Rendu de fin-parcours
Projet Grain de Cell

Thomas Sibut-Pinote, Vincent Cohen-Addad, Ann Johansson, Lucca Hirschi,
Antoine Plet et Guillaume Lagarde

28 juin 2017

Résumé

Notre projet est un simulateur d'organismes vivant dans un univers persistant. Ces organismes sont composés de briques de bases, les cellules, qui possèdent un ADN.

Les utilisateurs ont accès à une interface de création d'organismes, sur laquelle ils peuvent programmer leur ADN dans un langage simple mais expressif. Ensuite, ils peuvent "injecter" leurs créations dans l'univers, et observer leur évolution vis-à-vis du milieu et des autres espèces, sans possibilité d'interagir avec elles.

Ce document constitue le rapport final du projet, il décrit principalement les réalisations techniques du projet.

Table des matières

1 Objectifs

Voici les objectifs pour le produit final que nous avions fixés :

1. **Un langage facile mais expressif :**
   Le langage se doit d'être facile d'accès pour que l'utilisateur puisse facilement créer des cellules. Mais également simple dans le sens ou l'on doit — à la manière des légos — proposer des briques de bases modulables à l'infini, afin de voir émerger des cellules aux comportements complexes, malgré l'aspect élémentaire des actions disponible.
   L'un des autres intérêt de forcer l'utilisateur à coder dans un langage entièrement créé par nous-même et de l'empêcher de hacker le système car nous avons un contrôle absolu sur ce langage (par exemple, s'il pouvait coder directement en C++, il pourrait avoir accès à des zones mémoires/des variables/etc... qui pourrait faire planter le serveur, ou pire) ;
2. **Un compilateur pour ce langage** :
   Le compilateur doit transformer un code de notre langage CEL (dont les objectifs se trouvent ci-dessus) en un code C++ qui effectue les actions attendues. En outre, le compilateur doit également relever les erreurs dans le code CEL, et les renvoyer à l'utilisateur, lui permettant de les retrouver facilement et de les corriger ;
3. **Un environnement capable d'ajouter des cellules dynamiquement :**
   On doit pouvoir stocker le monde avec ses cellules et le faire évoluer. Il faut que les choix d'implémentations permettent modularité et ajout de fonctionnalités (pouv pouvoir ajuster les règles du jeux). On ne veut pas devoir recompiler l'environnement à chaque fois que l'on veut intégrer une nouvelle cellule. L'environnement se doit de gérer dynamiquement l'ajout de cellule car à terme, un serveur doit faire tourner en continu cet environnement, ou des utilisateurs pourront rajouter leur cellules à la volée ;
4. **Un serveur qui distribue la carte et reçoit de nouvelles cellules :**
   Important car l'on veut pouvoir mettre en place un aspect compétitif au jeu. Il faut dont pouvoir mettre en place un serveur où les utilisateurs se retrouveront afin de faire concourir leurs cellules dans une jungle impitoyable d'adversaires. Ce qui pousse l'utilisateur à faire des cellules complexe et élaborer des stratégies,..., et c'est là tout l'intérêt de grain de cell !
5. **Une interface graphique :**
   L'objectif de l'interface est d'être assez fluide pour pouvoir regarder les cellules évoluer sans phénomènes de lattences par exemple ; mais elle doit être assez jolie pour donner à l'utilisateur l'envie de jouer au jeu. Un compromis est à trouver !

2 Commentaires sur l'organisation du travail

Nous travaillons à l'aide de l'outil `git`, notre dépôt est hebergé sur gitorious. Nous avons construit un site internet dont le public visé rassemble les utilisateurs et de potentiels contributeurs. Voici les différentes adresses :

— le dépôt git : `http://gitorious.org/cellulite`
— le site internet : `http://graal.ens-lyon.fr/graindecell/`
— la liste de diffusion interne `projectcell@listes.ens-lyon.fr`
— la liste de diffusion publique `http://groups.google.com/group/gdc-public`

3 Travail achevé

Pour l'organisation du temps, on se référera au diagramme de Gantt donné en figure 6. On présente pour chaque *work-package* le travail réalisé à mi-parcours puis en fin de parcours.

3.1 Théorique

Mi-parcours *Membres : Thomas Sibut-Pinote, Guillaume Lagarde et Lucca Hirschi.*
Nous avons produit le document `regles.tex` dans `Document/regles_langages/`. C'est un document interne qui a pour vocation de fixer une fois pour toute les spécifications de la simulation (« règles du jeu », définition du langage, etc.). Une version pédagogique de ce document fera office de « mode d'emploi » pour l'utilisateur.

Il contient les règles « physiques » régissants la simulation, la définition des cellules (attributs, actions possibles) ainsi que la grammaire du langage qu'utiliseront les utilisateurs pour coder le comportement de leurs cellules. Des exemples de codes d'ADN sont ensuite donnés : ils permettent d'avoir de bons exemples d'utilisation du langage et de se rendre compte, qu'il est possible de coder des comportements compliqués en quelques lignes. En l'état, ces exemples sont à corriger car le lexer du compilateur a été traduit en anglais et la grammaire a été légérement modifiée il y a une semaine. Enfin, nous définissons le « but pour l'utilisateur » à savoir comptabiliser le meilleur score. Nous expliquons comment ces scores sont calculés.

Fin de parcours *Membres : Lucca Hirschi, Antoine Plet, Thomas Sibut-Pinote et Guillaume Lagarde*

Des modifications ont été apportées au langage nouvellement appelé `CEL` tout au long du semestre. Certaines nous sont apparues nécessaires pour faciliter la compilation, d'autres pour rendre l'utilisation de Grain de Cell plus agréable au programmeur (ceci rassemble d'infimes modifications syntaxiques et des révolutions conceptuelles...). Plusieurs des modifications de la seconde catégorie facilitaient la prise en charge de notre langage par le *tuareg-mode* de l'éditeur Emacs, ce qui le rend beaucoup plus facile à utiliser !

Refonte des messages Nous avions laissé en suspens les problèmes de directions et de messages qui nous étaient apparus à cause de la compilation. Nous avons entièrement revu cela, et la solution choisie donne entière satisfaction : elle marche, et de plus elle est très simple et intuitive à manipuler.

Les messages sont à présent des entiers comme les autres (d'ailleurs ils n'existent que d'un point de vue sémantique, rien ne les distingue dans le compilateur), ainsi que les vecteurs. Comme expliqué dans la partie compilation, une direction est en fait un vecteur de directions "caché", c'est à dire que son i^{me} bit dit si la direction i est sélectionnée. Pour plus de détails sur les messages, voir le *work-package* « Compilation » 3.6.

Traitement des messages : Très tôt dans le WP théorique nous était apparue la nécessité de fournir un traitement souple des messages, qui permette en le moins de tours possibles d'apporter une réponse satisfaisante à une surcharge de la boîte de réception. Un cas typique qui nous est venu à l'esprit est celui d'une cellule recevant des messages de demande urgente d'énergie pour résister à des assaillants, il s'agit de se rendre compte tout de suite quelles sont les directions concernées et quelle réponse doit être apportée. La version finale de ce traitement est de la forme

```
if Exists(var1,var2,Expr)
then Prog1
else Prog2
```

Lorsque cette commande est appelée, la condition "il existe un message **var1** tel que Expr" est testée sur tous les messages, et le vecteur "**var2**" est affecté à 1 pour toutes les directions qui la vérifient. **var1** et **var2** restent alors définies dans l'exécution de Prog1 pendant le tour suivant (plus de détails dans la documentation). Ceci fournit au programmeur un précieux sucre syntaxique, à la manière du "let in" de Caml, et que notre expérience de programmeurs `CEL` confirme comme très utile.

Petites modifications du langage :

— Ajout de la possibilité d'accéder au i^{me} bit d'un entier x grâce à l'expression $x[i]$ (contre $x[i,i]$ précédemment).

— Redéfinition du lexème signifiant l'exécution séquentielle sur plusieurs tours : &&

— Ajout des primitives de directions : NextDir, OpDir, PrevDir qui calculent respectivement la direction suivante (dans le sens horaire), la direction opposée et la direction précédente de leur argument.

Nous avons également écrit une documentation complète (utilisation, règles du jeu, spécifications du langage CEL, etc...) : voir le *work-package* « Communication » 3.4.

3.2 Environnement

Le travail de ce *WP* est concentré dans les fichiers du dossier `serveur/environnement/`.

Mi-parcours *Membres : Lucca Hirschi et Antoine Plet*

Nous avons implémenté les structures de positions, buffers de messages, cellule et en partie celle de l'environnement pour permettre de lancer le serveur avec un environnement qui évolue. L'environnement est capable d'intégrer de nouvelles cellules à partir d'une librairie dynamique ainsi que d'envoyer les données d'affichage dans un flux (`cout` ou fichier ouvert par exemple). Chaque classe est implémentée dans un fichier qui lui est propre. Les fichiers concernés se trouvent dans `/serveur/environnement`. Nous avons écrit quelques codes de celulles en C++ (qui seront produits automatiquement par le compilateur quand il sera achevé). Ce sont les fichiers `code_compile.cpp` et `code_compile2.cpp`.

Pour l'instant, seules les méthodes SendMessage et Duplication implémentants les actions « envoie de message » et « duplication » ne sont pas encore écrites. Pour tout le reste, l'environnement est capable de faire évoluer le monde et ses cellules.

Fin de parcours *Membre : Lucca Hirschi*

L'environnement a dû être complété en concertation avec les autres *work-package*, patché et maintenu. Dans la continuité du travail déjà effectué, l'environnement est bien segmenté en classes/objets (un pour la mémoire des cellules, buffer des messages, système de position et terrain, un objet pour la cellule, l'environnement contenant des cellules, etc...) et gère la temporisation (on ne doit pas privilégier des cellules en donnant un ordre d'exécution à chaque tour [1]). D'autre part, les constantes de la physique du jeu on été ajusté atout au long des différents tests.

1. Globalement, la solution adoptée est de *bufferisé* chaque modification. Par exemple les modifications du niveau d'énergie de chaque cellule sont appliquées seulement quand toutes les cellules ont agi.

Actions de cellules et tests En priorité, les méthodes d'actions manquantes de l'objet cellule ont été implémentées (envoie de message et duplication). Il a fallu tester la robustesse de l'environnement en le faisant tourner longtemps avec des cellules différentes (codes écrits en `C++` à l'époque) pour détecter toute fuite mémoire ou bugg. Quelques *benchmarks* ont également permis d'évaluer les performances de l'environnement. Ces tests peuvent êtres reproduit en laçant `make all` dans `serveur/environnement/`.

Authentification En coopération avec le *work-package* « Connexions réseaux » qui s'est occupé de l'authentification des clients, une partie de cette authentification a été intégré à l'environnement. Au sein de cet environnement, il existe des structures qui décrivent les utilisateurs et leurs familles de cellules.

Scores et suppression de familles En coopération avec le *work-package* « Interface », nous avons ajouté les scores à l'environnement. A chaque tour, les scores sont calculés pour chaque joueur et chacune de ses familles. La formule calculant ces scores peut-être modifié très simplement dans `env.cpp`.

Des méthode de l'objet environnement permettant de supprimer entièremet une famille de cellule et de donner un numéro de famille libre pour un joueur ont été implémentées.

Sauvegarde et chargement Dans l'objectif de rendre la phase de beta-test possible, il a fallu ajouté une fonctionnalité de sauvegarde/chargement de l'environnement. La solution adoptée est pragmatique : la sauvegarde enregistre dans un fichier la position, l'état et le code source d'une cellule par famille pour chaque joueur et rétabli ces cellules au chargement. Le serveur enregistre l'environnement à chaque ajout de cellules. Nous pouvons ainsi stopper le serveur à tout moment, le patcher et le recompiler puis relancer le serveur avec un environnement proche de celui qui existait auparavant.

Gestion et journal des erreurs Toujours en souciant des beta-test, l'environnement rattrape ses propres erreurs et les décrit de façon précise dans un journal des erreurs consultable pendant que le serveur tourne. Ainsi, si un code de cellule `C++` tourne avec une gestion erronnée de la mémoire, les *segfault* sont rattrapés et suivis. L'environnement ne plante pas (les autres utilisateurs peuvent continuer à jouer) et nous pouvons patcher le serveur/environnement.

3.3 Connexions réseaux

Mi-parcours *Membres : Antoine Plet et Vincent Cohen-Addad*
Nous avons réalisé un serveur qui fait évoluer l'environnement tout en envoyant les données d'affichage aux clients connectés. Pour se faire, nous avons choisi de répartir les différentes taches entre plusieurs threads à l'aide la librairie `pthread`. Ce serveur est ainsi capable de recevoir des fichiers envoyés par un client (code de cellule) tout en lui envoyant les informations inhérentes à l'environnement. D'autre part, nous avons implémenté deux programmes pour le client : un programme qui se connecte au serveur puis reçoit les données que le serveur lui envoie et un autre qui envoie un fichier au serveur. Actuellement, nous sommes donc en mesure de lancer le serveur qui fait évoluer l'environnement en permanence, accepte les connections et envoie les données d'affichage et parallèlement, des clients qui se connectent et reçoivent les données d'affichage et qui envoient un fichier au serveur.

Fin de parcours *Membre : Vincent Cohen-Addad*
Les différentes évolutions du serveur après la mi-parcours ont été élaborées en collaboration avec les workpackage interface et l'environnement. Bien entendu, nous avons intégré la compilateur dans le serveur pour que les cellules soient compilées dès leur réception. Ceci a été fait à l'aide de Syscall.

Authentification Afin de garantir un service fiable et sécurisé à nos utilisateurs nous avons décidé d'authentifier les participants. Ainsi nous avons chargé les threads destinés à l'envoi de la carte d'identifier les utilisateurs avant de leur envoyer la carte mais aussi d'enregistrer les nouveaux participants. Un nouvel utilisateur envoie sont login et son mot de passe qui seront ensuite stockés sur le serveur si le login est disponible. Bien sûr, le mot de passe est stocké au format SHA-1 pour prévenir toute usurpation en cas d'attaque.
Ainsi, les threads vérifient l'identité de l'utilisateur avant de lui envoyer la carte et il en va de même lors d'une reception de cellule. Par ailleurs, maintenir une liste de tous les utilisateurs dans un fichier nous permet de répertorier les cellules de chaque utilisateur.

Scores Comme précisé plus haut, nous avons implémenté un système de scores. L'environnement sauvegarde les scores avant que les threads ne diffusent aux clients connectés la table des scores par le même procédé que pour la carte.

Stabilité Pour proposer une expérience intéressante aux joueurs, nous avons travaillé sur la stabilité du serveur.
À l'aide de mutex, nous avons implémenté une version fiable du serveur en le préservant des deadlocks et famines. En outre, nous avons protégé le serveur d'éventuel dépassement de buffer lors de la réception des cellules en limitant la taille des fichiers transmis.
Enfin, un client cherchant à ralentir le serveur sera déconnecté.

Gestion de la latence Les échanges clients - serveur sont nécessairement soumis au trafic réseau. Nous avons choisi une connexion TCP pour assurer la transmission des paquets. Le découpage en frame par les sockets nous a conduit à envoyer le nombre de caractères à recevoir. Ainsi, pour chaque échange, le destinataire attend d'avoir reçu la totalité du message.

3.4 Communication

Mi-parcours *Membre : Ann Johansson*
Nous avons acquis un emplacement sur l'infrastructure de l'ENS pour le site web ainsi qu'un nom de domaine (`http://graal.ens-lyon.fr/graindecell/`). Le site web y a été installé ainsi qu'un forum.

Fin de parcours *Membres : Ann Johansson & Lucca Hirschi*
Il fallait en priorité créer du contenu pour le site Web. Nous avons commencé par écrire une description rapide de Grain de Cell avec des *screenshots* de l'interface pour le `Home` du site. Tout le contenu que l'on créer est simultanément traduit et ajouté sur la version anglaise du site.

Documentation Nous nous sommes alors lancé dans la rédaction d'une documentation complète (installation de Grain de Cell, règles du jeu, spécification complète du langage `CEL` présentée de façon pédagogique mais exhaustive et exemples de codes). Cette documentation est

ajoutée au site internet (en HTML directement consultable et en PDF). Cette documentation est également traduite.

Pour expliquer plus rapidement Grain de Cell nous avons également écrit un tutoriel qui se limite aux règles du jeu.

β-test Nous avons également communiquer pour le lancement de la phase de β-test. Nous avons mis en place une liste de diffusion publique (`http://groups.google.com/group/gdc-public`) pour communiquer avec les β-testeurs ainsi qu'une page (« Téléchargement ») sur le site annonçant la β-test et permettant de télécharger Grain de Cell. Tout cela deux semaines avant la présentation dans le but de présenter un produit testé.

3.5 Interface

Mi-parcours *Membre : Vincent Cohen-Addad*
L'objectif du *Work-package* Interface était de mettre en place une interface basique offrant la possibilité de visualiser l'environnement dans de très simples graphismes. Ceci pour nous permettre différents tests avant de concevoir l'interface finale.
Cet objectif est rempli ; il est possible d'observer en temps réel l'environnement en fonctionnement sur un serveur distant et il est aussi possible de lui faire ajouter des cellules pré-compilées, cf. figure 1.
L'interface a été conçue en python et utilise le module Tkinter pour la partie graphique. Celle-ci éxécute en sous-processus les programmes C chargés de l'envoie et de la reception des données par le réseau. L'échange d'informations entre les processus se fait par signaux ainsi que par un pipe entre les sorties standards.
La vitesse autorisée par les différents processus de l'interface est de 50ms.
Il est aussi possible d'ajouter simplement les fonctions d'authentification sur le serveur.

Fin de parcours *Membres : Vincent Cohen-Addad, Antoine Plet*
Pour concevoir une interface graphique séduisante, nous avons choisi d'utiliser une partie de l'interface Tkinter ainsi qu'un affichage 3D OpenGL.

Affichage de la carte On affiche donc un plateau sur lequel la carte est présente quatre fois pour mieux visualiser la structure torique et offrir une continuité sur les "limites" du terrain. Au centre, une carte est délimitée par un enclos pour permettre de repérer les répétitions du tore. Les cellules sont représentées par des demi ellipsoides dont la couleur dépend du propriétaire et de la famille et dont la hauteur représente le niveau d'énergie. L'ensemble est implémenté par trois classes C++ : une classe cEllipsoid pour dessiner une cellule avec certains paramètres (hauteur, position, ...), une classe cEye pour gérer la caméra et une classe cRenderer qui gère les deux classes précédentes et qui sera la seule utilisée par le client de réception. Pour l'affichage d'une cellule, la surface est paramétrée partitionnée en quadrilatères et en triangles. La classe cRenderer s'occupe quant-à elle de mettre à jour la carte et transmet les actions de l'utilisateur à la caméra, (voir figure 2). C'est cette classe qui s'occupe de l'affichage du plateau et de la couleur des cellules.

Interface utilisateur Nous avons décidé de réunir les différentes commandes utilisateur (Connexion, Déconnexion, Inscription, Envoi de Cellule, Suppression de Famille) dans l'interface Tkinter. Chaque commande lance en sous-routine un programme `C++` qui se connecte au serveur afin de réaliser l'action souhaitée. L'interface récupère ensuite le résultat de la sous-routine pour transmettre à l'utilisateur le résultat de sa requête.

Toutefois, pour la connexion, nous avons implémenté un sous programme chargé d'afficher l'interface graphique 3D et de transmettre à l'interface Tkinter l'évolution des scores.
Ce sous programme est un peu plus complexe car, pour avoir un affichage en temps réel couplé à une interface réactive au déplacement de la caméra par l'utilisateur, il est nécessaire de concevoir un programme multithreadé. Nous avons donc assigné un thread à la reception de la carte et des scores et un second thread à l'affichage de la carte en 3D. Le premier thread est aussi chargé de transmettre les scores à l'interface Tkinter.

Scores Pour que chaque joueur puisse suivre l'évolution de son score et le comparer aux autres participants, nous avons décidé d'envoyer à chaque joueur connecté le tableau des scores. Comme expliqué dans le paragraphe

ci-dessus, les scores sont transmis depuis le programme `C++` chargé de la connection avec le serveur jusqu'au programme `python`. Le programme `python` affiche le score des cinq familles de chaque joueur, tour par tour.

Les figures 3 et 4 présentent les interfaces python et OpenGL réunies.

Bac à Sable Dans l'idée de permettre aux joueurs de construire rapidement leur premier code de cellule et de faciliter leur progression, nous avons souhaité que ceux-ci puissent tester facilement et sans connexion leurs codes.
Nous avons donc développé cet outil, qui compile le code CELL vers le C++ puis du C++ vers une librairie dynamique avant de l'insérer dans un environnement vide soumis aux mêmes règles que l'environnement en ligne. Néanmoins, nous avons choisi d'accélérer le nombre de tours par minutes dans le Bac à Sable pour que l'utilisateur puisse voir rapidement l'évolution de sa cellule. La visualisation du Bac à Sable se fait par l'interface 3D en OpenGL présentée précédemment. La encore le programme est multithreadé. Le premier thread se charge de faire évoluer l'environnement pendant que le second thread s'occupe de l'affichage.

3.6 Compilation

Les fichiers sont dans le dossier `Compilation/`.

Mi-parcours *Membres : Guillaume Lagarde, Thomas Sibut-Pinote*
L'objectif du *Work-package* Compilation pour dans une semaine était d'être terminée. Le parser et le lexer sont finis (dans la mesure où ils transcrivent fidèlement la grammaire actuelle, cela reste sous réserve des modifications ultérieures que nous pourrions avoir à y apporter), et ils construisent ensemble un objet du type décrit dans `ast.ml`. Nous avons commencé trois tâches distinctes, réparties entre Antoine, Guillaume et Thomas. Ce sont respectivement l'analyse statique (vérification du typage, de l'absence d'appels récursifs, renommage des variables), la traduction de l'arbre de syntaxe abstraite vers un autre arbre contenant des informations plus proches du code `C++` final, et finalement la traduction proprement dite vers le *C++*. Les trois tâches sont en cours de réalisation, même si il nous paraît difficile de tenir nos engagements sur ce *Work-package*. Nous avons rencontré des difficultés qui nous ont forcé à modifier la grammaire, et nous pensons refondre prochainement la gestion des messages, trop compliquée à notre goût. Les appels de fonction ont été revus. Nous avons également ajouté la possibilité de lancer une séquence d'actions, qui devient bloquante (c'est-à-dire qu'à chaque fois qu'on se retrouve dans la fonction "mère" d'un tel appel, on saute automatiquement à l'endroit où s'était arrêté le calcul, sachant que l'on ne peut faire qu'une seule action par tour.

Fin de parcours *Membres : Guillaume Lagarde, Antoine Plet, Thomas Sibut-Pinote*
La compilation est "terminée" au sens où la totalité des objectifs que nous nous étions fixés pour la fin académique du projet ont été accomplis. Bien entendu, il reste constamment des choses à améliorer, dont certaines le seront parallèlement et/ou ultérieurement à ce rapport. Néanmoins, nous sommes fiers de pouvoir dire que notre compilateur est robuste et *user-friendly*. Ci-dessous, une description de ce que nous avons eu à faire à chaque étape du compilateur. Exhaustive, cette description serait aussi fatiguante qu'inutile, c'est pourquoi nous ne fournissons que les points clés et les difficultés techniques auxquelles nous avons dû trouver des solutions.

Analyse Statique L'analyse statique s'occupe de plusieurs type de vérifications importantes pour le fonctionnement du compilateur :

— *D'existence :* Appels de fonctions qui existent, utilisations de variables uniquement si déclarées et initialisées auparavant, vérifier qu'on utilise pas de fonctions primitives (comme *nrj* par exemple) qui n'existent pas (ou mal orthographiées etc.), qu'on ne fait que des appels de fonctions crées par l'utilisateurs, etc.

— *De typage :* Nous voulions tout d'abord mettre un typage des données pour séparer les types DIR (direction), INT et BOOL, mais finalement nous avons fait le choix d'être flexible et de tout coder sur des entiers, ce qui nous permet d'effectuer n'importe quelles opérations sur n'importe quelles données.
En revanche, la phase de typage doit s'occuper de vérifier l'arité des fonctions, des fonctions de coût, des calls et des duplications.

— *De récursivité et de dépendances :* Puisque chaque programme doit être terminant afin que chaque cellule s'execute en temps fini, on n'autorise pas les appels récursifs. Une vérification des dépendances a donc été faite. D'autre part, afin que le reste du compilateur puisse gérer les duplications, une table de hachage est passé à la traduction en code intermédiaire, donnant à chaque fonction les fonctions susceptibles d'être appelé par cette fonction (ce qui permet de savoir ce que la cellule fille doit hériter comme code lorsqu'il y a une duplication)

— *De place en mémoire et taille du code :* L'analyse statique s'occupe aussi de calculer la taille du code (un simple comptage du nombre de noeuds pour le coup) mais également de la place que va prendre en mémoire le code (gérer les variables persistance, les variables par adresse ou les simples variables, pour la traduction en code intermédiaire)

La grande majorité des fonctions crées sont basées sur l'utilisation de tables de hachages (précalculées ou calculées "au vol", qui facilitent les différentes vérifications à faire.

Transformation de l'arbre de syntaxe abstraite Cette étape avait deux enjeux principaux. Le premier consistait à attribuer à chaque variable de la mémoire d'une cellule sa position dans la mémoire et à remplacer toutes les occurences de cette variable par cette position. Le second enjeu était le traitement des séquences de programmes (&&) : il a fallu attribuer une position en mémoire à la variable utilisée, transformer la séquence de programmes en une disjonction de cas sur cette variable et positionner et faire remonter les labels utilisés. Pour

faire cela, nous utilisons une variable globale que nous incrémentons au cours du parcours des programmes ainsi que les informations sur la mémoire nécessaire à chaque programme.

Conversion vers le C++

— Les fonctions `CEL` sont converties vers des fonctions `C++`.

— Les actions effectuées par les cellules deviennent des appels de méthodes de l'environnement

— **Expressions :** Il a fallu porter attention au fait que les expressions en `CEL` doivent rester des expressions en `C++`. Par exemple, il eut été naturel en `Ocaml` de convertir l'expression y/x en

```
if x = 0 then failwith "division par zero" else y / x
```

Cependant, il faut adopter en `C++` la notation $(x == 0)?(/* \textit{sanction pour la cellule} */) : (y/x)$

— **Mémoire :** Toutes les variables utilisées par le programmeur deviennent les cases du tableau qui représente la mémoire de chaque cellule. Selon que les appels se font pas valeur ou par adresse, on envoie aux fonctions la valeur d'une case ou son numéro. Cette structure est entièrement invisible et intouchable pour le programmeur `CEL`.

— **Directions :** Il a fallu jongler avec deux conventions ayant toutes les deux leurs mérites propres. La première, destinée aux utilisateurs, consiste à considérer les directions comme des puissances de deux, $1, 2, 4, 8, 16, 32, 64, 128$, permet d'additionner des directions pour former des vecteurs de directions (très pratique pour envoyer un message). La seconde, liée au fait que les directions sont un type énuméré, les compte de 0 à 7. Nous avons donc créé un fichier `dir_managing.h` qui fait des traductions d'une convention vers l'autre.

3.7 β-test

Nous avons lancé une phase de β-test 2 semaines avant la présentation publique du projet. Nous avons pu déceler quelques buggs et ajuster les constantes de la physique du jeu. Sinon tout c'est bien déroulé.

On peut dire que le produit final passe à l'échelle !

4 Bilan

4.1 Conclusions

Quand l'idée de ce projet nous est venue pour la première fois, nous le voyions comme un projet pour une douzaine de personnes. Quand nous nous sommes rendus compte que nous ne pouvions le faire qu'à six, nous avons hésité à continuer, pensant qu'il serait très difficile d'atteindre notre objectif dans le temps imparti.

Nous avons bien fait de continuer, puisque l'état actuel de Grain de Cell est plus avancé que ce que nous osions espérer.

Tout au long du semestre, nous avons su rester motivés, notamment grâce à notre politique qui consistait à se rapprocher le plus possible à chaque étape d'un environnement fonctionnel : quitte à reporter des améliorations, nous préférions que le jeu marche plutôt que de se retrouver face à un arbre de *features* alléchant mais risquant de ne jamais aboutir faute de temps et de motivation pour développer des outils sans les voir fonctionner.

Cela n'a pas empêché l'apparition de bonds qualitatifs grâce au dynamisme de l'équipe : des prises d'initiative comme celle de l'interface OpenGL (développée sur son temps libre par Antoine) ont beaucoup contribué à faire de notre projet un succès. Des joueurs nous ont rejoint spontanément, notre site reçoit une vingtaine de visites par jour et nous sommes en contact avec une étudiante en biologie qui est intéressée par notre modélisation. Elle a été sollicitée pour proposer des améliorations afin que nous collions plus à la réalité, au moins symboliquement.

Sur les deux plans qui nous intéressaient (pédagogique et ludique), il s'agit donc d'un franc succès.

4.2 Accomplissement des objectifs

Voici un résumé rapide expliquant en quoi les objectifs ont été tenus, voire dépassés.

1. **Un langage facile mais expressif :**
   Ayant nous-même testé pendant de longues heures notre langage, il nous a paru remplir ces deux critères, et cela a été confirmé par des gens extérieurs au projet. La conception du langage, qui a été longue et s'est révélée parfois un vrai casse-tête, est une réussite.
2. **Un compilateur pour ce langage** :
   Rien à ajouter de ce point de vue là, le compilateur est fonctionnel.
3. **Un environnement capable d'ajouter des cellules dynamiquement :**
   C'est bien le cas grâce à l'utilisation de librairies dynamiques en `C++`
4. **Un serveur qui distribue la carte et reçoit de nouvelles cellules :**
   Ce serveur est en place et tourne en permanence, il fonctionne avec un système d'inscription, d'authentification et de hashage des mots de passe.
5. **Une interface graphique :**
   Non seulement nous disposons d'une interface graphique, mais elle est en 3D.

4.3 Perspectives

Dans l'avenir, nous souhaiterions développer plusieurs points afin de rendre le jeu .

À court terme Nous comptons corriger dans les prochains jours, avec l'appuie de la communauté de β-testeur, les quelques bugs encore présents.
Dans peu de temps, nous comptons organiser un tournoi. Une première partie sera consacrée à des compétitions un contre un suivies d'une deuxième partie consacrée à des affrontements chacun pour soi.
Nous souhaitons aussi permettre aux joueurs d'utiliser des librairies. Ces librairies permettraient d'utiliser des fonctions pré-construites pour les joueurs les moins initiés à la programmation.

À long terme Il serait intéressant d'étendre le langage `CEL` pour proposer de nouvelles actions pour les cellules. Enfin, nous pensons développer une interface web pour Grain de Cell. Ceci permettrait à un plus large public de nous rejoindre.
Notre objectif est de créer une réelle communauté de joueurs.

A Liste des *work-packages*

Théorique Nous devons définir les spécifications du langage codant l'ADN (le langage niveau utilisateur) ainsi que le langage niveau interpréteur (exploité par l'environnement). Nous devons également définir la forme que prendra le flot de modifications du terrain.

Environnement Nous écrivons ici la routine qui calcule la simulation et qui écrit le flot de modifications.

Compilation Dans ce *work-package*, il faut écrire un compilateur du langage niveau utilisateur vers le langage niveau-interpréteur. Il est important de préserver la structure de "sous ADN" utile à la duplication avec spécialisation de la cellule.

Connexions réseaux Concernant les connexions réseaux, il faut fixer un protocle de communication entre serveur et client, écrire le code necéssaire pour la connexion au serveur, l'écriture et la lecture de fichiers publics. Ensuite, il faut instaurer un système d'authentification : il faut pouvoir se créer un compte et s'authentifier depuis le client. Il est important de vérifier que le système est "à peu près sûr". Par exemple : on ne doit pas pouvoir lire le code de l'ADN de cellules ennemies.

Interface L'interface s'appuie sur le travail effectué sur les connexions réseaux. Il faut prévoir :

- un programme qui permet d'envoyer un code d'ADN au serveur (suivant l'ambition : un envoi simple de fichier texte ou une aide à la saisie) ;
- un programme qui pour un flot de modifications et un état initial accessibles localement affiche l'état du terrain pas à pas (ici aussi, l'affichage peut se limiter à du simple "ASCII Art" mais peut aussi être intégré à une interface graphique avec zoom etc.).

Communication

β-test Ce travail consiste à re-équilibrer le jeu en le testant. Il faut par exemple fixer les constantes des règles du jeu. Bien sûr, n'importe qui peut participer à ce travail.

A.1 Dépendances

Nous représentons les dépendances entre les différents *work-packages* dans la figure 5.

B Calendrier

Nous avons réparti la force de travail sur tous les *work-packages* en respectant les dépendances dans le diagramme de la figure 6.

On peut résumer quelques *deadlines* importantes :

1. dès le début du projet, nous mettons en place un dépot GIT (sur Gitorious), nous demandons un serveur et un nom de domaine ;
2. à la fin de la semaine 2, dans le *work-package* connexions réseaux (Antoine et Vincent), il ne reste que la partie authentification à faire et le *work-package* Théorique (Thomas, Lucca et Guillaume) est achevé ;

3. à la fin de la semaine 4, l'environnement, la première interface graphique, la compilation et les connexions réseaux fonctionnent tous dans une version allégée (tout n'est pas encore integré – uniquement les déplacements par exemple – mais les interfaces entre programmes fonctionnent) ;
4. à la fin de la semaine 6, nous devons être capable de fournir une simulation, une compilation et les fonctions réseaux (sans l'authentification) fonctionnels avec toutes les actions possibles (il ne reste plus que l'authentification et les interfaces utilisateurs) ;
5. à la fin de la semaine 8, si l'interface pose des problèmes, nous nous contentons d'une interface en ASCII art. De la semaine 8 à 10, 2 personnes se mettent à re-travailler sur l'environnement pour intégrer les fonctions réseaux ;
6. à la fin de la semaine 10, tout doit fonctionner côté programmes. Il ne reste plus qu'a équilibrer les règles : nous rentrons tous dans une phase de β-test.
7. à la fin de la semaine 16, nous pouvons rendre la version finale.

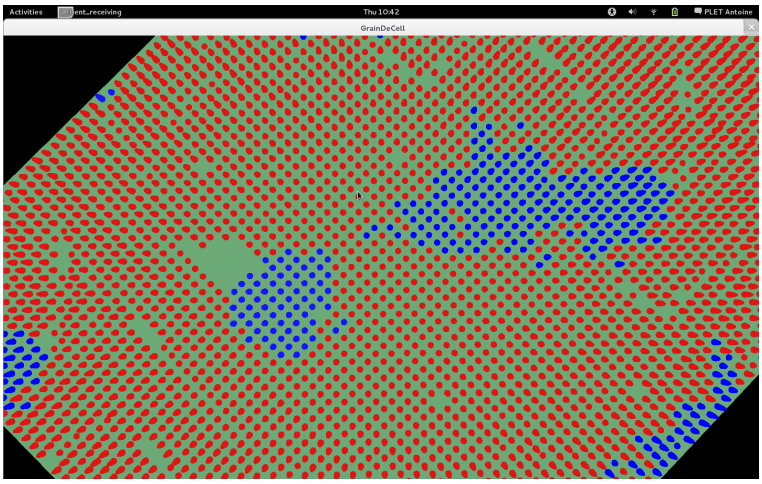

FIGURE 1 – Vue par le haut de l'Interface Graphique 3D, OpenGL

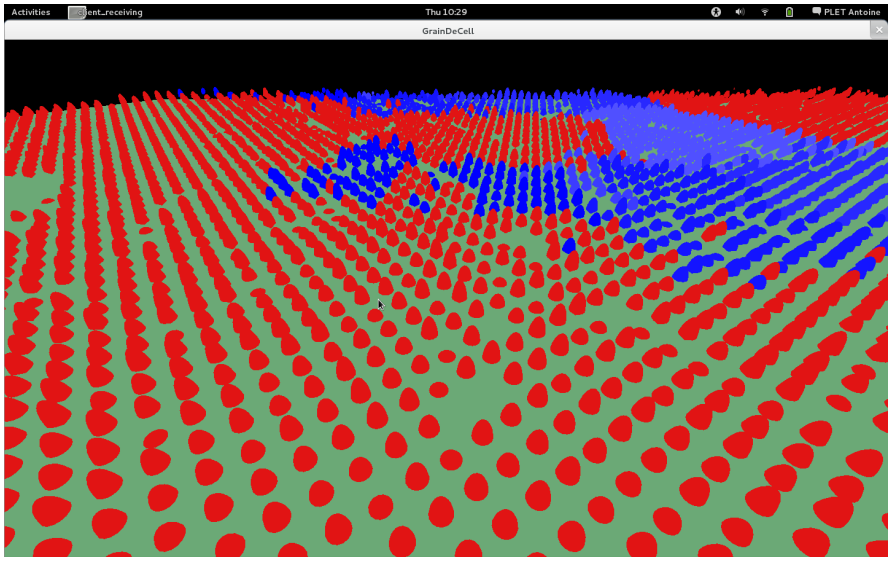

FIGURE 2 – Interface Graphique 3D, créée avec la librairie graphique OpenGL

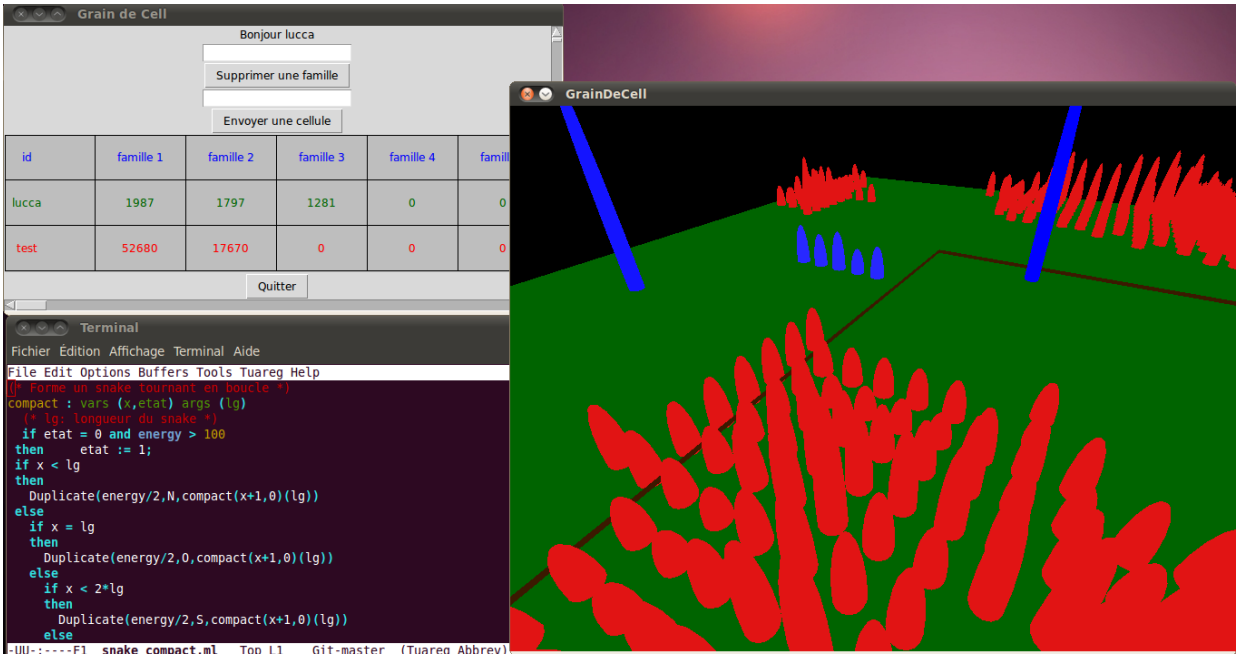

FIGURE 3 – Interface python et OpenGL réunie

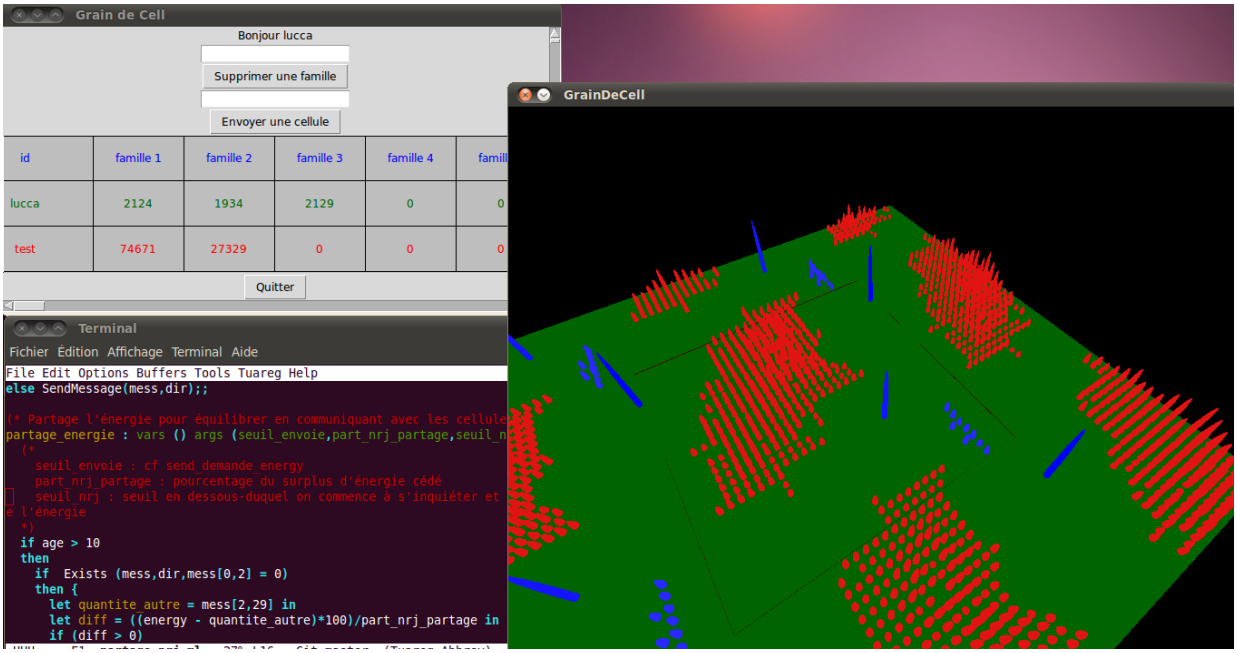

FIGURE 4 – Interface python et OpenGL réunie - autre vue

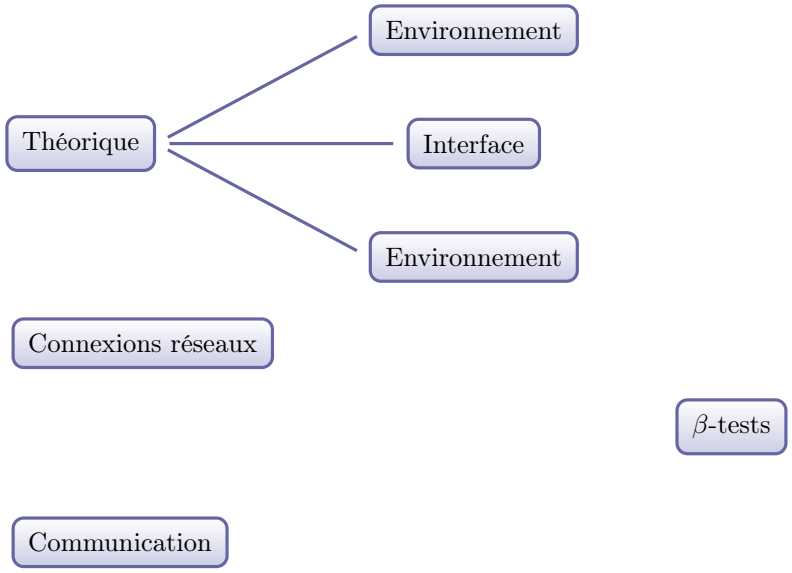

FIGURE 5 – Dépendances des *work-packages*

Diagramme de Gantt

		S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12
1	Théorique	3											
2	Réseau	2				2		2					
3	Com	1		1		1		1		1			
4	Test												6
5	Environnement			2		1,5				2			
6	Compilation			2		1,5							
7	Interface			1				3		3			
	Somme (personnes)	6		6		6		6		6		6	

Somme (personnes*2 semaines)
3
6
5
6
5,5
3,5
7

FIGURE 6 – Diagramme de Gantt (S_i représente la $i^{\text{ème}}$ semaine)