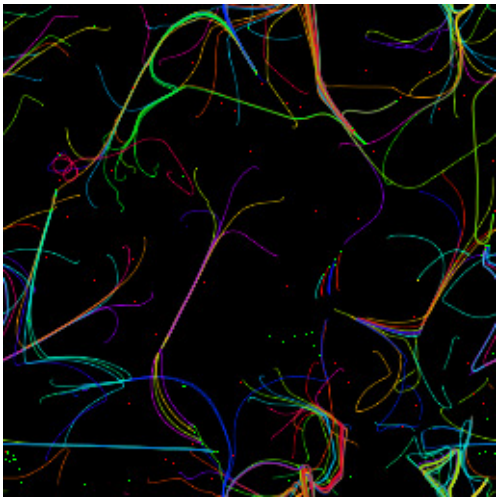

Figure 14. $\varepsilon = 0.01$. Valeur par retenue.

On remarque peu de virages, seulement un à gauche à $1/3$ de hauteur.

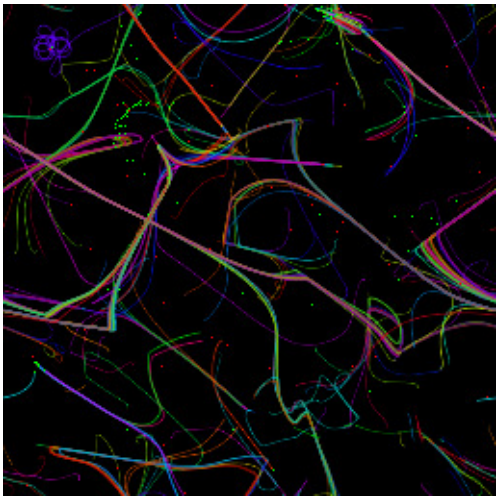

Figure 15. $\varepsilon = 0.00$

On voit plus de virages serrés, et un ensemble de tortillons en haut à gauche.

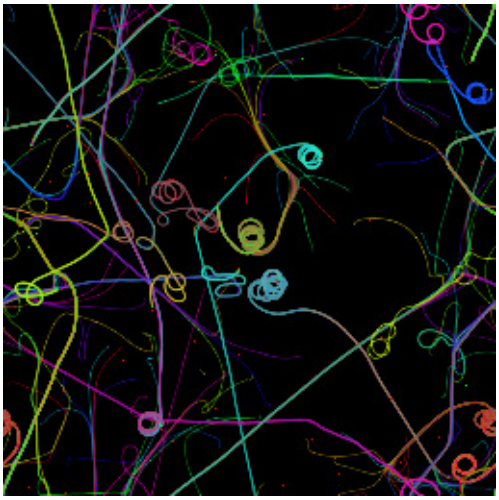

Figure 16. $\varepsilon = 0.5$

Il saute au yeux clairement que les virages ont très souvent tendence à finir en boucles infernales.

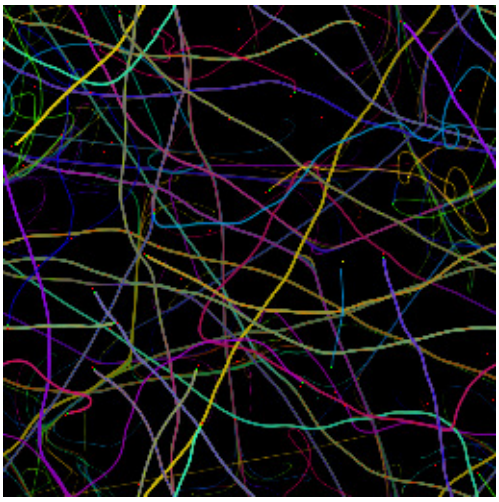

Figure 17. $\varepsilon = 0.9$

Pour des valeurs élevées de ε, les courses poursuites ne s'achèvent plus, et les lignes ont très peu tendance à faire de virages.

Au final nous avons retenu $\varepsilon = 0.01$ car cette valeur avait une bonne tendance à casser les boucles (sans pour autant les supprimer totalement malheureusement) et ne cassait pas trop l'aspect aléatoire des courbes.

Conclusion

Si nous n'avons pas spécialement mis l'accent sur la beauté des images générées, nous avons surtout cherché à créer une architecture extrêmement modulaire et riche en fonctionnalités, tout en gardant de bonnes performances malgré l'utilisation d'un langage interprété.

Notre projet est libre et disponible sous licence BSD. Nous espérons que cette disponibilité libre ainsi que la modularité et la multitude de fonctionnalités intéresseront d'autres personnes à la recherche d'une plateforme de simulation ou d'art numérique !